IONA

Orbix®

.NET Connector Programmer's Guide

Version 6.1, December 2003

Making Software Work Together™

Updated: 21-May-2004

M 3 1 5 1

Contents

List of Figures vii

Preface ix

Chapter 1 .NET and CORBA Frameworks 1
- **.NET versus CORBA** 2
- **CORBA Principles** 3

Chapter 2 Introduction to .NET Connector 7
- **.NET Connector Overview** 8
- **.NET Connector System Components** 11
- **.NET Client to CORBA Server Usage Model** 13

Chapter 3 Getting Started 15
- **Prerequisites** 16
- **Developing .NET Clients** 19
  - Introduction 20
  - Generating .NET Metadata from OMG IDL 21
  - Writing a Visual Basic .NET Client 22
  - Writing a C# Client 25
  - Building and Running the Client 28

Chapter 4 Client Callbacks 29
- **Introduction to Callbacks** 30
- **Implementing Callbacks** 31
  - Defining the OMG IDL Interfaces 32
  - Implementing the Client in C# 34
  - Implementing the Server in C++ 36

Chapter 5 Development Support Tools 37

Generating .NET Metadata 38

Managing the Type Store 39

The Role of the Type Store 40

The Caching Mechanism of the Type Store 42

Adding New Information to the Type Store 44

Emptying the Type Store Cache 46

Dumping the Type Store Contents 47

Chapter 6 Deploying a .NET Connector Application 49

Deployment Model 50

Deployment Steps 52

Chapter 7 Introduction to OMG IDL 55

IDL 56

Modules and Name Scoping 57

Interfaces 58

Introduction to Interfaces 59

Interface Contents 61

Operations 62

Attributes 65

Exceptions 66

Empty Interfaces 67

Inheritance of Interfaces 68

Multiple Inheritance 69

Inheritance of the Object Interface 71

Inheritance Redefinition 72

Forward Declaration of IDL Interfaces 73

Local Interfaces 74

Valuetypes 76

Abstract Interfaces 77

IDL Data Types 78

Built-in Data Types 79

Extended Built-in Data Types 82

Complex Data Types 85

Enum Data Type 86

Struct Data Type 87

Union Data Type 88

Arrays 90
Sequence 91
Pseudo Object Types 92
Defining Data Types **93**
Constants 94
Constant Expressions 97

Chapter 8 Mapping CORBA to .NET **99**
Mapping for Basic Types **101**
Mapping for Extended Types **102**
Mapping for Interfaces **103**
Mapping for Interface Inheritance **105**
Mapping for Complex Types **106**
Mapping for Structs 107
Mapping for Enums 108
Mapping for Unions 109
Mapping for Arrays 111
Mapping for Sequences 112
Mapping for System Exceptions 113
Mapping for User Exceptions 114
Mapping for the Any Type 115
Mapping for Object References **120**
Mapping for Modules **121**
Mapping for Constants **122**

Chapter 9 .NET Connector Configuration **125**
Overview **126**
Configuration Variables **127**

Chapter 10 .NET Connector Utility Arguments **131**
Itts2il Argument Details **132**
Ittypeman Argument Details **135**

Chapter 11 Advanced Topics **139**
.NET Metadata versus Type Store Information **140**
Enabling Advanced CORBA Features **142**

Index 145

List of Figures

Figure 1: Role of the ORB in Client-Server Communication 6
Figure 2: Role of .NET Connector 9
Figure 3: .NET Client to CORBA Server 13
Figure 4: .NET Connector Type Store and the Development Utilities 40
Figure 5: Overview of Typical Deployment Scenario 50
Figure 6: Overview of Deployment Steps 54
Figure 7: Inheritance Hierarchy for PremiumAccount Interface 70
Figure 8: .NET Metadata and Dynamic Type Information Usage 140
Figure 9: CORBA Features as Plug-Ins to Remoting Channel 142

Preface

The .NET Connector provides a high performance bridge that enables transparent communication between .NET clients and CORBA servers. It is designed to allow .NET programmers who use any .NET language (for example, Visual Basic .NET, C#, J#, and so on) to easily access CORBA applications running in Windows, UNIX, or OS/390 environments. It means that .NET programmers can use the tools familar to them to build heterogeneous systems that use both .NET and CORBA components within a .NET environment.

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com.

Audience

This guide is intended for .NET application programmers who want to use the .NET Connector to develop and deploy distributed applications that combine CORBA and .NET components within a .NET environment. This guide assumes that the reader already has a working knowledge of .NET-based tools, such as Visual Basic .NET and C#.

Required versions

To use the .NET Connector, you need at least Microsoft .NET Framework 1.1 and Microsoft Visual Studio .NET 2003 installed on your machine.

Organization of this guide

This guide is divided as follows:

Chapter 1, ".NET and CORBA Frameworks"

Both .NET (more specifically .NET Remoting) and CORBA are recognised as industry-standard frameworks for distributed object computing. This chapter introduces comparisons between these two frameworks. It also provides an introductory overview of CORBA and its main principles for the sake of novice CORBA users.

Chapter 2, "Introduction to .NET Connector"

IONA's .NET Connector enables transparent communication between clients that are running in a Microsoft .NET environment and servers that are running in a CORBA environment. This chapter introduces the .NET Connector, first by outlining the distributed component concepts supported by .NET, and then by describing how the .NET Connector implements these concepts.

Chapter 3, "Getting Started"

This chapter is provided as a means to getting started quickly in application programming with the .NET Connector. It explains the basics you need to know to develop a simple .NET client, written in Visual Basic .NET or C#, which can call objects in an existing CORBA server.

Chapter 4, "Client Callbacks"

The typical .NET Connector scenario involves .NET clients invoking operations on objects in CORBA servers. However, .NET clients can implement some of the functionality associated with servers, and all servers can act as clients. A callback invocation is a programming technique that takes advantage of this. This chapter describes how to implement client callbacks.

Chapter 5, "Development Support Tools"

This chapter describes how to use the itts2il utility to generate .NET metadata from existing OMG IDL and perform various type store management tasks.

Chapter 6, "Deploying a .NET Connector Application"

This chapter provides an overview of the deployment model you can adopt when deploying a distributed application with the .NET Connector. It also describes the steps you must follow to deploy a distributed .NET Connector application.

Chapter 7, "Introduction to OMG IDL"

An object's interface describes that object to potential clients through its attributes and operations, and their signatures. This chapter describes the semantics and uses of the CORBA Interface Definition Language (OMG IDL), which is used to describe the interfaces to CORBA objects.

Chapter 8, "Mapping CORBA to .NET"

CORBA types are defined in OMG IDL, and .NET types are defined in Microsoft Intermediate Language (MSIL). To allow interworking between .NET clients and CORBA servers, .NET clients must be presented with metadata that describes the interfaces exposed by CORBA objects. Therefore, it must be possible to translate CORBA types to .NET types. When using .NET Remoting, the .NET types must use the .NET Common Type System (CTS). This chapter outlines the CORBA-to-.NET CTS mapping rules.

Chapter 9, ".NET Connector Configuration"

This chapter describes the configuration variables that are specific to the .NET Connector, and their associated values.

Chapter 10, ".NET Connector Utility Arguments"

This chapter describes the various arguments that are available with the ittypeman and itts2il command-line utilities.

Chapter 11, "Advanced Topics"

This chapter provides details of topics that might be of interest to more advanced users of the .NET Connector, including an explanation of the difference between static .NET metadata and dynamic runtime type information, and a description of how to programatically enable advanced CORBA features.

Related reading

The following related reading material is recommended:

- The Common Object Request Broker: Architecture and Specification at `http://www.omg.org/docs/formal/01-09-01.pdf`.

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts, about Orbix and other products. You can access the knowledge base at the following location:

http://www.iona.com/support/knowledge_base/

The IONA update center contains the latest releases and patches for IONA products:

http://www.iona.com/support/updates/

The IONA newsgroup and discussion forums provide feedback and answers to questions about IONA products:

http://www.iona.com/products/newsgroups.htm

Typographical conventions

This guide uses the following typographical conventions:

`Constant width` — Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

Italic — Italic words in normal text represent *emphasis* and *new terms*.

Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

% cd /users/*your_name*

Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, Windows 95, or Windows 98 command prompt.
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

.NET and CORBA Frameworks

Both .NET (more specifically .NET Remoting) and CORBA are recognised as industry-standard frameworks for distributed object computing. This chapter introduces comparisons between these two frameworks. It also provides an introductory overview of CORBA and its main principles for the sake of novice CORBA users.

In this chapter

This chapter discusses the following topics:

.NET versus CORBA	page 2
CORBA Principles	page 3

Note: A knowledge of CORBA is not a prerequisite for using the .NET Connector. This is because the .NET Connector abstracts the details of CORBA from the .NET programmer. Unless you are using advanced CORBA features, no knowledge of the CORBA server is required apart from its contact details. This chapter is provided for reference purposes only. An in-depth study of .NET or CORBA is outside the scope of this guide.

.NET versus CORBA

Overview

.NET and CORBA are both industry-standard frameworks for distributed object computing. Both share the common goals of:

- Enabling interoperability of distributed applications written in heterogeneous languages.
- Allowing for modifications to the implementation of objects in a particular language without the need for changes to other objects implemented in other languages.

This section provides an introductory comparison of .NET and CORBA concepts.

Comparison

Table 1 provides an introductory comparison of .NET and CORBA concepts.

Table 1: *Comparison of .NET and CORBA Concepts*

.NET	CORBA
.NET metadata and Microsoft Intermediate Language (MSIL) provide language-independent definitions of .NET object interfaces.	Interface Definition Language (IDL) provides language-independent definitions of CORBA object interfaces.
.NET metadata is stored in a .NET assembly.	IDL is stored in an Interface Repository.
Common Type System specifies types that have mappings to various language implementations.	Standardized IDL type mappings exist for various language implementations.
Common Language Runtime uses metadata to marshal requests between distributed applications.	Object Request Broker (ORB) runtime uses IDL to marshal requests between distributed applications.
.NET Remoting Channel used for communication. (For example, Microsoft-proprietary binary protocol over TCP transport.)	Standard protocols used for communication. (For example, Internet Inter-ORB Protocol (IIOP) over TCP/IP transport).

CORBA Principles

Overview

This section provides an introductory overview of the main principles of CORBA for novice CORBA users. It discusses the following topics:

- "Basic principles" on page 3.
- "CORBA objects" on page 4.
- "Object IDs and references" on page 4.
- "CORBA object interfaces" on page 4
- "CORBA client requests" on page 4.
- "CORBA object lifetime" on page 5.
- "Object request broker" on page 5.
- "Multiple inheritance" on page 6.

Note: A knowledge of CORBA is not a prerequisite for using the .NET Connector. This is because the .NET Connector abstracts the details of CORBA from the .NET programmer. Unless you are using advanced CORBA features, no knowledge of the CORBA server is required apart from its contact details. This section is provided for reference purposes only. A more in-depth study of CORBA is outside the scope of this guide.

Basic principles

Some of the basic principles of CORBA are:

- The system architecture is based around the concept of objects.
- An object is a discrete unit of functionality that exposes its behavior through a set of well defined interfaces.
- The details of an object's implementation are hidden from the clients that want to make requests on it.
- An object is an independent component with a related set of behaviors, transparently available to any CORBA client, regardless of where the object or client are implemented in the system.
- The domain of an object is typically an arbitrarily scalable distributed network.

- The purpose of CORBA is to allow independent components of a distributed system to be shared among a wide variety of possibly unrelated applications and objects in that distributed system.

CORBA objects

A CORBA object is a discrete, independent unit of functionality, comprising a related set of behaviors. A particular CORBA object can be described as an entity that exhibits a consistency of interface, behavior (or functionality), and state over its lifetime.

CORBA uses the concept of a portable object adapter (POA), which is used to map abstract CORBA objects to their actual implementations. A CORBA object can be implemented in any programming language that CORBA supports, such as C++ or Java.

Object IDs and references

A CORBA object has both an object ID and an object reference. An object ID identifies an object with respect to a particular POA instance. An object reference contains unique details about an object, including its object ID and POA identifier, which can be used by clients to locate and invoke on that object. See "CORBA client requests" on page 4 for more details about the use of object references.

CORBA object interfaces

A CORBA object presents itself to its clients through a published interface, defined in OMG interface definition language (IDL). The concept of keeping an object's interface separate from its implementation means that a client can make requests on an object without needing to know how or where that object is implemented.

The IDL interfaces for CORBA objects can be stored (registered) in an interface repository. CORBA identifies an interface by means of an interface repository ID. Even if you update a particular interface in some way, its repository ID can remain the same.

CORBA client requests

In CORBA, a client can access an object's interface and its underlying functionality by making one or more requests on that object. Each client request is made on a specific instance of an object, which is identifiable and contactable via an object reference that is unique to that object instance. An object reference is a name that is used to consistently identify a particular object during that object's lifetime. An object reference in CORBA is roughly equivalent to the concept of an object reference in .NET.

CORBA client requests can contain parameters consisting of object references or data values that correspond to particular types of data supported by the system. A client request can be dynamically created at runtime (rather than simply being statically defined at compile time) on any object whose interfaces are stored in an interface repository.

CORBA object lifetime

The in-memory lifetime of a CORBA object is independent of the lifetime of any clients that hold a reference to it. This means that a client that is no longer running can continue to maintain object references. It also means that a server object can deactivate and remove itself from memory when it becomes idle (although this does consequently mean that the server application must be made to explicitly decide when this should happen).

Object request broker

A CORBA system is based on an architectural abstraction called the object request broker (ORB). An ORB allows for:

- Interception and transfer of client requests to servers across the network, and the return of output from the server back to the client.
- Registration of data types and their interfaces, defined in OMG IDL.
- Registration of object instance identities, from which the ORB can construct appropriate object references for use by clients that want to make requests on those object instances.
- Location (and activation, if necessary) of objects.

Orbix is IONA's implementation of an ORB.

Figure 1 provides an overview of the role of the ORB in CORBA client-server communication.

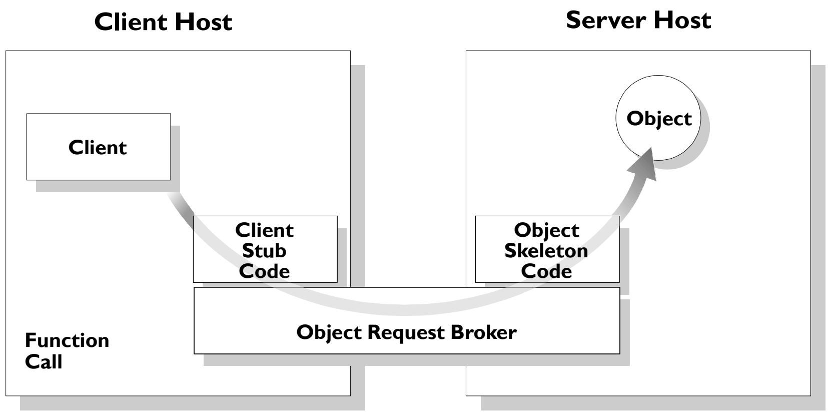

Figure 1: *Role of the ORB in Client-Server Communication*

Multiple inheritance

CORBA supports the concept of multiple interface inheritance. This basically means that a CORBA object interface can be extended by making it derive from one or more other interfaces. The derived interface ends up having not only its own defined functionality, but also the functionality of the interface(s) from which it derives. Interfaces can also be evolved, by having new interfaces derive from existing interfaces.

A CORBA object reference refers to a CORBA object that exposes a single, most-derived interface in which any and all parent interfaces are joined. CORBA does not support the concept of objects with multiple, disjoint interfaces. See “Introduction to OMG IDL” on page 55 for more details of multiple inheritance.

Introduction to .NET Connector

IONA's .NET Connector enables transparent communication between clients that are running in a Microsoft .NET environment and servers that are running in a CORBA environment. This chapter introduces the .NET Connector, first by outlining the distributed component concepts supported by .NET, and then by describing how the .NET Connector implements these concepts.

In this chapter

This chapter discusses the following topics:

.NET Connector Overview	page 8
.NET Connector System Components	page 11

Note: The .NET Connector supports development and deployment of .NET clients that can communicate with CORBA servers. Any CORBA C++ server examples provided in this guide are supplied for reference purposes only. It is assumed that you already have a CORBA server implementation product. The examples provided are for use with Orbix 6.1

.NET Connector Overview

Overview

This section provides an introductory overview of the .NET Connector in terms of how it facilitates communication between .NET clients and CORBA servers. The following topics are discussed:

- "What is the .NET Connector?" on page 8.
- "Note for existing COMet users" on page 8.
- "Graphical Overview of Role" on page 9.
- "Advantages for the .NET Programmer" on page 9.
- "Supported Protocols" on page 10.

What is the .NET Connector?

The .NET Connector is a custom .NET remoting channel, referred to as `Orbix.Remoting`, from IONA Technologies. Its purpose is to support application integration across network boundaries, different operating systems, and different programming languages. Specifically, it provides a high performance bridge that enables integration between .NET clients and CORBA objects. It allows you to develop and deploy .NET client applications that can interact with existing CORBA server applications that might be running on Windows or another platform.

Note for existing COMet users

Users of IONA's COMet product should note that the .NET Connector is the natural upgrade solution if you are planning to migrate from a COM-based development environment to a .NET-based development environment.

The .NET Connector provides the same or similar advantages to the .NET programmer that COMet provides to the COM/Automation programmer. For example:

- It provides enterprise CORBA scalability to the .NET environment.
- It facilitates development and deployment of .NET client applications that can call CORBA servers, with no presumption of CORBA knowledge on the .NET programmers' part, and with no need for modifications to the CORBA server-side implementation.
- It enables dynamic runtime mapping between CORBA and .NET types (in this case, via supplied utilties that generate required .NET metadata from the OMG IDL defined for the target CORBA objects).

Graphical Overview of Role

Figure 2 provides a conceptual overview of how the .NET Connector facilitates integration of .NET clients and CORBA servers.

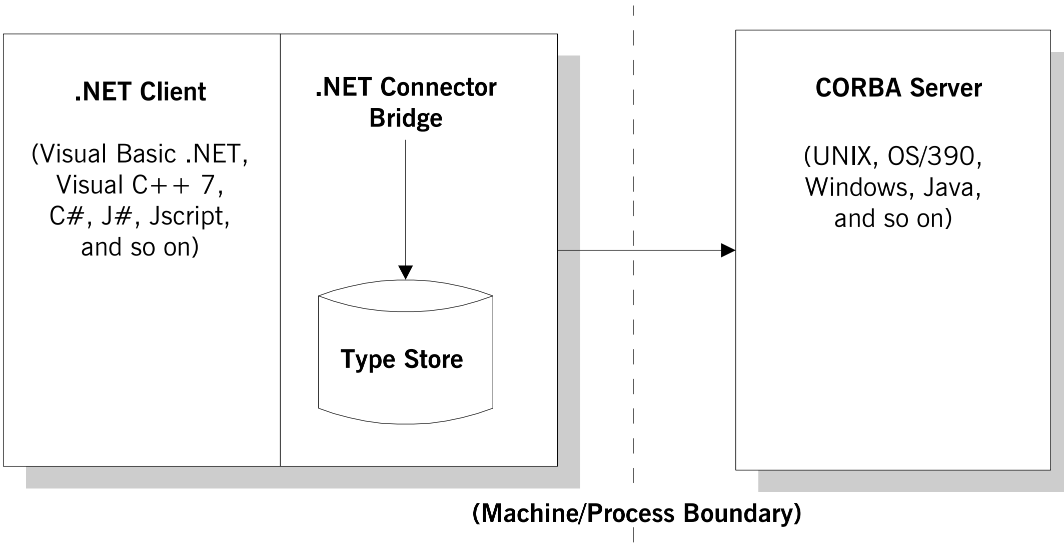

Figure 2: *Role of .NET Connector*

Advantages for the .NET Programmer

The .NET Connector provides two main advantages to .NET programmers:

1. The .NET Connector provides access to existing CORBA servers, which can be implemented on any operating system and in any language supported by a CORBA implementation. Orbix supports a range of operating systems, such as Windows, UNIX, and OS/390. It also supports different programming languages, including C++ and Java.
2. Using the .NET Connector, a .NET programmer can use familiar .NET-based tools to build heterogeneous systems that use both .NET and CORBA components within a .NET environment. The .NET Connector, therefore, presents a programming model that is familiar to the .NET programmer.

Supported Protocols

The .NET Connector supports any protocol that Orbix supports (for example, IIOP, SSL-IIOP, UDP (EGMIOP)). This means that a .NET application can interact with an ORB that uses any of these protocols.

.NET Connector System Components

Overview

This section describes the various components that comprise a .NET Connector system. The following topics are discussed:

- "Bridge" on page 11.
- "Type Store" on page 11.
- ".NET Client" on page 12.
- "CORBA Server" on page 12.

Bridge

The bridge is a synonym for the .NET Connector itself. It is implemented as a custom remoting channel, referred to as `Orbix.Remoting`. It is implemented in a mixture of managed and umanaged C++. This channel uses a dynamic marshaller and type store to formulate dynamic requests that can be invoked on the CORBA server. The bridge provides the mappings and performs the necessary translation between .NET common type system (CTS) and CORBA types.

The bridge is used in conjunction with a .NET Connector utility, called `itts2il`, which generates .NET metadata from OMG IDL.

The bridge allows .NET clients to take advantage of all the CORBA services that are available to an ordinary C++ client, such as security and portable interceptors.

Type Store

As shown in Figure 2 on page 9, the .NET Connector uses a component called the type store. The type store is used to hold a cache of information about all the CORBA types in your system. The .NET Connector can retrieve this information from the Interface Repository (IFR) at application runtime, and then automatically update the type store with this information for subsequent use, instead of having to query the IFR for it again. See "The Caching Mechanism of the Type Store" on page 42 and ".NET Metadata versus Type Store Information" on page 140 for more details about the type store.

.NET Client	A .NET client can use the .NET Connector to communicate with a CORBA server. This client can be written in a language such as Visual Basic .NET, Visual C++, C#, J#, Jscript, or any other .NET-compatible language.
CORBA Server	A CORBA server can be contacted by .NET clients, using the .NET Connector. This is a normal CORBA server written in any language and running on any platform supported by an ORB.

.NET Client to CORBA Server Usage Model

Overview

This section describes the typical usage model supported by the .NET Connector—a .NET client communicating with a CORBA server. It discusses the following topics:

- "Graphical overview" on page 13.
- ".NET client and bridge" on page 14.
- "CORBA server" on page 14.

Graphical overview

Figure 3 shows a graphical overview of this usage model.

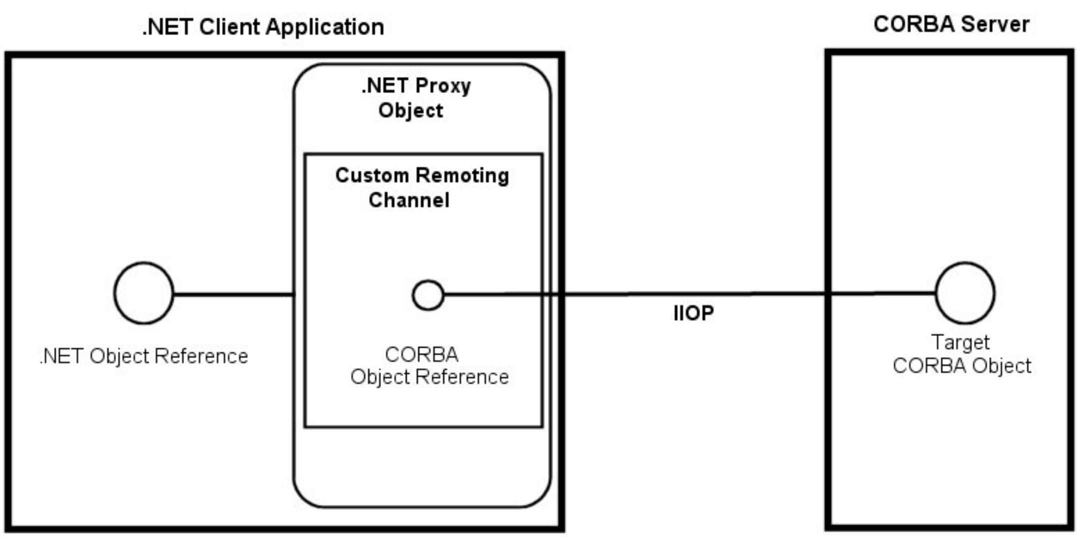

Figure 3: *.NET Client to CORBA Server*

.NET client and bridge

A dynamic bridge for .NET is provided by a custom remoting channel, referred to as `Orbix.Remoting`. The .NET client loads this bridge in-process (that is, in the client's address space). This involves the use of IIOP as the wire protocol for communication between the .NET client machine and CORBA server.

The .NET client registers the `Orbix.Remoting` custom remoting channel. The .NET client then creates a proxy object for the remote CORBA object. The .NET client can subsequently make calls on this proxy object as if it were a local .NET object. The proxy object uses the `Orbix.Remoting` channel to make a corresponding call on the target object in the CORBA server.

Because the `Orbix.Remoting` channel exposes mapped .NET types as metadata contained in a .NET assembly, automatic mapping of .NET object references to CORBA interfaces and object references at runtime is enabled.

The client does not need to know that the target object is a CORBA object. A .NET client can be written in Visual Basic .NET, C#, J#, or any language that supports the .NET runtime.

CORBA server

The CORBA server presents an OMG IDL interface to its objects. The server application can exist on platforms other than Windows. It can be written in any language supported by a CORBA implementation, such as C++ or Java.

Getting Started

This chapter is provided as a means to getting started quickly in application programming with the .NET Connector. It explains the basics you need to know to develop a simple .NET client, written in Visual Basic .NET or C#, which can call objects in an existing CORBA server.

In This Chapter

This chapter discusses the following topics:

Prerequisites	page 16
Developing .NET Clients	page 19

Prerequisites

Overview

This section describes the prerequisites to starting application development with the .NET Connector. The following topics are discussed:

- "Required versions" on page 16.
- "Client-Side Requirements" on page 16.
- "Server-Side Requirements" on page 17.
- "Registering OMG IDL Type Information" on page 17.
- "Adding .NET Connector to the Global Assembly Cache" on page 17.
- "Making .NET Connector Available to Add References dialog" on page 18.

Required versions

To use the .NET Connector, you need at least Microsoft .NET Framework 1.1 and Microsoft Visual Studio .NET 2003 installed on your machine.

Required runtime libraries

In this release of the .NET Connector, the .NET framework requires that the following Visual C++ 7.1 runtime libraries are installed:

- `msvcr71.dll`
- `msvcp71.dll`

Client-Side Requirements

Ensure that both Orbix and the .NET Connector are installed and configured correctly. Ensure that all required options are installed. See the Orbix 6.1 *Installation Guide* for more details about installation. See the Orbix 6.1 *Administrator's Guide* for details about configuring both Orbix and the .NET Connector.

> **Note:** An Interface Repository (IFR) service must be configured when setting up your configuration domain, to allow the .NET type store to obtain the OMG IDL type information it requires. It is sufficient to deploy only one centralized IFR server on your network. You do not need to have an IFR service installed on each client machine.

Server-Side Requirements

The .NET Connector requires no changes to existing CORBA servers. See the Orbix documentation set for details of how to manage servers. This chapter assumes that you are using Orbix as your server-side object request broker (ORB), but any CORBA-compliant ORB can be used on the server side.

Registering OMG IDL Type Information

As explained in "Introduction to .NET Connector" on page 7, the .NET Connector uses a custom remoting channel between .NET clients and CORBA servers. The bridge is driven by OMG IDL type information derived from a CORBA Interface Repository (IFR).

Before you run an application, ensure that your OMG IDL is registered in the IFR. This is because the .NET custom remoting channel is designed to automatically retrieve the required type information from the IFR at application runtime. The .NET Connector then saves this information to the type store for subsequent use. See "Registering OMG IDL" on page 21 for more details of how to do this.

Adding .NET Connector to the Global Assembly Cache

As explained in ".NET client and bridge" on page 14, the .NET Connector is implemented as a custom remoting channel in managed C++. This custom remoting channel is called `Orbix.Remoting` and is contained in the `Orbix.Remoting.dll` assembly. To use the `Orbix.Remoting` channel, the .NET framework must be able to obtain and access the `Orbix.Remoting.dll` assembly from either of the following:

- The directory from which the client program is run.
- The Global Assembly Cache (GAC).

By default, the supplied demonstrations are configured to use a local copy of the `Orbix.Remoting` channel.

If you want to register the `Orbix.Remoting` channel with the GAC, do either of the following:

- Register the channel from the command line, by entering the following command (where *`install-dir`* represents the full path to your Orbix installation):

```
gacutil -I install-dir\bin\Orbix.Remoting.dll
```

- Register the channel graphically, as follows:
  i. Select **Settings|Control Panel|Administrative Tools|.NET Framework 1.1 Configuration** from your Windows **Start** menu.
  ii. Right-click **Assembly Cache**.
  iii. Click **Add**.
  iv. Browse to *`install-dir`*`\bin\Orbix.Remoting.dll`.
  v. Click **Open**.

Note: Adding the .NET Connector to the GAC is not mandatory. The advantage to doing it is that it means you do not need to copy the `Orbix.Remoting.dll` assembly to your client program directory. If the user has administrative rights, Orbix attempts to register the `Orbix.Remoting` channel automatically in the GAC.

Making .NET Connector Available to Add References dialog

When you are adding a reference in Visual Studio .NET, you are presented with an **Add References** dialog that contains a list of references from which you can choose. The displayed list is determined from the sub-keys (and their properties) corresponding to the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\
    AssemblyFolders
```

If you want to add the .NET Connector to this list:

1. Add the following registry key:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\
       AssemblyFolders\Orbix.Remoting
   ```

2. In the preceding key, set its default value to *`install-dir`*`\bin` (where *`install-dir`* represents the full path to your Orbix installation.

Note: Making the .NET Connector available to the **Add References** dialog is not mandatory. The advantage to doing it is that it means you do not need to search the hard disk for the `Orbix.Remoting.dll` assembly.

Developing .NET Clients

Overview

This section describes how to use the .NET Connector to develop .NET clients in Visual C++, C#, and Visual Basic .NET.

In This Section

This section discusses the following topics:

Introduction	page 20
Generating .NET Metadata from OMG IDL	page 21
Writing a Visual Basic .NET Client	page 22
Writing a C# Client	page 25
Building and Running the Client	page 28

Introduction

Overview

This subsection provides an introduction to the .NET client demonstrations provided. The following topics are discussed:

- "The grid demonstration" on page 20.
- "OMG IDL Grid interface" on page 20.
- "Location of .NET client demonstration source files" on page 20.

The grid demonstration

The examples developed in this section are .NET clients, written in Visual Basic .NET and C#, which can access and modify values that are assigned to cells within a grid that is implemented as an object in a supplied CORBA server.

OMG IDL Grid interface

The `Grid` object in the CORBA server implements the following OMG IDL `Grid` interface:

```
// OMG IDL
interface Grid {
     readonly attribute short height;
     readonly attribute short width;
     void set(in short n, in short m, in long value);
     long get(in short n, in short m);
};
```

Location of .NET client demonstration source files

The source code for the Visual Basic .NET demonstration client is in *`install-dir`*`\asp\6.1\demos\common\dotnet\grid\vb_client`, where *`install-dir`* represents the Orbix installation directory.

The source code for the C# demonstration client is in *`install-dir`*`\asp\6.1\demos\common\dotnet\grid\csharp_client`, where *`install-dir`* represents the Orbix installation directory.

Generating .NET Metadata from OMG IDL

Overview

The first step in implementing a .NET client that can communicate with a CORBA server is to generate the .NET metadata that describes the target CORBA interface. This in turn provides the .NET clients with a familar interface to the remote CORBA objects. This subsection provides an overview of .NET metadata and how to generate it.

.NET metadata

.NET metadata is required so that .NET applications that are to make invocations on remote objects can be compiled, and to allow .NET to create proxy objects. Ordinarily, when .NET applications are communicating with each other, the metadata for .NET objects can be found as part of the .NET assembly. However, this is obviously not the case for CORBA objects. Therefore, the .NET Connector provides an `itts2il` utility that allows you to generate .NET metadata based on OMG IDL type information for CORBA objects. The `itts2il` utility generates a .NET assembly in which the generated metadata is contained.

Registering OMG IDL

Before you attempt to use `itts2il` to create .NET metadata from the OMG IDL for your target CORBA objects, you must ensure that the OMG IDL is registered with the IFR. This is because `itts2il` reads the OMG IDL information from the IFR. For example, the following command registers the OMG IDL defined in the `grid.idl` file with the IFR:

```
idl -R grid.idl
```

Generating .NET metadata

The following `itts2il` command, for example, generates a .NET metadata assembly within a `Grid.dll` file, based on the OMG IDL `Grid` interface:

```
itts2il Grid
```

See “Development Support Tools” on page 37 for more details about `itts2il` and creating .NET metadata from OMG IDL.

Writing a Visual Basic .NET Client

Overview

This subsection describes the steps to develop a simple Visual Basic .NET client of a CORBA server. The steps are:

Step	Action
1	Read IOR for the target object from file.
2	Register the required remoting channel.
3	Create proxy object for client invocations.
4	Invoke operations on target object.

Step 1—Read IOR from file

Start by reading the interoperable object reference (IOR) for the target object, which is contained in the .ref file that the user specifies on the command line when starting the client. The following code raises an error if the .ref file is not specified on the command line.

```
' Visual Basic .NET
If args.Length < 1 Then
     Throw New System.Exception("IOR filename not specified on
        command line")
End If

Dim iorFile As New StreamReader(args(0))
Dim ior As String = iorFile.ReadToEnd()
iorFile.Close()
```

Step 2—Register remoting channel

The following line registers the remoting channel that the client wants to use. The custom remoting channel should be registered in the same way as any other .NET remoting channel.

```
' Visual Basic .NET
ChannelServices.RegisterChannel(New OrbixClientChannel)
```

The preceding code tells the .NET application that when it is attempting to access an object outside of its application domain, it should attempt to use the `OrbixClientChannel` remoting channel.

Step 3—Create proxy object

The following code creates a proxy instance of the remote target object in the client's address space.

```
' Visual Basic .NET
Dim GridObj As Grid = CType(Activator.GetObject(GetType(Grid),
    ior), Grid)
```

The call to `GetObject()` specifies the name of the target object to which the client wants to connect (in this case, `Grid`). The main difference between the preceding call example and a call to a native .NET object is that instead of passing an object URL to the call, the client must instead pass an IOR, a corbaloc reference, or a Naming Service reference. The call to `GetObject()` creates both the proxy object and an `OrbixClientChannelSink` channel sink.

The channel sink parses the reference (that is, IOR, corbaloc reference, or Naming Service reference) passed by the client and creates a CORBA object reference either by:

- Using `string_to_object()`, if an IOR or corbaloc reference has been passed.
- Resolving the Naming Service reference, if a Naming Service reference has been passed.

Note: An alternative way of creating the proxy object (instead of calling `Activator.GetObject()`) is to use the `new` operator for the .NET `Grid` type. In this case, the reference must be specified in the application's configuration file. This alternative approach is useful in that it allows you to dynamically specify the reference at deployment time, rather than statically at compile time.

Step 4—Invoke operations on target object

Now that the proxy object has been created, the following code obtains the width and height of the grid, and then sets a particular element of it to a particular value (in this case, it sets the element in row 2 column 4 to the value 123).

```
' Visual Basic .NET
Dim height As Short = GridObj.height
Dim width As Short = GridObj.width
Console.WriteLine("Grid’s size : " & height & " x " & width)

…
Console.WriteLine("Set element 2 x 4 to 123")
GridObj.set(2, 4, 123)
Dim 1_Value As Int32 = GridObj.get(2, 4)
Console.WriteLine("2 x 4 Element’s value : " & 1_Value)
If (1_Value = 123) Then
    Console.WriteLine("Demo succeeded")
Else
    Console.WriteLine("Demo failed, incorrect value returned")
End If
```

Writing a C# Client

Overview

This subsection describes the steps to develop a simple C# client of a CORBA server. The steps are:

Step	Action
1	Read IOR for the target object from file.
2	Register the required remoting channel.
3	Create proxy object for client invocations.
4	Invoke operations on target object.

Step 1—Read IOR from file

Start by reading the interoperable object reference (IOR) for the target object, which is contained in the .ref file that the user specifies on the command line when starting the client. The following code raises an error if the .ref file is not specified on the command line.

```
// C#
if (args.Length < 1)
     throw new Exception("IOR filename not specified");
     string url;
     using (StreamReader iorFile = new StreamReader(args [0]))
     {
         url = iorFile.ReadToEnd();
     }
```

Step 2—Register remoting channel

The following line registers the remoting channel that the client wants to use. The custom remoting channel should be registered in the same way as any other .NET remoting channel.

```
// C#
ChannelServices.RegisterChannel(new OrbixClientChannel());
```

The preceding code tells the .NET application that when it is attempting to access an object outside of its application domain, it should attempt to use the `OrbixClientChannel` remoting channel.

Step 3—Create proxy object

The following code creates a proxy instance of the remote target object in the client's address space.

```
// C#
Grid GridObj = (Grid) Activator.GetObject(typeof (Grid), url);
```

The call to `GetObject()` specifies the name of the target object to which the client wants to connect (in this case, `Grid`). The main difference between the preceding call example and a call to a native .NET object is that instead of passing an object URL to the call, the client must instead pass an IOR, a corbaloc reference, or a Naming Service reference. The call to `GetObject()` creates both the proxy object and an `OrbixClientChannelSink` channel sink.

The channel sink parses the reference (that is, IOR, corbaloc reference, or Naming Service reference) passed by the client and creates a CORBA object reference either by:

- Using `string_to_object()`, if an IOR or corbaloc reference has been passed.
- Resolving the Naming Service reference, if a Naming Service reference has been passed.

Note: An alternative way of creating the proxy object (instead of calling `Activator.GetObject()`) is to use the `new` operator for the .NET `Grid` type. In this case, the reference must be specified in the application's configuration file. This alternative approach is useful in that it allows you to dynamically specify the reference at deployment time, rather than statically at compile time.

Step 4—Invoke operations on target object

Now that the proxy object has been created, the following code obtains the width and height of the grid, and then sets a particular element of it to a particular value (in this case, it sets the element in row 2 column 4 to the value 123).

```
// C#
Int16 height = GridObj.height;
Int16 width = GridObj.width;
Console.WriteLine("Grid’s size : " + height + " x " + width);

…
Console.WriteLine("Set element 2 x 4 to 123");
GridObj.set(2, 4, 123);
Int32 1_Value = GridObj.get(2,4);
Console.WriteLine("2 x 4 Element’s value : " + 1_Value);
if (1_Value == 123) Then
    Console.WriteLine("Demo succeeded");
else
    Console.WriteLine("Demo failed, incorrect value returned");
```

Building and Running the Client

Overview

This subsection describes how to build and run the supplied grid client demonstrations. It discusses the following topics:

Building and running the Visual Basic .NET client

The steps to build and run the Visual Basic .NET grid client demonstration are:

1. Navigate to the `install-dir\asp\6.1\demos\common\dotnet\grid\vb_client\bin` directory.
2. Enter `nmake`.
3. Enter `vb_client ..\..\grid.ref`.

Note: An alternative way to build the client is to open the Visual Basic .NET project files in the Visual Studio IDE and perform the build from there.

Building and running the C# client

The steps to build and run the C# grid client demonstration are:

1. Navigate to the `install-dir\asp\6.1\demos\common\dotnet\grid\csharp_client` directory.
2. Enter `nmake`.
3. Enter `csharp_client ..\..\..\grid.ref`.

Note: An alternative way to build the client is to open the C# project files in the Visual Studio IDE and perform the build from there.

Output

The output from the demonstrations is as follows:

```
Grid's size : 5 x 5
Set element 2 x 4 to 123
2 x 4 Element's value :123
Demo succeeded
```

Client Callbacks

The typical .NET Connector scenario involves .NET clients invoking operations on objects in CORBA servers. However, .NET clients can implement some of the functionality associated with servers, and all servers can act as clients. A callback invocation is a programming technique that takes advantage of this. This chapter describes how to implement client callbacks.

In this chapter

This chapter discusses the following topics:

Introduction to Callbacks	page 30
Implementing Callbacks	page 31

Introduction to Callbacks

Overview

This chapter introduces the concept of client callbacks. The following topics are discussed:

- "What is a callback?" on page 30.
- "Typical use" on page 30.

What is a callback?

A callback is an operation invocation made from a server to an object that is implemented in a client. A callback allows a server to send information to clients without forcing clients to explicitly request the information.

Typical use

Callbacks are typically used to allow a server to notify a client to update itself. For example, in a banking application, clients might maintain a local cache to hold the balance of accounts for which they hold references. Each client that uses the server's account object maintains a local copy of its balance. If the client accesses the balance attribute, the local value is returned if the cache is valid. If the cache is invalid, the remote balance is accessed and returned to the client, and the local cache is updated.

> **Note:** The .NET Connector bridge holds an Orbix proxy object for each .NET object.

When a client makes a deposit to, or withdrawal from, an account, it invalidates the cached balance in the remaining clients that hold a reference to that account. These clients must be informed that their cached value is invalid. To do this, the real account object in the server must notify (that is, call back) its clients whenever its balance changes.

Implementing Callbacks

Overview

This section describes how to implement callbacks.

In this section

This section discusses the following topics:

Defining the OMG IDL Interfaces	page 32
Implementing the Client in C#	page 34
Implementing the Server in C++	page 36

Note: A demonstration that implements callback functionality is provided in `install-dir\asp\6.1\demos\common\dotnet\callback`, where `install-dir` represents your Orbix installation directory.

Defining the OMG IDL Interfaces

Overview

This section describes the first step in implementing client callback functionality, which is to define the OMG IDL interfaces for the server objects and client objects. The following topics are discussed:

- "Client interface example" on page 32.
- "Client interface explanation" on page 32.
- "Server interface example" on page 32.
- "Server interface explanation" on page 33.

Client interface example

The client implements an IDL interface that the server uses to call back clients. A suitable IDL interface for the client might be defined as follows:

```
// OMG IDL
interface ClientObject{
     oneway void op1(in string s);
}
```

Client interface explanation

In the preceding example, the `op1()` operation is declared as oneway, because it is important that the server is not blocked when it calls back its clients.

Server interface example

The server implements an IDL interface that allows it to maintain a list of clients that should be notified of changes in its objects' data. A suitable IDL interface for the server might be defined as follows:

```
// OMG IDL
interface Callback{
     oneway void Register(in ClientObject obj);
}
```

Server interface explanation

In the preceding example, the `Register()` operation registers a client with the server. The parameter to `Register()` is of the `ClientObject` type, so that the client can pass a reference to itself to the server. The server can maintain this reference in a list of clients that should be notified of events of interest.

Implementing the Client in C#

Overview

After you have defined the OMG IDL interfaces for the server and client, you can start implementing the client and server. To write a client based on the IDL in "Defining the OMG IDL Interfaces" on page 32, you must implement the `ClientObject` interface defined for the client objects. This subsection describes how to implement the client in C#. The following topics are discussed:

- "Client implementation code" on page 34.
- "Main client code" on page 35.

Note: Because it implements an interface, the client is acting as a server. However, the client does not have to register its implementation object with the bridge, and it is not registered in the Implementation Repository. Therefore, the server cannot bind to the client's implementation object.

Client implementation code

The following is the code in the generated `ClientObjectImpl.cs` file:

```
public class ClientObjectImpl : ClientObject
{
    public System.Boolean called;
    public ClientObjectImpl()
    {
        called = false;
    }
    #region ClientObject Members

    public void op1(string s)
    {
        Console.WriteLine("ClientObjectImpl::op1(): called.");
        Console.WriteLine("    s = " + s);
       Console.WriteLine("ClientObjectImpl::op1(): returning.");
        called = true;
    }

    #endregion

}
```

As shown in the preceding example, the C# class `ClientObjectImpl` inherits from the `ClientObject` interface.

Main client code

The following code extract is from the `client.cs` file:

```
…
       // Create the remote proxy
       // The URL parameter to this call could either be an ior, a
       // corbaloc or a Naming Service ref.
       // The new operator can be used here instead of GetObject,
       // in this case the url can be specified in a config file.
1      CallBack CallBackObj = (CallBack)
           Activator.GetObject(typeof (CallBack), url);
       // Instansiate the ClientObject and try to register
       // it with the server.
       Console.WriteLine("Calling Register...");
2      ClientObjectImpl ClientObj = new ClientObjectImpl();
3      CallBackObj.Register((ClientObject) ClientObj);
       Console.WriteLine("Called Register.");
       while (!ClientObj.called)
       {
           Thread.Sleep(1000);
       }
…
```

The preceding code extract can be explained as follows:

1. It binds to an object, `CallBackObj`, of the `Callback` type in the server.
2. It creates an implementation object, `ClientObj`, of the `ClientObject` type.
3. It calls the `Register()` operation on the `CallbackObj` server object, and passes it a reference to its implementation object, `ClientObj`. This allows the server to subsequently invoke operations on the callback object.

Implementing the Server in C++

Overview

This section describes the steps to implement a server for the purpose of client callbacks, based on the IDL in "Defining the OMG IDL Interfaces" on page 32. The steps are:

Step	Action
1	Implement the `Callback` interface.
2	Invoke the `op1()` operation on the client object.

Note: See the *CORBA Programmer's Guide, C++* for more details of how to implement servers.

Step 1—Implementing the Callback interface

You must provide an implementation class for the `Callback` interface.

The implementation of the `Register` operation receives an object reference from the client. When the client invokes the `Register` operation on the server, an Orbix proxy object for the client's `ClientObject` object is created in the .NET Connector bridge.

The server uses the Orbix proxy object to call back to the client. The implementation of the `Register()` operation should store the reference to the Orbix proxy for this purpose.

Step 2—Invoking the op1() operation on the client

After the Orbix proxy object for the client's `ClientObject` object has been created in the .NET Connector bridge, the server can then invoke the `op1()` operation on this proxy object.

CHAPTER 5

Development Support Tools

This chapter describes how to use the itts2il utility to generate .NET metadata from existing OMG IDL and perform various type store management tasks.

In this chapter

This chapter discusses the following topics:

Generating .NET Metadata	page 38
Managing the Type Store	page 39

Note: The `itts2il` and `ittypeman` command-line utilities described in this chapter are located in *`install-dir`*`\asp\6.1\bin`, where *`install-dir`* represents your Orbix installation directory.

Generating .NET Metadata

Overview

The first step in writing a .NET client that is to communicate with a CORBA server is to obtain .NET metadata, which describes the target CORBA interfaces and types as .NET interfaces and types. You can generate .NET metadata from existing OMG IDL information in the type store. To minimize manual lookups, you should ensure that each IDL file contains a module.

Registering OMG IDL

Before you attempt to create .NET metadata from the OMG IDL for your target CORBA objects, you must ensure that the OMG IDL is registered with the Interface Repository (IFR). This is because `itts2il` reads the OMG IDL information from the IFR. For example, the following command registers the OMG IDL in the `grid.idl` file with the IFR:

```
idl -R grid.idl
```

Generating metadata

The following command creates a .NET metadata assembly within a `Grid.dll` file, based on the OMG IDL `Grid` interface:

```
itts2il Grid
```

Usage String

You can call up the usage string for `itts2il` as follows:

```
itts2il -?
```

The usage string for `itts2il` is:

```
Usage: [options] <type name> [[<type name>] …]
         -f : file name (defaults to <type name #1>.dll)
         -a : assembly name (defaults to <type name #1>)
         -m : module name (defaults to <type name #1>)
         -i : always connect to the IFR
         -e : lookup and cache type entries from the IFR
              (use "*" to look up the entire IFR)
         -c : list the type store contents
         -w : wipe the type store cache clean
         -v : verbose mode
```

Managing the Type Store

Overview

This section first describes the .NET Connector type store in terms of its role and how it works. It then describes how to use `itts2il` to perform various type store management tasks.

In this section

This section discusses the following topics:

The Role of the Type Store	page 40
The Caching Mechanism of the Type Store	page 42
Adding New Information to the Type Store	page 44
Emptying the Type Store Cache	page 46
Dumping the Type Store Contents	page 47

The Role of the Type Store

Overview

This subsection describes the role of the type store. The following topics are discussed:

- "Graphical Overview" on page 40.
- "Role" on page 41.

Graphical Overview

Figure 4 provides a graphical overview of the central role played by the type store in the use of the .NET Connector development utilities.

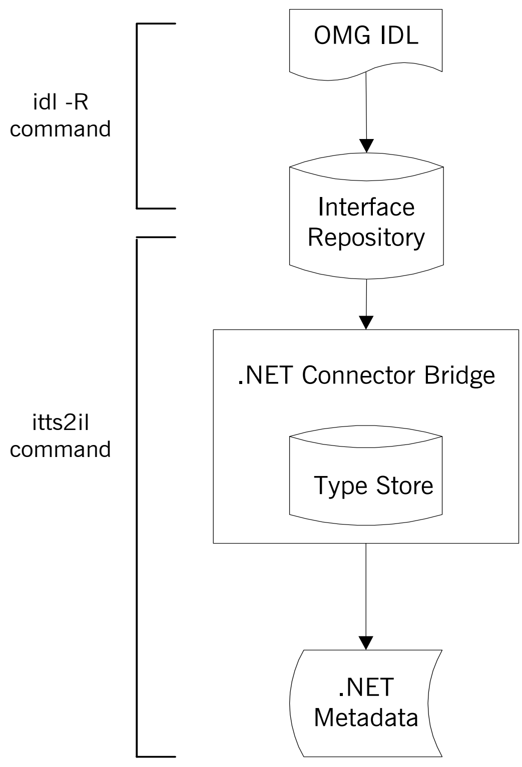

Figure 4: *.NET Connector Type Store and the Development Utilities*

Role

As shown in Figure 4 on page 40, the type store plays a central role in the use of the .NET Connector development utilities. The `itts2il` utility uses the OMG IDL type information in the cache to generate the .NET metadata used by .NET clients to communicate with CORBA objects. The .NET metadata assembly is stored in a DLL file that is also generated using the `itts2il` utility.

The Caching Mechanism of the Type Store

Overview

This subsection describes how type information is stored in the type store. The following topics are discussed:

- "OMG IDL" on page 42.
- "Memory and Disk Cache" on page 42.
- "Type Information Management" on page 42.

OMG IDL

OMG IDL files define the IDL interfaces for CORBA objects. As shown in Figure 4 on page 40, you can register OMG IDL in a CORBA Interface Repository (IFR), where it is stored in binary format. To register an IDL file, enter the following command from the directory where the IDL file is located (where `filename` represents the IDL filename):

```
idl -R filename
```

The .NET Connector uses the OMG IDL type information available in the IFR. The type information can consist of any IDL content (for example, module names, interface names, or data types).

Memory and Disk Cache

A possible performance bottleneck might result at application runtime, if the .NET Connector needs to contact the IFR for each OMG IDL definition. This is because every query might involve multiple remote invocations.

To avoid any bottlenecks, the .NET Connector uses a memory and disk cache of type information. Both the `itts2il` and `ittypeman` utilities are capable of converting OMG IDL type information to an ORB-neutral binary format and caching it in memory. The use of a memory cache means that the .NET Connector has to query the IFR only once for each OMG IDL definition. This memory cache can then be saved to disk for future use.

Type Information Management

At application runtime, when the .NET Connector is marshalling information, and method invocations are being made, the type store cache holds the required type information in memory. The type information is handled on a first-in-first-out basis in the memory cache. This means that the most recently accessed information becomes the most recent in the queue.

On exiting the application process, or when the memory cache size limit has been reached, new entries in the memory cache are written to persistent storage, and are reloaded on the next run of a .NET Connector application.

The memory cache and disk cache are quite separate. Initially, on starting up, the memory cache is primed with the most recently accessed elements of the disk cache. (The number of elements in the memory cache depends on the configuration settings, as described in ".NET Connector Configuration" on page 125.) When lookups are performed, if the required type information is not already in the memory cache, `ittypeman` pulls it out of the disk cache. If the required type information is not in the disk cache, `ittypeman` pulls it out of the IFR. The related type information then becomes the most recent item in the queue in the type store memory cache.

Adding New Information to the Type Store

Overview

This section describes how to use the `itts2il` utility to add new OMG IDL type information to the .NET Connector type store.

Note: All of the commands described here can also be performed using the `ittypeman` utility. It is simply a matter of replacing `itts2il` with `ittypeman` in each case.

Priming the cache

Adding new information to the type store is also known as *priming* the cache. Priming the cache is not mandatory and the only advantage in doing it is that it can help to optimize the first run of a .NET Connector application that is using OMG IDL types that were not already in the type store. As explained in "The Caching Mechanism of the Type Store" on page 42, the type store obtains its information from the IFR on an as-needed basis at application runtime. So, the only reason you might want to prime the cache is if you want to avoid the type store having to contact the IFR on start-up.

Registering OMG IDL

Before you can prime the cache, you must ensure that the relevant OMG IDL is registered with the IFR. This is because the utilities used to prime the cache need to read the OMG IDL information from the IFR. The following command, for example, registers the OMG IDL defined in the `grid.idl` file with the IFR:

```
idl -R grid.idl
```

Priming the Type Store with an Individual Entry

To prime the type store with, for example, the OMG IDL `mygrid` interface, enter:

```
itts2il -e mymodule::myinterface
```

In this case, the `-e` argument instructs `itts2il` to query the IFR for the specified `myinterface` interface, and then add it to the type store. Ensure that you enter the fully scoped name of the OMG IDL type, as shown. This means you must precede the interface name with the module name (that is, `mymodule::` in the previous example).

Priming the Type Store with Multiple Entries

To prime the type store with multiple OMG IDL entries simultaneously, enter for example:

```
itts2il -e module1::interface1 module2::interface2 module3::int3
```

Note: As shown in the preceding example, ensure there is a space between each entry.

Priming the Type Store with the entire IFR

To prime the type store with the entire contents of the IFR, enter:

```
itts2il -e *
```

This is a convenient way of simultaneously priming the cache with the full contents of the IFR.

Emptying the Type Store Cache

Overview

When making changes to IDL during development, it is possible that your .NET metadata and cache of type information in the type store can become inconsistent with the IDL in the IFR. This in turn results in runtime errors. Therefore, if you modify an IDL interface definition, you should subsequently empty the type store cache and then regenerate the .NET metadata. This section describes how to use the `itts2il` utility to empty the contents of the type store.

Note: It is not possible to selectively delete only some type store entries. To delete entries, you must empty the entire cache.

Using the command line

The following command empties the type store (that is, `typeman`) data files:

```
itts2il -w
```

Note: See "Itts2il Argument Details" on page 132 for more details of the -w argument and the type store data files. As an alternative to using the `itts2il -w` command, you can also use the `ittypeman -wm` command to empty the type store data files.

Repriming the cache

The cache can be reprimed with type information in the following ways:

- When you use the `itts2il` command to subsequently regenerate your .NET metadata, the corresponding type information is automatically added to the type store cache.
- If an item of type information cannot be obtained from the type store cache at application runtime on a deployment machine, it is then obtained from the IFR and automatically added to the cache.
- The `itts2il -e *` command can be specified on a deployment machine to add the full contents of the IFR to the cache. This might be done, for example, to avoid a potential performance bottleneck at application runtime that could result if different clients were simultaneously trying to contact the IFR for type information not currently in the local cache.

Dumping the Type Store Contents

Overview

This section describes how to use the `itts2il` utility to list (that is, dump) the contents of the type store cache.

Using the command line

The following command lists the type store contents:

```
itts2il -c
```

Note: As an alternative to using the `itts2il -c` command, you can also use `ittypeman -c` to list type store contents.

Example output

The following is an example of output resulting from the preceding command:

```
OperationSet   TypeTest
CORBA_ENUM               TypeTest::Beer
CORBA_USER_DEF_STRUCT    TypeTest::FixedLength
CORBA_USER_DEF_STRUCT    TypeTest::UserExc
CORBA_USER_DEF_STRUCT    TypeTest::VarLength
CORBA_USER_DEF_UNION     TypeTest::BeerUnion
CORBA_TYPEDEF            TypeTest::LongSeqnce
CORBA_TYPEDEF            _IDL_SEQUENCE_long
```

Deploying a .NET Connector Application

This chapter provides an overview of the deployment model you can adopt when deploying a distributed application with the .NET Connector. It also describes the steps you must follow to deploy a distributed .NET Connector application.

In This Chapter

This chapter discusses the following topics:

Deployment Model	page 50
Deployment Steps	page 52

Deployment Model

Overview

This section provides an overview of the typical deployment model.

Deployment scenario overview

Figure 5 provides a graphical overview of the typical deployment scenario involved in using the .NET Connector to enable .NET clients to communicate with CORBA servers.

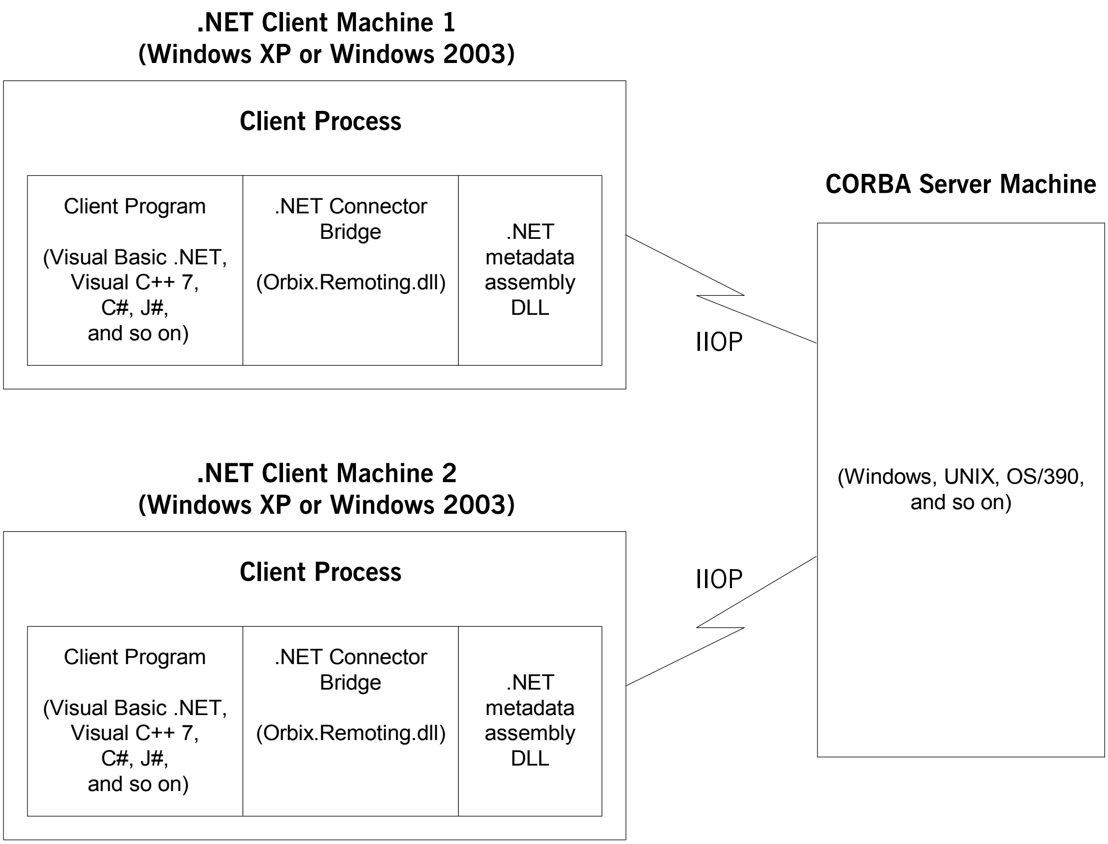

Figure 5: *Overview of Typical Deployment Scenario*

Explanation

The deployment scenario overview in Figure 5 on page 50 can be outlined as follows:

- Each .NET client machine must be running on either Windows XP or Windows 2003.
- The .NET Connector bridge (that is, Orbix.Remoting custom remoting channel) always runs in-process (that is, within the client process).
- The .NET metadata DLL file is also exposed within the client process.
- Each client machine uses IIOP to communicate with the CORBA server.
- The CORBA server process can be running on any platform that is supported by the server-side ORB being used.

Deployment Steps

Overview

This section describes the steps involved in deploying a .NET Connector application.

Required components

Two components are required for successful deployment of a .NET Connector client:

- The .NET client executable.
- The .NET metadata assembly DLL.

These must be copied from the development host to every deployment host.

Steps

The steps to deploy a .NET Connector client application are:

1. Install the Orbix 6.*x* runtime on the deployment host.
2. Use the `itconfigure` utility to create a configuration domain on the deployment host that either contains a local IFR or provides access to a remote centralized IFR.

   > **Note:** An alternative to this step is to copy the typestore cache files from the development machine to the deployment host. If you do this, you do not need to configure an IFR at all for the deployment host.

3. Copy the client executable and the .NET metadata DLL to the deployment host.

Repeat these steps as necessary for each deployment host on your system. See Figure 6 on page 54 for a graphical overview of these steps.

Points to note

Note the following points:

- You can choose to copy the typestore cache files from the development machine to the deployment host. If you do this, it removes the need to configure an IFR at all for the deployment host.
- Using the `itts2il -e *` command on the deployment host primes the local typestore cache with the entire contents of the IFR. If you do this, it removes the need to contact an IFR at runtime.
- A configuration repository-based domain is very useful for the purposes of deploying and managing the deployment of multiple clients.
- If you are deploying onto multiple hosts, it might be convenient to use the `-nogui -load` *`domain_descriptor`* arguments with the `itconfigure` utility. See the Orbix 6.*x Administrator's Guide* for more details.

Graphical overview

Figure 6 provides a graphical overview of the steps involved in deploying a .NET Connector client application.

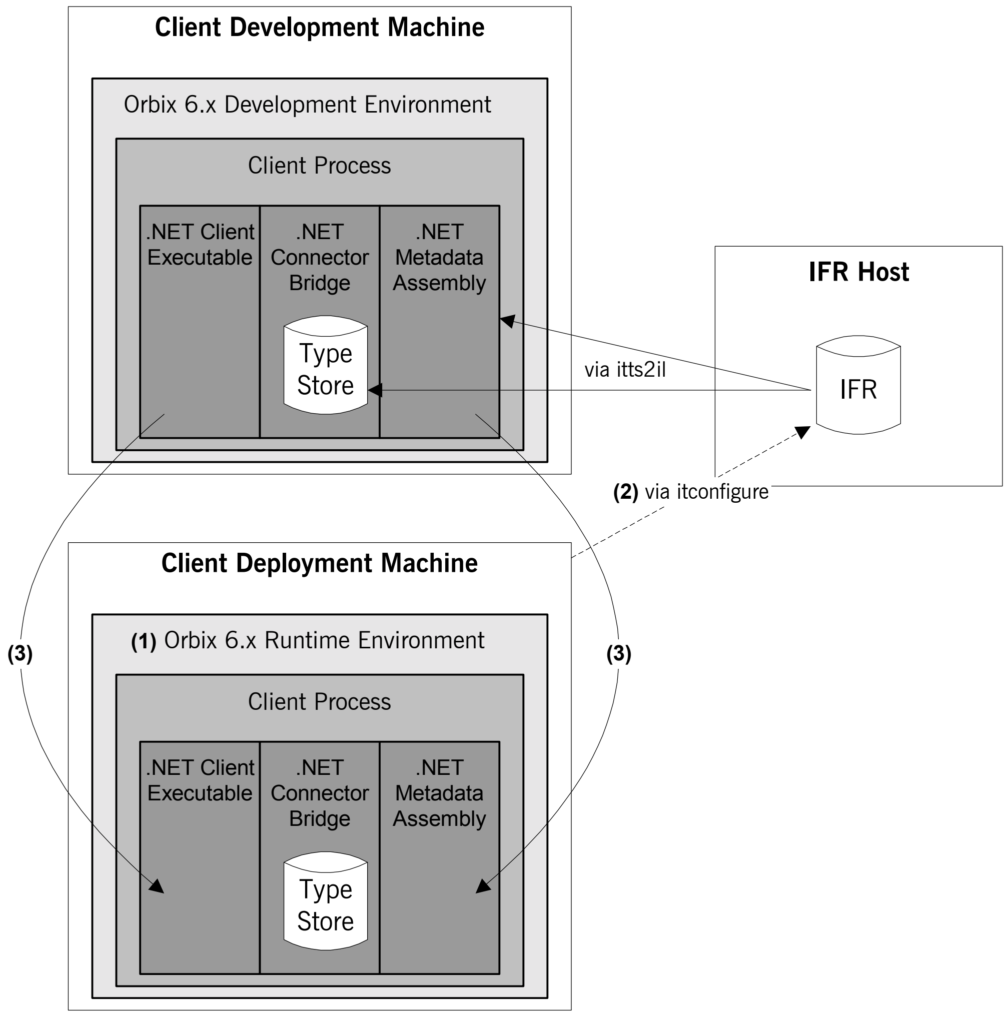

Figure 6: *Overview of Deployment Steps*

Introduction to OMG IDL

An object's interface describes that object to potential clients through its attributes and operations, and their signatures. This chapter describes the semantics and uses of the CORBA Interface Definition Language (OMG IDL), which is used to describe the interfaces to CORBA objects.

In This Chapter

This chapter discusses the following topics:

IDL	page 56
Modules and Name Scoping	page 57
Interfaces	page 58
IDL Data Types	page 78
Defining Data Types	page 93

Note: .NET does not support all the OMG IDL types described in this chapter. See "Mapping CORBA to .NET" on page 99 for details of the OMG IDL types that the .NET Connector supports.

IDL

Overview

An IDL-defined object can be implemented in any language that IDL maps to, including C++, Java, COBOL, and PL/I. By encapsulating object interfaces within a common language, IDL facilitates interaction between objects regardless of their actual implementation. Writing object interfaces in IDL is therefore central to achieving the CORBA goal of interoperability between different languages and platforms.

IDL Standard Mappings

CORBA defines standard mappings from IDL to several programming languages, including C++, Java, COBOL, and PL/I. Each IDL mapping specifies how an IDL interface corresponds to a language-specific implementation. The Orbix 2000 IDL compiler uses these mappings to convert IDL definitions to language-specific definitions that conform to the semantics of that language.

Overall Structure

You create an application's IDL definitions within one or more IDL modules. Each module provides a naming context for the IDL definitions within it. Modules and interfaces form naming scopes, so identifiers defined inside an interface need to be unique only within that interface.

IDL Definition Structure

In the following example, two interfaces, `Bank` and `Account`, are defined within the `BankDemo` module:

```
module BankDemo
{
interface Bank {
         //…
     };

     interface Account {
         //…
     };
};
```

Modules and Name Scoping

Resolving a Name

To resolve a name, the IDL compiler conducts a search among the following scopes, in the order outlined:

1. The current interface.
2. Base interfaces of the current interface (if any).
3. The scopes that enclose the current interface.

Referencing Interfaces

Interfaces can reference each other by name alone within the same module. If an interface is referenced from outside its module, its name must be fully scoped, with the following syntax:

```
module-name::interface-name
```

For example, the fully scoped names of the `Bank` and `Account` interfaces shown in "IDL Definition Structure" on page 56 are, respectively, `BankDemo::Bank` and `BankDemo::Account`.

Nesting Restrictions

A module cannot be nested inside a module of the same name. Likewise, you cannot directly nest an interface inside a module of the same name. To avoid name ambiguity, you can provide an intervening name scope as follows:

```
module A
{
    module B
    {
        interface A {
            //…
        };
    };
};
```

Interfaces

Overview

This section provides details about OMG IDL interfaces.

In This Section

The following topics are discussed in this section:

Introduction to Interfaces	page 59
Interface Contents	page 61
Operations	page 62
Attributes	page 65
Exceptions	page 66
Empty Interfaces	page 67
Inheritance of Interfaces	page 68
Multiple Inheritance	page 69

Introduction to Interfaces

Overview

This subsection provides an introductory overview of OMG IDL interfaces.

What Are Interfaces?

Interfaces are the fundamental abstraction mechanism of CORBA. An interface defines a type of object, including the operations that object supports in a distributed enterprise application.

Objects and Interfaces

Every CORBA object has exactly one interface. However, the same interface can be shared by many CORBA objects in a system. CORBA object references specify CORBA objects (that is, interface instances). Each reference denotes exactly one object, which provides the only means by which that object can be accessed for operation invocations.

Public Members

Because an interface does not expose an object's implementation, all members are public. A client can access variables in an object's implementation only through an interface's operations and attributes.

Operations and Attributes

An IDL interface generally defines an object's behavior through operations and attributes:

- Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.
- An IDL attribute is short-hand for a pair of operations that get and, optionally, set values in an object.

Account Interface IDL Sample

In the following example, the `Account` interface in the `BankDemo` module describes the objects that implement the bank accounts:

```
module BankDemo
{
    typedef float CashAmount; // Type for representing cash
    typedef string AccountId; // Type for representing account
                              // ids
    //…
    interface Account {
        readonly attribute AccountId  account_id;
        readonly attribute CashAmount balance;

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

Code Explanation

This interface has two readonly attributes, `AccountId` and `balance`, which are respectively defined as typedefs of the `string` and `float` types. The interface also defines two operations, `withdraw()` and `deposit()`, which a client can invoke on this object.

Interface Contents

IDL Interface Components

An IDL interface definition typically has the following components.

- Operation definitions.
- Attribute definitions
- Exception definitions.
- Type definitions.
- Constant definitions.

Of these, operations and attributes must be defined within the scope of an interface, all other components can be defined at a higher scope.

Operations

Overview

Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.

Operation Components

IDL operations define the signature of an object's function, which client invocations on that object must use. The signature of an IDL operation is generally composed of three components:

- Return value data type.
- Parameters and their direction.
- Exception clause.

An operation's return value and parameters can use any data types that IDL supports.

Operations IDL Sample

In the following example, the `Account` interface defines two operations, `withdraw()` and `deposit()`, and an `InsufficientFunds` exception:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //...
    interface Account {
        exception InsufficientFunds {};

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

Code Explanation

On each invocation, both operations expect the client to supply an argument for the `amount` parameter, and return `void`. Invocations on the `withdraw()` operation can also raise the `InsufficientFunds` exception, if necessary.

Parameter Direction

Each parameter specifies the direction in which its arguments are passed between client and object. Parameter-passing modes clarify operation definitions and allow the IDL compiler to accurately map operations to a target programming language. The COBOL runtime uses parameter-passing modes to determine in which direction or directions it must marshal a parameter.

Parameter-Passing Mode Qualifiers

There are three parameter-passing mode qualifiers:

`in`	This means that the parameter is initalized only by the client and is passed to the object.
`out`	This means that the parameter is initialized only by the object and returned to the client.
`inout`	This means that the parameter is initialized by the client and passed to the server; the server can modify the value before returning it to the client.

In general, you should avoid using `inout` parameters. Because an `inout` parameter automatically overwrites its initial value with a new value, its usage assumes that the caller has no use for the parameter's original value. Thus, the caller must make a copy of the parameter in order to retain that value. By using the two parameters, `in` and `out`, the caller can decide for itself when to discard the parameter.

One-Way Operations

By default, IDL operations calls are *synchronous*—that is, a client invokes an operation on an object and blocks until the invoked operation returns. If an operation definition begins with the keyword `oneway`, a client that calls the operation remains unblocked while the object processes the call.

The COBOL runtime cannot guarantee the success of a one-way operation call. Because one-way operations do not support return data to the client, the client cannot ascertain the outcome of its invocation. The COBOL runtime indicates failure of a one-way operation only if the call fails before it exits the client's address space; in this case, the COBOL runtime raises a system exception.

A client can also issue non-blocking, or asynchronous, invocations. See the *CORBA Programmer's Guide, C++* for more details.

One-Way Operation Constraints

Three constraints apply to a one-way operation:

- The return value must be set to `void`.
- Directions of all parameters must be set to `in`.
- No `raises` clause is allowed.

One-Way Operation IDL Sample

In the following example, the `Account` interface defines a one-way operation that sends a notice to an `Account` object:

```
module BankDemo {
    //…
    interface Account {
        oneway void notice(in string text);
        //…
    };
};
```

Attributes

Attributes Overview

An interface's attributes correspond to the variables that an object implements. Attributes indicate which variables in an object are accessible to clients.

Qualified and Unqualified Attributes

Unqualified attributes map to a pair of `get` and `set` functions in the implementation language, which allow client applications to read and write attribute values. An attribute that is qualified with the `readonly` keyword maps only to a `get` function.

IDL Readonly Attributes Sample

For example the `Account` interface defines two readonly attributes, `AccountId` and `balance`. These attributes represent information about the account that only the object's implementation can set; clients are limited to readonly access:

```
module BankDemo
{
     typedef float CashAmount; // Type for representing cash
     typedef string AccountId; //Type for representing account
        ids
     //…
     interface Account {
         readonly attribute AccountId  account_id;
         readonly attribute CashAmount balance;

         void
         withdraw(in CashAmount amount)
         raises (InsufficientFunds);

         void
         deposit(in CashAmount amount);
     };
};
```

Code Explanation

The `Account` interface has two readonly attributes, `AccountId` and `balance`, which are respectively defined as typedefs of the `string` and `float` types. The interface also defines two operations, `withdraw()` and `deposit()`, which a client can invoke on this object.

Exceptions

IDL and Exceptions

IDL operations can raise one or more CORBA-defined system exceptions. You can also define your own exceptions and explicitly specify these in an IDL operation. An IDL exception is a data structure that can contain one or more member fields, formatted as follows:

```
exception exception-name {
     [member;]…
};
```

Exceptions that are defined at module scope are accessible to all operations within that module; exceptions that are defined at interface scope are accessible on to operations within that interface.

The raises Clause

After you define an exception, you can specify it through a `raises` clause in any operation that is defined within the same scope. A `raises` clause can contain multiple comma-delimited exceptions:

```
return-val operation-name( [params-list] )
     raises( exception-name[, exception-name] );
```

Example of IDL-Defined Exceptions

The `Account` interface defines the `InsufficientFunds` exception with a single member of the `string` data type. This exception is available to any operation within the interface. The following IDL defines the `withdraw()` operation to raise this exception when the withdrawal fails:

```
module BankDemo
{
     typedef float CashAmount;  // Type for representing cash
     //…
     interface Account {
         exception InsufficientFunds {};

         void
         withdraw(in CashAmount amount)
         raises (InsufficientFunds);
         //…
     };
};
```

Empty Interfaces

Defining Empty Interfaces

IDL allows you to define empty interfaces. This can be useful when you wish to model an abstract base interface that ties together a number of concrete derived interfaces.

IDL Empty Interface Sample

In the following example, the CORBA `PortableServer` module defines the abstract `Servant Manager` interface, which serves to join the interfaces for two servant manager types, `ServantActivator` and `ServantLocator`:

```
module PortableServer
{
    interface ServantManager {};

    interface ServantActivator : ServantManager {
        //…
    };

    interface ServantLocator : ServantManager {
        //…
    };
};
```

Inheritance of Interfaces

Inheritance Overview

An IDL interface can inherit from one or more interfaces. All elements of an inherited, or *base* interface, are available to the *derived* interface. An interface specifies the base interfaces from which it inherits, as follows:

```
interface new-interface : base-interface[, base-interface]…
{…};
```

Inheritance Interface IDL Sample

In the following example, the `CheckingAccount` and `SavingsAccount` interfaces inherit from the `Account` interface, and implicitly include all its elements:

```
module BankDemo{
     typedef float CashAmount;  // Type for representing cash
     interface Account {
         //…
     };

     interface CheckingAccount : Account {
         readonly attribute CashAmount overdraftLimit;
         boolean orderCheckBook ();
     };

     interface SavingsAccount : Account {
         float calculateInterest ();
     };
};
```

Code Sample Explanation

An object that implements the `CheckingAccount` interface can accept invocations on any of its own attributes and operations as well as invocations on any of the elements of the `Account` interface. However, the actual implementation of elements in a `CheckingAccount` object can differ from the implementation of corresponding elements in an `Account` object. IDL inheritance only ensures type-compatibility of operations and attributes between base and derived interfaces.

Multiple Inheritance

Multiple Inheritance IDL Sample

In the following IDL definition, the `BankDemo` module is expanded to include the `PremiumAccount` interface, which inherits from the `CheckingAccount` and `SavingsAccount` interfaces:

```
module BankDemo {
    interface Account {
        //…
    };

    interface CheckingAccount : Account {
        //…
    };

    interface SavingsAccount : Account {
        //…
    };

    interface PremiumAccount :
        CheckingAccount, SavingsAccount {
        //…
    };
};
```

Multiple Inheritance Constraints

Multiple inheritance can lead to name ambiguity among elements in the base interfaces. The following constraints apply:

- Names of operations and attributes must be unique across all base interfaces.
- If the base interfaces define constants, types, or exceptions of the same name, references to those elements must be fully scoped.

Inheritance Hierarchy Diagram

Figure 7 shows the inheritance hierarchy for the `Account` interface, which is defined in “Multiple Inheritance IDL Sample” on page 69.

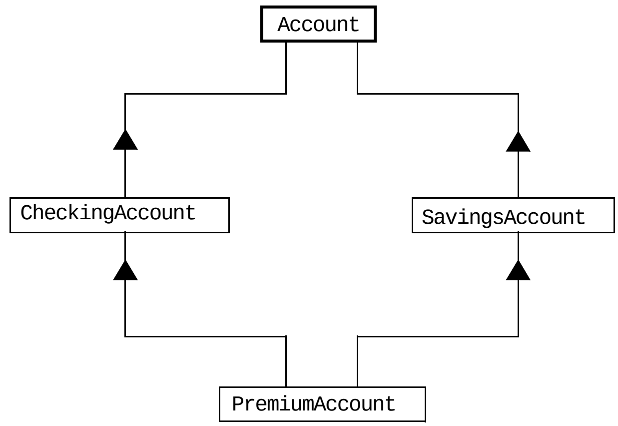

Figure 7: *Inheritance Hierarchy for PremiumAccount Interface*

Inheritance of the Object Interface

User-Defined Interfaces

All user-defined interfaces implicitly inherit the predefined `Object` interface. Thus, all `Object` operations can be invoked on any user-defined interface. You can also use `Object` as an attribute or parameter type, to indicate that any interface type is valid for the attribute or parameter.

Object Locator IDL Sample

For example, the following `getAnyObject()` operation serves as an all-purpose object locator:

```
interface ObjectLocator {
    void getAnyObject (out Object obj);
};
```

Note: It is illegal in IDL syntax to explicitly inherit the `Object` interface.

Inheritance Redefinition

Overview

A derived interface can modify the definitions of constants, types, and exceptions that it inherits from a base interface. All other components that are inherited from a base interface cannot be changed.

Inheritance Redefinition IDL Sample

In the following example, the `CheckingAccount` interface modifies the definition of the `InsufficientFunds` exception, which it inherits from the `Account` interface:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //…
    interface Account {
        exception InsufficientFunds {};
        //…
    };
    interface CheckingAccount : Account {
        exception InsufficientFunds {
            CashAmount overdraftLimit;
        };
    };
    //…
};
```

Note: While a derived interface definition cannot override base operations or attributes, operation overloading is permitted in interface implementations for those languages, such as C++, that support it. However, COBOL does not support operation overloading.

Forward Declaration of IDL Interfaces

Overview

An IDL interface must be declared before another interface can reference it. If two interfaces reference each other, the module must contain a forward declaration for one of them; otherwise, the IDL compiler reports an error. A forward declaration only declares the interface's name; the interface's actual definition is deferred until later in the module.

Forward Declaration IDL Sample

In the following example, the `Bank` interface defines a `create_account()` and `find_account()` operation, both of which return references to `Account` objects. Because the `Bank` interface precedes the definition of the `Account` interface, `Account` is forward-declared:

```
module BankDemo
{
    typedef float CashAmount; // Type for representing cash
    typedef string AccountId; //Type for representing account ids

    // Forward declaration of Account
    interface Account;

    // Bank interface...used to create Accounts
    interface Bank {
        exception AccountAlreadyExists { AccountId account_id; };
        exception AccountNotFound      { AccountId account_id; };

        Account
        find_account(in AccountId account_id)
        raises(AccountNotFound);

        Account
        create_account(
            in AccountId account_id,
            in CashAmount initial_balance
        ) raises (AccountAlreadyExists);
    };

    // Account interface…used to deposit, withdraw, and query
    // available funds.
    interface Account { //…
    };
};
```

Local Interfaces

Overview

An interface declaration that contains the IDL `local` keyword defines a *local interface*. An interface declaration that omits this keyword can be referred to as an *unconstrained interface*, to distinguish it from local interfaces. An object that implements a local interface is a *local object*.

Characteristics

Local interfaces differ from unconstrained interfaces in the following ways:

- A local interface can inherit from any interface, whether local or unconstrained. Unconstrained interfaces cannot inherit from local interfaces.
- Any non-interface type that uses a local interface is regarded as a local type. For example, a struct that contains a local interface member is regarded as a local struct, and is subject to the same localization constraints as a local interface.
- Local types can be declared as parameters, attributes, return types, or exceptions only in a local interface, or as state members of a valuetype.
- Local types cannot be marshalled, and references to local objects cannot be converted to strings through `ORB::object_to_string()`. Any attempts to do so throw a `CORBA::MARSHAL` exception.
- Any operation that expects a reference to a remote object cannot be invoked on a local object. For example, you cannot invoke any DII operations or asynchronous methods on a local object; similarly, you cannot invoke pseudo-object operations such as `is_a()` or `validate_connection()`. Any attempts to do so throw a `CORBA::NO_IMPLEMENT` exception.
- The ORB does not mediate any invocations on a local object. Thus, local interface implementations are responsible for providing the parameter copy semantics that a client expects.
- Instances of local objects that the OMG defines, as supplied by ORB products, are exposed either directly or indirectly through `ORB::resolve_initial_references()`.

Implementation

Local interfaces are implemented by `CORBA::LocalObject` to provide implementations of `Object` pseudo-operations, and other ORB-specific support mechanisms that apply. Because object implementations are language-specific, the `LocalObject` type is only defined by each language mapping.

Local Object Pseudo-Operations

The `LocalObject` type implements the `Object` pseudo-operations shown in Table 2.

Table 2: *CORBA::LocalObject Pseudo-Operations and Return Values*

Operation	Always returns
`is_a()`	An exception of `NO_IMPLEMENT`.
`get_interface()`	An exception of `NO_IMPLEMENT`.
`get_domain_managers()`	An exception of `NO_IMPLEMENT`.
`get_policy()`	An exception of `NO_IMPLEMENT`.
`get_client_policy()`	An exception of `NO_IMPLEMENT`.
`set_policy_overrides()`	An exception of `NO_IMPLEMENT`.
`get_policy_overrides()`	An exception of `NO_IMPLEMENT`.
`validate_connection()`	An exception of `NO_IMPLEMENT`.
`non_existent()`	False.
`hash()`	A hash value that is consistent with the object's lifetime.
`is_equivalent()`	True, if the references refer to the same `LocalObject` implementation.

Valuetypes

Overview

Valuetypes enable programs to pass objects by value across a distributed system. This type is especially useful for encapsulating lightweight data such as linked lists, graphs, and dates.

Characteristics

Valuetypes can be seen as a cross between the following:

- Data types, such as `long` and `string`, which can be passed by value over the wire as arguments to remote invocations.
- Objects, which can only be passed by reference.

When a program supplies an object reference, the object remains in its original location; subsequent invocations on that object from other address spaces move across the network, rather than the object moving to the site of each request.

Valuetype Support

Like an interface, a valuetype supports both operations and inheritance from other valuetypes; it also can have data members. When a valuetype is passed as an argument to a remote operation, the receiving address space creates a copy of it. The copied valuetype exists independently of the original; operations that are invoked on one have no effect on the other.

Valuetype Invocations

Because a valuetype is always passed by value, its operations can only be invoked locally. Unlike invocations on objects, valuetype invocations are never passed over the wire to a remote valuetype.

Valuetype Implementations

Valuetype implementations necessarily vary, depending on the languages used on sending and receiving ends of the transmission, and their respective abilities to marshal and demarshal the valuetype's operations. A receiving process that is written in C++ must provide a class that implements valuetype operations and a factory to create instances of that class. These classes must be either compiled into the application, or made available through a shared library. Conversely, Java applications can marshal enough information on the sender, so the receiver can download the bytecodes for the valuetype operation implementations.

Abstract Interfaces

Overview

An application can use abstract interfaces to determine at runtime whether an object is passed by reference or by value.

IDL Abstract Interface Sample

In the following example, the IDL definitions specify that the `Example::display()` operation accepts any derivation of the abstract interface, `Describable`:

```
abstract interface Describable {
    string get_description();
};

interface Example {
    void display(in Describable someObject);
};
```

Abstract Interface IDL Sample

Based on the preceding IDL, you can define two derivations of the `Describable` abstract interface—the `Currency` valuetype and the `Account` interface:

```
interface Account : Describable {
    // body of Account definition not shown
};

valuetype Currency supports Describable {
    // body of Currency definition not shown
};
```

Note: Because the parameter for `display()` is defined as a `Describable` type, invocations on this operation can supply either `Account` objects or `Currency` valuetypes.

IDL Data Types

In This Section

The following topics are discussed in this section:

Built-in Data Types	page 79
Extended Built-in Data Types	page 82
Complex Data Types	page 85
Enum Data Type	page 86
Struct Data Type	page 87
Union Data Type	page 88
Arrays	page 90
Sequence	page 91
Pseudo Object Types	page 92

Data Type Categories

In addition to IDL module, interface, valuetype, and exception types, IDL data types can be grouped into the following categories:

- Built-in types such as `short`, `long`, and `float`.
- Extended built-in types such as `long long` and `wstring`.
- Complex types such as `enum`, `struct`, and `string`.
- Pseudo objects.

Built-in Data Types

List of Types, Sizes, and Values

Table 3 shows a list of CORBA IDL built-in data types (where the ≤ symbol means 'less than or equal to').

Table 3: *Built-in IDL Data Types, Sizes, and Values*

Data type	Size	Range of values
short	≤ 16 bits	$-2^{15}...2^{15}-1$
unsigned short	≤ 16 bits	$0...2^{16}-1$
long	≤ 32 bits	$-2^{31}...2^{31}-1$
unsigned long	≤ 32 bits	$0...2^{32}-1$
float	≤ 32 bits	IEEE single-precision floating point numbers
double	≤ 64 bits	IEEE double-precision floating point numbers
char	≤ 8 bits	ISO Latin-1
string	Variable length	ISO Latin-1, except NUL
string<bound>	Variable length	ISO Latin-1, except NUL
boolean	Unspecified	TRUE or FALSE
octet	≤ 8 bits	`0x0` to `0xff`
any	Variable length	Universal container type

Floating Point Types

The float and double types follow IEEE specifications for single-precision and double-precision floating point values, and on most platforms map to native IEEE floating point types.

Char Type

The char type can hold any value from the ISO Latin-1 character set. Code positions 0-127 are identical to ASCII. Code positions 128-255 are reserved for special characters in various European languages, such as accented vowels.

String Type

The `string` type can hold any character from the ISO Latin-1 character set, except `NUL`. IDL prohibits embedded `NUL` characters in strings. Unbounded string lengths are generally constrained only by memory limitations. A bounded string, such as `string<10>`, can hold only the number of characters specified by the bounds, excluding the terminating `NUL` character. Thus, a `string<6>` can contain the six-character string, cheese.

Bounded and Unbounded Strings

The declaration statement can optionally specify the string's maximum length, thereby determining whether the string is bounded or unbounded:

```
string[length] name
```

For example, the following code declares the `ShortString` type, which is a bounded string with a maximum length of 10 characters:

```
typedef string<10> ShortString;
attribute ShortString shortName; // max length is 10 chars
```

Octet Type

Octet types are guaranteed not to undergo any conversions in transit. This lets you safely transmit binary data between different address spaces. Avoid using the char type for binary data, because characters might be subject to translation during transmission. For example, if a client that uses ASCII sends a string to a server that uses EBCDIC, the sender and receiver are liable to have different binary values for the string's characters.

Any Type

The any type allows specification of values that express any IDL type, which is determined at runtime, thereby allowing a program to handle values whose types are not known at compile time. An any logically contains a `TypeCode` and a value that is described by the `TypeCode`. A client or server can construct an any to contain an arbitrary type of value and then pass this call in a call to the operation. A process receiving an any must determine what type of value it stores and then extract the value via the typecode. See the *CORBA Programmer's Guide, C++* for more details about the any type.

Extended Built-in Data Types

List of Types, Sizes, and Values

Table 4 shows a list of CORBA IDL extended built-in data types (where the ≤ symbol means 'less than or equal to').

Table 4: *Extended built-in IDL Data Types, Sizes, and Values*

Data Type	Size	Range of Values
long long[a]	≤ 64 bits	$-2^{63}...2^{63}$-1
unsigned long long[a]	≤ 64 bits	0...-2^{64}-1
long double[b]	≤ 79 bits	IEEE double-extended floating point number, with an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. The `long double` type is currently not supported on Windows NT.
wchar	Unspecified	Arbitrary codesets
wstring	Variable length	Arbitrary codesets
fixed[c]	Unspecified	≤ 31significant digits

a. Due to compiler restrictions, the COBOL range of values for the `long long` and `unsigned long long` types is the same range as for a `long` type (that is, 0...2^{31}-1).

b. Due to compiler restrictions, the COBOL range of values for the `long double` type is the same range as for a double type (that is, ≤ 64 bits).

c. Due to compiler restrictions, the COBOL range of values for the fixed type is ≤ 18 significant digits.

Long Long Type

The 64-bit integer types, `long long` and `unsigned long long`, support numbers that are too large for 32-bit integers. Platform support varies. If you compile IDL that contains one of these types on a platform that does not support it, the compiler issues an error.

Long Double Type

Like 64-bit integer types, platform support varies for the `long double` type, so its use can yield IDL compiler errors.

Wchar Type

The `wchar` type encodes wide characters from any character set. The size of a `wchar` is platform-dependent. Because Orbix 2000 currently does not support character set negotiation, use this type only for applications that are distributed across the same platform.

Wstring Type

The `wstring` type is the wide-character equivalent of the `string` type. Like `string` types, `wstring` types can be unbounded or bounded. Wide strings can contain any character except `NUL`.

Fixed Type

IDL specifies that the `fixed` type provides fixed-point arithmetic values with up to 31 significant digits. However, due to restrictions in the COBOL compiler for OS/390, only up to 18 significant digits are supported.

You specify a `fixed` type with the following format:

```
typedef fixed<digit-size,scale> name
```

The format for the fixed type can be explained as follows:

- The *`digit-size`* represents the number's length in digits. The maximum value for *`digit-size`* is 31 and it must be greater than *`scale`*. A `fixed` type can hold any value up to the maximum value of a `double` type.
- If *`scale`* is a positive integer, it specifies where to place the decimal point relative to the rightmost digit. For example, the following code declares a fixed type, `CashAmount`, to have a digit size of 10 and a scale of 2:

  ```
  typedef fixed<10,2> CashAmount;
  ```

  Given this typedef, any variable of the `CashAmount` type can contain values of up to (+/-)99999999.99.

- If *scale* is a negative integer, the decimal point moves to the right by the number of digits specified for *scale*, thereby adding trailing zeros to the fixed data type's value. For example, the following code declares a fixed type, bigNum, to have a digit size of 3 and a scale of -4:

  ```
  typedef fixed <3,-4> bigNum;
  bigNum myBigNum;
  ```

  If myBigNum has a value of 123, its numeric value resolves to 1230000. Definitions of this sort allow you to efficiently store numbers with trailing zeros.

Constant Fixed Types

Constant fixed types can also be declared in IDL, where *digit-size* and *scale* are automatically calculated from the constant value. For example:

```
module Circle {
    const fixed pi = 3.142857;
};
```

This yields a fixed type with a digit size of 7, and a scale of 6.

Fixed Type and Decimal Fractions

Unlike IEEE floating-point values, the fixed type is not subject to representational errors. IEEE floating point values are liable to inaccurately represent decimal fractions unless the value is a fractional power of 2. For example, the decimal value, 0.1, cannot be represented exactly in IEEE format. Over a series of computations with floating-point values, the cumulative effect of this imprecision can eventually yield inaccurate results.

The fixed type is especially useful in calculations that cannot tolerate any imprecision, such as computations of monetary values.

Complex Data Types

IDL Complex Data Types

IDL provide the following complex data types:

- Enums.
- Structs.
- Multi-dimensional fixed-sized arrays.
- Sequences.

Enum Data Type

Overview

An enum (enumerated) type lets you assign identifiers to the members of a set of values.

Enum IDL Sample

For example, you can modify the `BankDemo` IDL with the `balanceCurrency` enum type:

```
module BankDemo {
    enum Currency {pound, dollar, yen, franc};

    interface Account {
        readonly attribute CashAmount balance;
        readonly attribute Currency balanceCurrency;
        //…
    };
};
```

In the preceding example, the `balanceCurrency` attribute in the `Account` interface can take any one of the values `pound`, `dollar`, `yen`, or `franc`.

Ordinal Values of Enum Type

The ordinal values of an enum type vary according to the language implementation. The CORBA specification only guarantees that the ordinal values of enumerated types monotonically increase from left to right. Thus, in the previous example, `dollar` is greater than `pound`, `yen` is greater than `dollar`, and so on. All enumerators are mapped to a 32-bit type.

Struct Data Type

Overview

A struct type lets you package a set of named members of various types.

Struct IDL Sample

In the following example, the `CustomerDetails` struct has several members. The `getCustomerDetails()` operation returns a struct of the `CustomerDetails` type, which contains customer data:

```
module BankDemo{
     struct CustomerDetails {
         string custID;
         string lname;
         string fname;
         short age;
         //…
     };

     interface Bank {
         CustomerDetails getCustomerDetails(in string custID);
         //…
     };
};
```

Note: A struct type must include at least one member. Because a struct provides a naming scope, member names must be unique only within the enclosing structure.

Union Data Type

Overview

A union type lets you define a structure that can contain only one of several alternative members at any given time. A union type saves space in memory, because the amount of storage required for a union is the amount necessary to store its largest member.

Union Declaration Syntax

You declare a union type with the following syntax:

```
union name switch (discriminator) {
     case label1 : element-spec;
     case label2 : element-spec;
     […]
     case labeln : element-spec;
     [default : element-spec;]
};
```

Discriminated Unions

All IDL unions are *discriminated*. A discriminated union associates a constant expression (`label1…labeln`) with each member. The discriminator's value determines which of the members is active and stores the union's value.

IDL Union Date Sample

The following IDL defines a `Date` union type, which is discriminated by an enum value:

```
enum dateStorage
{ numeric, strMMDDYY, strDDMMYY };

struct DateStructure {
     short Day;
     short Month;
     short Year;
};
union Date switch (dateStorage) {
     case numeric: long digitalFormat;
     case strMMDDYY:
     case strDDMMYY: string stringFormat;
     default: DateStructure structFormat;
};
```

Sample Explanation

Given the preceding IDL:

- If the discriminator value for `Date` is numeric, the `digitalFormat` member is active.
- If the discriminator's value is `strMMDDYY` or `strDDMMYY`, the `stringFormat` member is active.
- If neither of the preceding two conditions apply, the default `structFormat` member is active.

Rules for Union Types

The following rules apply to union types:

- A union's discriminator can be `integer`, `char`, `boolean`, `enum`, or an alias of one of these types; all `case` label expressions must be compatible with the relevant type.
- Because a union provides a naming scope, member names must be unique only within the enclosing union.
- Each union contains a pair of values: the discriminator value and the active member.
- IDL unions allow multiple case labels for a single member. In the previous example, the `stringFormat` member is active when the discriminator is either `strMMDDYY` or `strDDMMYY`.
- IDL unions can optionally contain a `default` case label. The corresponding member is active if the discriminator value does not correspond to any other label.

Arrays

Overview

IDL supports multi-dimensional fixed-size arrays of any IDL data type, with the following syntax (where *dimension-spec* must be a non-zero positive constant integer expression):

```
[typedef] element-type array-name [dimension-spec]…
```

IDL does not allow open arrays. However, you can achieve equivalent functionality with sequence types.

Array IDL Sample

For example, the following defines a two-dimensional array of bank accounts within a portfolio:

```
typedef Account portfolio[MAX_ACCT_TYPES][MAX_ACCTS]
```

Note: For an array to be used as a parameter, an attribute, or a return value, the array must be named by a typedef declaration. You can omit a typedef declaration only for an array that is declared within a structure definition.

Array Indexes

Because of differences between implementation languages, IDL does not specify the origin at which arrays are indexed. For example, C and C++ array indexes always start at `0`, but COBOL, PL/I, and Pascal always start at `1`. Consequently, clients and servers cannot exchange array indexes unless they both agree on the origin of array indexes and make adjustments, as appropriate, for their respective implementation languages. Usually, it is easier to exchange the array element itself, instead of its index.

Sequence

Overview

IDL supports sequences of any IDL data type with the following syntax:

`[typedef] sequence <` *`element-type[, max-elements]`* `>` *`sequence-name`*

An IDL sequence is similar to a one-dimensional array of elements; however, its length varies according to its actual number of elements, so it uses memory more efficiently.

For a sequence to be used as a parameter, an attribute, or a return value, the sequence must be named by a typedef declaration. You can omit a typedef declaration only for a sequence that is declared within a structure definition.

A sequence's element type can be of any type, including another sequence type. This feature is often used to model trees.

Bounded and Unbounded Sequences

The maximum length of a sequence can be fixed (bounded) or unfixed (unbounded):

- Unbounded sequences can hold any number of elements, up to the memory limits of your platform.
- Bounded sequences can hold any number of elements, up to the limit specified by the bound.

Bounded and Unbounded IDL Definitions

The following code shows how to declare bounded and unbounded sequences as members of an IDL struct:

```
struct LimitedAccounts {
     string bankSortCode<10>;
     sequence<Account, 50> accounts; // max sequence length is 50
};

struct UnlimitedAccounts {
     string bankSortCode<10>;
     sequence<Account> accounts; // no max sequence length
};
```

Pseudo Object Types

Overview

CORBA defines a set of pseudo-object types that ORB implementations use when mapping IDL to a programming language. These object types have interfaces defined in IDL; however, these object types do not have to follow the normal IDL mapping rules for interfaces and they are not generally available in your IDL specifications.

Defining

You can use only the following pseudo-object types as attribute or operation parameter types in an IDL specification:

```
CORBA::NamedValue
CORBA::TypeCode
```

To use these types in an IDL specification, include the `orb.idl` file in the IDL file as follows:

```
#include <orb.idl>
//…
```

This statement instructs the IDL compiler to allow the `NamedValue` and `TypeCode` types.

Defining Data Types

Overview

With `typedef`, you can define more meaningful or simpler names for existing data types, regardless of whether those types are IDL-defined or user-defined.

The following code defines the `typedef` identifier, `StandardAccount`, so that it can act as an alias for the `Account` type in later IDL definitions:

```
module BankDemo {
    interface Account {
        //…
    };

    typedef Account StandardAccount;
};
```

In This Section

This section contains the following subsections:

Constants	page 94
Constant Expressions	page 97

Constants

Overview

IDL lets you define constants of all built-in types except the any type. To define a constant's value, you can use either another constant (or constant expression) or a literal. You can use a constant wherever a literal is permitted.

Integer Constants

IDL accepts integer literals in decimal, octal, or hexadecimal:

```
const short     I1 = -99;
const long      I2 = 0123;  // Octal 123, decimal 83
const long long I3 = 0x123; // Hexadecimal 123, decimal 291
const long long I4 = +0xaB; // Hexadecimal ab, decimal 171
```

Both unary plus and unary minus are legal.

Floating-Point Constants

Floating-point literals use the same syntax as C++:

```
const float       f1 = 3.1e-9; // Integer part, fraction part,
                               // exponent
const double      f2 = -3.14;  // Integer part and fraction part
const long double f3 = .1      // Fraction part only
const double      f4 = 1.      // Integer part only
const double      f5 = .1E12   // Fraction part and exponent
const double      f6 = 2E12    // Integer part and exponent
```

Character and String Constants

Character constants use the same escape sequences as C++:

```
const char C1 = 'c';        // the character c
const char C2 = '\007';     // ASCII BEL, octal escape
const char C3 = '\x41';     // ASCII A, hex escape
const char C4 = '\n';       // newline
const char C5 = '\t';       // tab
const char C6 = '\v';       // vertical tab
const char C7 = '\b';       // backspace
const char C8 = '\r';       // carriage return
const char C9 = '\f';       // form feed
const char C10 = '\a';      // alert
const char C11 = '\\';      // backslash
const char C12 = '\?';      // question mark
const char C13 = '\'';      // single quote
// String constants support the same escape sequences as C++
const string S1 = "Quote: \"";      // string with double quote
const string S2 = "hello world";    // simple string
const string S3 = "hello" " world"; // concatenate
const string S4 = "\xA" "B";        // two characters
                                    // ('\xA' and 'B'),
                               // not the single character '\xAB'
```

Wide Character and String Constants

Wide character and string constants use C++ syntax. Use universal character codes to represent arbitrary characters. For example:

```
const wchar     C = L'X';
const wstring   GREETING = L"Hello";
const wchar     OMEGA = L'\u03a9';
const wstring   OMEGA_STR = L"Omega: \u3A9";
```

IDL files always use the ISO Latin-1 code set; they cannot use Unicode or other extended character sets.

Boolean Constants

Boolean constants use the FALSE and TRUE keywords. Their use is unnecessary, inasmuch as they create unnecessary aliases:

```
// There is no need to define boolean constants:
const CONTRADICTION = FALSE;    // Pointless and confusing
const TAUTOLOGY = TRUE;         // Pointless and confusing
```

Octet Constants

Octet constants are positive integers in the range 0-255.

```
const octet O1 = 23;
const octet O2 = 0xf0;
```

Octet constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

Fixed-Point Constants

For fixed-point constants, you do not explicitly specify the digits and scale. Instead, they are inferred from the initializer. The initializer must end in `d` or `D`. For example:

```
// Fixed point constants take digits and scale from the
// initializer:
const fixed val1 = 3D;          // fixed<1,0>
const fixed val2 = 03.14d;      // fixed<3,2>
const fixed val3 = -03000.00D;  // fixed<4,0>
const fixed val4 = 0.03D;       // fixed<3,2>
```

The type of a fixed-point constant is determined after removing leading and trailing zeros. The remaining digits are counted to determine the digits and scale. The decimal point is optional.

Currently, there is no way to control the scale of a constant if it ends in trailing zeros.

Enumeration Constants

Enumeration constants must be initialized with the scoped or unscoped name of an enumerator that is a member of the type of the enumeration. For example:

```
enum Size { small, medium, large }

const Size DFL_SIZE = medium;
const Size MAX_SIZE = ::large;
```

Enumeration constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

Constant Expressions

Overview

IDL provides a number of arithmetic and bitwise operators. The arithmetic operators have the usual meaning and apply to integral, floating-point, and fixed-point types (except for `%`, which requires integral operands). However, these operators do not support mixed-mode arithmetic: you cannot, for example, add an integral value to a floating-point value.

Arithmetic Operators

The following code contains several examples of arithmetic operators:

```
// You can use arithmetic expressions to define constants.
const long MIN = -10;
const long MAX = 30;
const long DFLT = (MIN + MAX) / 2;

// Can't use 2 here
const double TWICE_PI = 3.1415926 * 2.0;

// 5% discount
const fixed DISCOUNT = 0.05D;
const fixed PRICE = 99.99D;

// Can't use 1 here
const fixed NET_PRICE = PRICE * (1.0D - DISCOUNT);
```

Evaluating Expressions for Arithmetic Operators

Expressions are evaluated using the type promotion rules of C++. The result is coerced back into the target type. The behavior for overflow is undefined, so do not rely on it. Fixed-point expressions are evaluated internally with 31 bits of precision, and results are truncated to 15 digits.

Bitwise Operators

Bitwise operators only apply to integral types. The right-hand operand must be in the range 0-63. The right-shift operator, >>, is guaranteed to insert zeros on the left, regardless of whether the left-hand operand is signed or unsigned.

```
// You can use bitwise operators to define constants.
const long ALL_ONES = -1;               // 0xffffffff
const long LHW_MASK = ALL_ONES << 16;   // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16;   // 0x0000ffff
```

IDL guarantees two's complement binary representation of values.

Precedence

The precedence for operators follows the rules for C++. You can override the default precedence by adding parentheses.

Mapping CORBA to .NET

CORBA types are defined in OMG IDL, and .NET types are defined in Microsoft Intermediate Language (MSIL). To allow interworking between .NET clients and CORBA servers, .NET clients must be presented with metadata that describes the interfaces exposed by CORBA objects. Therefore, it must be possible to translate CORBA types to .NET types. When using .NET Remoting, the .NET types must use the .NET Common Type System (CTS). This chapter outlines the CORBA-to-.NET CTS mapping rules.

In this chapter

This chapter discusses the following topics:

Mapping for Basic Types	page 101
Mapping for Extended Types	page 102
Mapping for Interfaces	page 103
Mapping for Interface Inheritance	page 105
Mapping for Complex Types	page 106
Mapping for Object References	page 120
Mapping for Modules	page 121

Mapping for Constants	page 122

Note: For the purposes of illustration, the .NET mapping is represented in this chapter in C# rather than MSIL. The mappings shown in this chapter are automatically performed by the .NET Connector.

Mapping for Basic Types

Overview

OMG IDL basic types translate to compatible types in .NET.

Mapping Rules

Table 5 shows the mapping rules for each basic type.

Table 5: *CORBA-to-.NET Mapping Rules for Basic Types*

OMG IDL Type	Description	.NET CTS Type	Description

`boolean`	Valid values are
	`0=FALSE`
	`1=TRUE`
		`System.Boolean`	Valid values are:
			`0=FALSE`
			`1=TRUE`

`char`	8-bit quantity	`System.Byte`	8-bit unsigned integer
`double`	IEEE 64-bit float	`System.Double`	IEEE 64-bit float
float	IEEE 32-bit float	`System.Single`	Single-precision floating point number
long	32-bit integer	`System.Int32`	32-bit signed integer
octet	8-bit quantity	`System.Byte`	8-bit unsigned integer
short	16-bit integer	`System.Int16`	16-bit signed integer
unsigned long	32-bit integer	`System.UInt32`	32-bit unsigned integer
unsigned short	16-bit integer	`System.UInt16`	16-bit unsigned integer
string	Series of characters	`System.String`	Series of unicode characters

Mapping for Extended Types

Overview

OMG IDL extended types translate to compatible types in .NET.

Mapping Rules

Table 6 shows the mapping rules for each extended type.

Table 6: *CORBA-to-.NET Mapping Rules for Extended Types*

OMG IDL Type	Description	.NET CTS Type	Description
`long long`	64-bit integer	`System.Int64`	64-bit signed integer
`unsigned long long`	64-bit integer	`System.UInt64`	64-bit unsigned integer
wchar	16-bit quantity	`System.Char`	16-bit character
wstring	Series of Unicode characters	`System.String`	Series of Unicode characters

Note: There is currently no supported .NET mapping for valutypes and `long double` and `fixed` CORBA types.

Mapping for Interfaces

Overview

This section describes how OMG IDL interfaces map to .NET.

Mapping rules

The rules for mapping OMG IDL interfaces to .NET are:

- An OMG IDL interface maps to a .NET interface that contains the appropriate .NET signatures.
- For each operation declared in an OMG IDL interface there must be a corresponding method defined in the .NET language of choice, with conforming return type and parameter declarations.
- For each attribute declared in an OMG IDL interface there must be a corresponding property defined in the .NET interface. (No set definitions are provided for read-only attributes.)

Example

The example can be broken down as follows:

1. Consider the following OMG IDL interface, `Grid`:

```
// OMG IDL
interface Grid
{
    readonly attribute short height;  // height of the grid
    readonly attribute short width;   // width of the grid

    // set the element [n,m] of the grid, to value:
    void set(in short n, in short m, in long value);

    // return element [n,m] of the grid:
    long get(in short n, in short m);
};
```

2. The preceding OMG IDL maps, for example, to the following C# interface defined using the Common Type System:

```
// C#
interface Grid
{
    Int16 height  // height of the grid
    {
        get;
    }
    Int16 width  //  width of the grid
    {
        get;
    }

    // set the element [n,m] of the grid, to value:
    void set(Int16 n, Int16 m, Int32 value);

    // return element [n,m] of the grid:
    Int32 get(Int16 n, Int16 m);
};
```

Mapping for Interface Inheritance

Overview

This section describes the CORBA-to-.NET mapping rules for interface inheritance.

Mapping rule

A hierarchy of inherited interfaces defined in OMG IDL maps to an identical hierarchy of .NET interfaces.

Mapping for Complex Types

Overview

This section describes the rules for mapping various OMG IDL complex types to .NET.

In this section

This section discusses the following topics:

Mapping for Structs	page 107
Mapping for Enums	page 108
Mapping for Unions	page 109
Mapping for Arrays	page 111
Mapping for Sequences	page 112
Mapping for System Exceptions	page 113
Mapping for User Exceptions	page 114
Mapping for the Any Type	page 115

Mapping for Structs

Overview

This subsection describes the CORBA-to-.NET mapping rules for structs.

Mapping rules

An OMG IDL struct maps to a .NET struct that contains data elements corresponding to the data elements of the OMG IDL struct. When a struct is being marshalled as an `in` parameter (that is, from a .NET client to a CORBA server), the marshaller uses dynamic `any` types to create the OMG IDL struct. When a struct is being marshalled as an `out` parameter (that is, from a CORBA server to a .NET client), .NET reflection is used to construct the .NET struct, as required.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
struct AccountDetails
{
    long number;
    float balance;
};
```

2. The preceding OMG IDL struct maps, for example, to the following C# struct:

```
// C#
public struct AccountDetails
{
    public System.Int32 number;
    public System.Single balance;
};
```

Mapping for Enums

Overview

This subsection describes the CORBA-to-.NET mapping rules for enums.

Mapping rules

An OMG IDL enum maps to a .NET `System.Enum` type. By default, the underlying type for a .NET `System.Enum` is `System.Int32`, but it can be configured to an alternative type.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
interface Typetest
{
    enum e_color [red, green, blue}
}
```

2. The preceding OMG IDL enum maps, for example, to the following C# System.Enum:

```
// C#
namespace IT_Enums
{
    namespace Typetest
    {
        public enum e_color {red, green, blue};
    }
}
```

Note: All enums must be defined within an `IT_Enums` namespace. This is due to a problem with .NET reflection in the current version of the .NET framework, whereby the `TypeBuilder.DefineNestedEnum()` method is not available.

Mapping for Unions

Overview

This subsection describes the CORBA-to-.NET mapping rules for unions.

Mapping rules

.NET does not have anything that equates to an OMG IDL union. For this reason, an OMG IDL union is mapped to a .NET class that provides similar functionality to the OMG IDL union.

When a union is being marshalled as an `in` parameter (that is, from a .NET client to a CORBA server), the marshaller uses dynamic any types to create the CORBA any types required to construct the CORBA request. When a union is being marshalled as an `out` parameter (that is, from a CORBA server to a .NET client), .NET reflection is used to construct the appropriate return parameters, as required.

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
union U switch(long)
{
    case 1: long l;
    case 2: float f;
};
```

2. The preceding OMG IDL maps to the following C# class that implements the union:

```
// C#
public class U
{
    private System.Int32 m_d;
    private System.Int32 l;
    private System.Single f;
    public Int16 _d
    {
        get{ return m_d;}
    }
    public Int16 l
    {
        get{
            if (m_d == 1) return l;
            else throw new Exception("Illegal access of
                union member U::l attempted");
        }
        set{ l = value; m_d = 1};
    }
    public Int16 f
    {
        get{
            if (m_d == 2) return f;
            else throw new Exception("Illegal access of
                union member U::f attempted");
        }
        set{ f = value; m_d = 2};
    }
}
```

Mapping for Arrays

Overview

This subsection describes the CORBA-to-.NET mapping rules for arrays.

Mapping rules

An OMG IDL array maps to a .NET `System.Array` of the type in question.

Mapping for Sequences

Overview

This subsection describes the CORBA-to-.NET mapping rules for sequences.

Mapping rules

An OMG IDL sequence maps to a .NET `System.Array` of the type in question.

Mapping for System Exceptions

Overview

This subsection describes the CORBA-to-.NET mapping rules for system exceptions.

Mapping rules

An OMG IDL system exception currently maps to a .NET exception that contains a stringified description of the exception.

Mapping for User Exceptions

Overview

This subsection describes the CORBA-to-.NET mapping rules for user exceptions.

Mapping rules

An OMG IDL user exception inherits from the .NET `System.Exception` class, and any user-defined fields are then added. When a user exception is thrown and is being marshalled as an `out` parameter (that is, from a CORBA server to a .NET client), .NET reflection is used to construct the .NET exception, as required.

Mapping for the Any Type

Overview

This section describes the CORBA-to-.NET mapping rules for the any type.

Standard mapping rule

For most types, the standard rule for passing a .NET value as an any type simply involves using the any type as a standard `System.Object` parameter to a .NET Remoting call. In this case, the .NET Connector uses the most convenient type mapping by default. For example, consider the following C# client demonstration:

```
//C#
//
//CORBA Any type
//

Int32 long_any_in_value =           18;
Int32 long_any_inout_in_val =       207;
Int32 long_any_inout_out_val =      1000346;
Int32 long_any_out_val =            1009044;
Int32 long_any_return_val =         1000019042;

TypeTestObj.any_in(long_any_in_value);

System.Object any_inout = long_any_inout_in_val;
TypeTestObj.any_inout(ref any_inout);
Debug.Assert((Int32)any_inout == long_any_inout_out_val,
   "any_inout");

System.Object any_out =(Int32) 0x00000000;
TypeTestObj.any_out(out any_out);
Debug.Assert((Int32)any_out == long_any_out_val, "any_out");

Int32 any_return = (Int32)TypeTestObj.any_return();
Debug.Assert(any_return == long_any_return_val, "any_return");
```

Exceptions to standard rule

For certain types, the mapping between the .NET type system and CORBA is not straightforward. These types include:

- char
- octet
- wstring

- sequence

To pass any of these types as an any in a .NET Remoting call, the type must be passed in an IT_Any object. The rest of this sub-section provides an overview of the IT_Any interface and illustrates the mapping rule for passing each of the non-standard types as an any.

IT_Any interface

The following is an overview of the IT_Any interface:

```
namespace IONA
{
    namespace Remoting
    {
        _gc public class IT_Any
        {
            public:

                IT_Any();
                ~IT_Any();

                // Convenience Constructor
                IT_Any(
                    System::String* type_name,
                    System::Object* value
                );

                void insert_char(System::Byte value);
                System::Byte get_char();

                void insert_octet(System::Byte value);
                System::Byte get_octet();

                void insert_wstring(System::String* value);
                System::String* get_wstring();

                void insert_sequence(System::String*
                    sequence_name, System::Object* value);
                System::Object* get_sequence();

                // Values can be "CORBA::Octet", "CORBA::Char",
               // "CORBA::WString", <Name of User Defined STRUCT>
                System::String* get_typename();
                System::Object* get_value();
        };
    }
}
```

Mapping char types

The CORBA char type is an 8–bit value, but the .NET char type is a 16–bit value. Therefore, to pass an 8–bit char type as an any in a .NET Remoting call, the char type must be passed as the .NET 8–bit `Byte` type inside an `IT_Any` object. For example:

1. Consider the following OMG IDL:

```
// OMG IDL
void char_in(in any val);
```

2. Based on the preceding OMG IDL, the following C# client code passes the char type inside an `IT_Any` object:

```
// C#
IT_Any any1 = new IT_Any();
any1.insert_char((System.Byte) ' a' );
TypeTestObj.char_in(any1);
```

Mapping octet types

The CORBA octet type is an 8–bit value, so potential ambiguity exists between it and the CORBA char type. Therefore, to pass an octet type as an any in a .NET Remoting call, the octet type must be passed as the .NET 8–bit `Byte` type inside an `IT_Any` object. For example:

1. Consider the following OMG IDL:

```
// OMG IDL
void octet_inout(inout any val);
```

2. Based on the preceding OMG IDL, the following C# client code passes the octet type inside an IT_Any object:

```
// C#

IT_Any any1 = new IT_Any();

//Insert octet to pass over to server
any1.insert_octet((System.Byte) 0x33);
Object octet_inout = (Object) any1;

TypeTestObj.octet_inout(ref octet_inout);

//Extract octet passed back from server
any1 = (IT_Any)octet_inout;
System.Byte = any1.get_octet();
```

Mapping wstring types

Because most deployed CORBA servers use the CORBA string type in preference to the CORBA wstring type, the .NET Connector uses the CORBA string type by default for its string mappings. To pass a wstring type as an any in a .NET Remoting call, the IT_Any interface must be used. For example:

1. Consider the following OMG IDL:

```
// OMG IDL
void wstring_out(out any val);
```

2. Based on the preceding OMG IDL, the following C# client code uses IT_Any to pass the wstring type.

```
// C#

IT_Any any1 = new IT_Any();

Object wstring_out = (Object)any1;

TypeTestObj.wstring_out(out wstring_out);

any1 = (IT_Any)wstring_out;
Console.WriteLine(any1.get_wstring());
```

Mapping sequence types

It is not possible to distinguish between CORBA sequences based on their structure alone. This is because two sequences might have the same structure and different typenames. To ensure that the .NET Connector passes a sequence as the correct type, the .NET Connector needs to know the sequence typename. To pass a sequence as an any in a .NET Remoting call, the `IT_Any` interface must be used. For example:

1. Consider the following OMG IDL:

```
// OMG IDL
typedef sequence<long> LongSeqnce;
void longseq_in(in any val);
any longseq_return();
```

2. Based on the preceding OMG IDL, the following C# client code uses `IT_Any` to pass the sequence:

```
// C#

IT_Any any1 = new IT_Any();

// In Any CORBA Sequence

Int32[] longseq_in_val = {64839149, 438521937, 1821949};
any1.insert_sequence("LongSeqnce", longseq_in_val);
TypeTestObj.longseq_in(any1);

// Return Any CORBA Sequence

Object sequence_return = (Object)any1;
sequence_return = TypeTestObj.longseq_return();
any1 = (IT_Any)sequence_return;

Console.WriteLine("Sequence name:" + any1.get_typename());
Int32[] longseq_return = (Int32[])any1.get_sequence();
```

Mapping for Object References

Overview

This section describes the CORBA-to-.NET mapping rules for object references.

Mapping rules

The .NET Connector bridge maintains a table that contains all of the CORBA object references that exist within the application. If a CORBA object reference is passed as a parameter of a CORBA operation, it is the proxy for this object that is actually passed. The bridge then finds the corresponding CORBA object reference for this proxy and passes it to the CORBA request. If a CORBA object reference is returned from the server, a proxy is generated for it, if necessary.

Mapping for Modules

Overview

This section describes the CORBA-to-.NET mapping rules for modules.

Mapping rules

An OMG IDL module maps to a .NET namespace that reflects the OMG IDL module name.

Mapping for Constants

Overview

This section describes the CORBA-to-.NET mapping rules for constant types.

Mapping rules

.NET does not support constant types at a global level, so all constants must be defined within a class or interface. Any CORBA consts defined at the global or module level map to a `value` field that represents the value of the const and is contained in a special .NET interface. (This is analagous to the IDL-to-Java mapping for consts.)

Example

The example can be broken down as follows:

1. Consider the following OMG IDL:

```
// OMG IDL
const string str = "foo";
module A
{
    const float flt = 123.45;
    module B
    {
        const short shrt = 678;
    };
};
```

2. The preceding OMG IDL maps, for example, to the following C# interface:

```
// C#
interface str
{
    public static String value = "foo";
};
namespace A
{
    interface flt
    {
        public static Single value = 123.45;
    };
    namespace B
    {
        interface shrt
        {
            public static Int16 value = 678;
        };
    };
};
```

OMG IDL constants defined at interface, struct, union, or exception level map to constants (that is, literal fields) within the mapped type.

.NET Connector Configuration

This chapter describes the configuration variables that are specific to the .NET Connector, and their associated values.

In this chapter

This chapter discusses the following topics:

Overview	page 126
Configuration Variables	page 127

Overview

Configuration domains

Configuration variables are stored in a configuration domain. A configuration domain can be based on one of two distinct configuration models, depending on whether your deployment needs are small scale or large scale. For small-scale deployment, you can implement a configuration domain as an ASCII text file that is stored locally on each machine and edited directly. For large-scale deployment, Orbix provides a distributed configuration repository server that enables centralized configuration for all applications spread across a network.

The OrbixRemoting: Scope

Almost all configuration variables specific to the .NET Connector are scoped within an `OrbixRemoting:` namespace.

See the *CORBA Administrator's Guide* for details of CORBA configuration variables.

Configuration Variables

Overview

This section describes the configuration variables associated with the .NET Connector component of Orbix. All of these variables are scoped within the `OrbixRemoting:TypeMan:` namespace. This section discusses the following topics:

- "TYPEMAN_CACHE_FILE" on page 127.
- "TYPEMAN_DISK_CACHE_SIZE" on page 127.
- "TYPEMAN_MEM_CACHE_SIZE" on page 128.
- "TYPEMAN_IFR_IOR_FILENAME" on page 128.
- "TYPEMAN_IFR_NS_NAME" on page 129.
- "TYPEMAN_READONLY" on page 129.

TYPEMAN_CACHE_FILE

This default setting for this variable is:

```
OrbixRemoting:TypeMan:TYPEMAN_CACHE_FILE="%{LOCAL_COMET_CACHE_
    FILE}";
```

The .NET Connector uses a memory and disk cache for efficient access to type information. This entry specifies the name and location of the file used. It is automatically set by the configuration script.

TYPEMAN_DISK_CACHE_SIZE

The default setting for this variable is:

```
OrbixRemoting:TypeMan:TYPEMAN_DISK_CACHE_SIZE="2000"
```

This variable is used in conjunction with `TYPEMAN_MEM_CACHE_SIZE`. It specifies the maximum number of entries allowed in the disk cache. When this value is exceeded, entries can be flushed from the cache. The nature of the applications using the bridge affects the value that should be assigned to this variable. However, as a general rule, the disk cache size should be about eight to ten times greater than the the memory cache. (See "TYPEMAN_MEM_CACHE_SIZE" on page 128 for more details about setting the maximum number of entries for the memory cache.)

A cache "entry" in this case corresponds to a user-defined type. For example, a union defined in OMG IDL results in one entry in the cache. An interface containing the definition of a structure results in two entries.

A good rule of thumb is that 1000 cache entries (given a representative cross section of user-defined types) corresponds to approximately 2 megabytes of disk space. Therefore, the default disk cache size of 2000 allows for a maximum disk cache file size of approximately 4 megabytes.

TYPEMAN_MEM_CACHE_SIZE

The default setting for this variable is:

```
OrbixRemoting:TypeMan:TYPEMAN_MEM_CACHE_SIZE="250"
```

This variable is used in conjunction with `TYPEMAN_DISK_CACHE_SIZE`. It specifies the maximum number of entries allowed in the memory cache. When this value is exceeded, entries can be flushed from the cache. The nature of the applications using the bridge affects the value that should be assigned to this variable. However, as a general rule, the disk cache size should be about eight to ten times greater than the the memory cache. Furthermore, to avoid unnecessary swapping into and out from disk, you should ensure the memory cache size is no smaller than 100. See "TYPEMAN_DISK_CACHE_SIZE" on page 127 for more details.

TYPEMAN_IFR_IOR_FILENAME

The default setting for this variable is:

```
OrbixRemoting:TypeMan:TYPEMAN_IFR_IOR_FILENAME=" "
```

When the dynamic marshalling engine in the .NET Connector encounters a type for which it cannot find corresponding type information in the type store, it must then retrieve the type information from the Interface Repository (IFR). The order in which the .NET Connector attempts to connect to the IFR is as follows:

- If a name is specified in the `OrbixRemoting:TypeMan:TYPEMAN_IFR_NS_NAME` variable, the .NET Connector looks up that name in the Naming Service to connect to the IFR.

- If a name is not specified in `OrbixRemoting:TypeMan:TYPEMAN_IFR_NS_NAME`, the .NET Connector checks to see if an IOR is specified in the `initial_references:InterfaceRepository:reference` variable. If so, it uses the IFR associated with that IOR.
- If an IOR is not specified in `initial_references:InterfaceRepository:reference`, the .NET Connector checks to see if a filename is specified in the `TYPEMAN_IFR_IOR_FILENAME` variable.

Consequently, you must set the `TYPEMAN_IFR_IOR_FILENAME` variable if you do not set `OrbixRemoting:TypeMan:TYPEMAN:IFR_NS_NAME` or `initial_references:InterfaceRepository:reference`. In this case, the value required is the full pathname to the file that contains the IOR for the IFR you want to use.

TYPEMAN_IFR_NS_NAME

The default setting for this variable is:

```
OrbixRemoting:TypeMan:TYPEMAN_IFR_NS_NAME=" "
```

This variable is needed if you are using the Naming Service to resolve the IFR. It specifies the name of the IFR in the Naming Service. You should register an IOR for the IFR in the Naming Service under a compound name. This variable should contain that compound name. As explained in "TYPEMAN_IFR_IOR_FILENAME" on page 128, this is the first configuration variable that the .NET Connector always checks if it needs to contact the IFR for type information that it cannot find in the type store.

TYPEMAN_READONLY

The default setting for this variable is:

```
OrbixRemoting:TypeMan:TYPEMAN_READONLY="no"
```

The valid settings for this variable are:

"no"	This means that clients have write access to the type store.
"yes"	This means that clients have readonly access to the type store.

This variable specifies whether clients have write access or readonly access to the type store. Because the .NET Connector bridge runs in-process to each client, there is a local copy of the type store on each client machine. If you want the local cache of type information to be locked, so that it cannot be further added to, you can set this variable to `"yes"`.

.NET Connector Utility Arguments

This chapter describes the various arguments that are available with the itts2il and ittypeman command-line utilities.

In This Chapter

This chapter discusses the following topics:

Itts2il Argument Details	page 132
Ittypeman Argument Details	page 135

Itts2il Argument Details

Overview

This section describes the arguments available with the `itts2il` utility. The `itts2il` utility performs two main functions:

- Generation of .NET metadata—this can be controlled via the `-f`, `-a`, `-m`, and `-i` arguments.
- Management of the type store—this can be controlled via the `-e`, `-c`, and `-w` arguments.

Usage string

You can call up the usage string for `itts2il` as follows:

```
itts2il -?
```

The usage string for `itts2il` is:

```
Usage: [options] <type name> [[<type name>] …]
         -f : file name (defaults to <type name #1>.dll)
         -a : assembly name (defaults to <type name #1>)
         -m : module name (defaults to <type name #1>)
         -i : always connect to the IFR
         -e : lookup and cache type entries from the IFR
              (use "*" to look up the entire IFR)
         -c : list the type store contents
         -w : wipe the type store cache clean
         -v : verbose mode
```

Specifying commands

When specifying an `itts2il` command, it is important that *all* command arguments precede any specified *type* names. This is because any arguments that are specified after a type name are ignored, because they are then also assumed to be type names. For example, in the following command, `itts2il` can recognize the `-v` argument, to run in verbose mode:

```
itts2il -i -v Grid
```

However, in the following example, `itts2il` cannot recognize `-v` as an argument and wrongly assumes it is simply a type called `-v`:

```
itts2il -i Grid -v
```

Summary of arguments for generating metadata

The arguments available with `itts2il` for the purposes of controlling metadata generation are:

`-f` This specifies the filename of the generated .NET metadata DLL. If you do not specify the `-f` argument, the generated DLL filename is based by default on the specified IDL interface name, with a `.dll` extension. This argument should be qualified with the name you want to assign to the DLL file. For example, the following command generates a .NET metadata DLL file called `myfile.dll` that contains an assembly called `test` with metadata corresponding to the `Grid` IDL interface:

```
itts2il -a test -f myfile.dll Grid
```

`-a` This specifies the name of the assembly contained in the generated .NET metadata DLL. If you do not specify the `-a` argument, the assembly name is based by default on the specified IDL interface name. This argument should be qualified with the name you want to assign to the assembly. For example, the following command generates an assembly called `test` that contains metadata corresponding to the `Grid` IDL interface:

```
itts2il -a test Grid
```

`-m` This specifies the module name in the generated .NET metadata DLL assembly manifest. If you do not specify the `-m` argument, the generated module name is based by default on the specified IDL interface name. This argument should be qualified with the name you want to assign.

`-i` By default, `itts2il` always queries the host's local typestore cache when generating the .NET metadata DLL. This argument instructs `itts2il` to query the IFR instead of the local typestore cache, to ensure that the most up-to-date type information is being used. You should specify this argument if the IDL in the IFR has changed since the typestore was last primed.

Note: Most development scenarios can simply accept the default for the .NET metatadata DLL, assembly, and module names.

Summary of arguments for managing type store

The arguments available with `itts2il` for the purposes of controlling typestore management are:

`-e` This instructs `itts2il` to prime the local typestore cache with type information from the IFR. You can qualify `-e` with an individual OMG IDL interface name, a series of names separated by spaces, or an asterisk (`*`) to prime the cache with the entire contents of the IFR. See "Adding New Information to the Type Store" on page 44 for details of how to specify each.

If you specify an OMG IDL interface name that is not already in the cache, `itts2il` looks up the IFR to obtain the relevant type information before copying it to the cache.

`-c` This allows you to view the contents of the type store disk cache.

`-w` This wipes the type store contents. This means that it empties the disk cache data files. The disk cache data files include:

- `typeman._dc`—The disk cache data file.
- `typeman.idc`—The disk cache index.
- `typeman.edc`—The disk cache empty record index.
- `typeman.map`—The UUID name mapper file.

Note: An alternative method of emptying the disk cache data files is to enter the following command (assuming, for example, that the `typeman` data files are held in `c:\temp`):

del c:\temp\typeman.*

The `OrbixRemoting.TypeMan.TYPEMAN_CACHE_FILE` configuration variable determines where the data files are stored.

The -v argument

The `-v` argument indicates that the utility is to run in verbose mode (that is, diagnostic messages are written to standard output). You can specify the `-v` argument in either of the following ways:

- As an independent argument, for example:

```
itts2il -w -v
```

- As an appendage to other arguments to make them verbose, for example:

```
itts2il -wv
```

Ittypeman Argument Details

Overview

This section describes the arguments available with the `ittypeman` utility.

> **Note:** The `ittypeman` utility is used in advanced management and diagnostics of the type store. It is not needed during typical development scenarios. It is provided primarily to assist in debugging.

Usage String

You can call up the usage string for `ittypeman` as follows:

```
ittypeman -?
```

The usage string for `ittypeman` is:

```
Usage:

TypeMan [filename | -e name|uuid|TLBName] [-v[s[i] method]]
        [options]

        filename: Name of input text file.
        -e:    Look up entry (name, {uuid} or type library
               pathname).
        -c[n][u]: List disk cache contents, n: Natural order,
                  u: display uuid.
        -w[m]: Delete (wipe) cache contents. [m]: Delete uuid-
               mapper contents.
        -f:    List type store data files.
        -r:    Resolve all references (use to generate static
               bridge compatible names for CORBA sequences).
        -i:    Always connect to IFR (for performance
               comparisons).
        -v[s[i] method]: Log v-table for interface/struct.
                         [s:search for method].
                         [i]: Ignore case. Use -v with -e option.
        -b:    Log mem cache hash-table bucket sizes.
        -h:    Log cache hits/misses.
        -z:    Log mem cache size after each addition.
        -l[+]: Log TS basic contents ['+' shows new's/delete’s].

        -?2: Priming input file format info.
```

Summary of arguments

The arguments available with `ittypeman` are:

`-b` This allows you to view the bucket sizes in the memory cache hash table.

`-c` **Note:** This provides the same functionality as `itts2il -c`.

This allows you to view the contents of the type store disk cache.

If you want to view the contents in the order in which they have been added to the cache, you can specify `-cn` instead. If you want to view the UUID of each type listed, you can specify `-cu` instead. (Every type in the type store has an associated UUID. The .NET Connector generates UUIDs for OMG IDL types, using the MD5 algorithm, as specified by the OMG.)

`-e` **Note:** This provides the same functionality as `itts2il -e`.

This instructs `ittypeman` to search the Interface Repository (IFR) for a specific item of type information, and then add it to the type store cache. You can qualify `-e` with an individual OMG IDL interface name, a series of names separated by spaces, or an asterisk (`*`) to prime the cache with the entire contents of the IFR. See "Adding New Information to the Type Store" on page 44 for details of how to specify each.

If you specify an OMG IDL interface name that is not already in the cache, `ittypeman` looks up the IFR to obtain the relevant type information before copying it to the cache.

`-f` This allows you to view the type store data files. These include the disk cache data file (`ittypeman._dc`), the disk cache index file (`ittypeman.idc`), the disk cache empty record index file (`ittypeman.edc`), and the UUID name mapper file (`ittypeman.map`).

`-h` This instructs `ittypeman` to display `"Cache miss"` on the screen, if a type it is looking for is not already in the cache. If the type is already in the cache, `ittypeman` displays `"Mem cache hit"` on the screen.

`-i` **Note:** This provides the same functionality as `itts2il -i`.

This instructs `ittypeman` to always query the IFR for an item of OMG IDL type information. This can be used to compare the performance of different ORBs, and so on.

`-l` This logs the type store basic contents to the screen. Enter `-l+` to log newly added and deleted entries.

`-r` This generates static bridge compatible names for OMG IDL sequences.

-v This allows you to view the v-table contents for an interface or struct. This option provides output such as the following:

```
Name Sorted          V-table         DispId  Offset
balance         get  makeLodgement    1       0
makeLodgement        makeWithdrawal   2       1
makeWithdrawal       balance          3       2
overdraftLimit  get  overdraftLimit   4       3
```

-w **Note:** This provides the same functionality as `itts2il -w`.

This wipes the type store contents. This means that it empties the disk cache data files.

If you also want to empty the UUID name mapper file (`ittypeman.map`), you can specify `-wm` instead. Wiping the type store contents is useful when you want to reprime the cache. You might want to reprime the cache, for example, if it contains type information for an interface that has subsequently been modified.

-z This allows you to view the actual size to which the memory cache temporarily grows when `ittypeman` is loading in a containing type (such as a module) to retrieve a contained type (such as an interface within that module).

-? This outputs the usage string for `ittypeman`.

-?2 This allows you to view the format of the entries that you can include in a text file, which you can specify with the `-e` option, if you want to prime the cache simultaneously with any number and combination of type names.

Advanced Topics

This chapter provides details of topics that might be of interest to more advanced users of the .NET Connector, including an explanation of the difference between static .NET metadata and dynamic runtime type information, and a description of how to programatically enable advanced CORBA features.

In this chapter

This chapter discusses the following topics:

.NET Metadata versus Type Store Information	page 140
Enabling Advanced CORBA Features	page 142

.NET Metadata versus Type Store Information

Overview

This section explains the distinction between static .NET metadata generated at compile time and type store information obtained at runtime.

Graphical overview

Figure 8 provides a graphical overview of the usage of both the static .NET metadata and dynamic runtime type information that are required to enable .NET client invocations on remote CORBA objects.

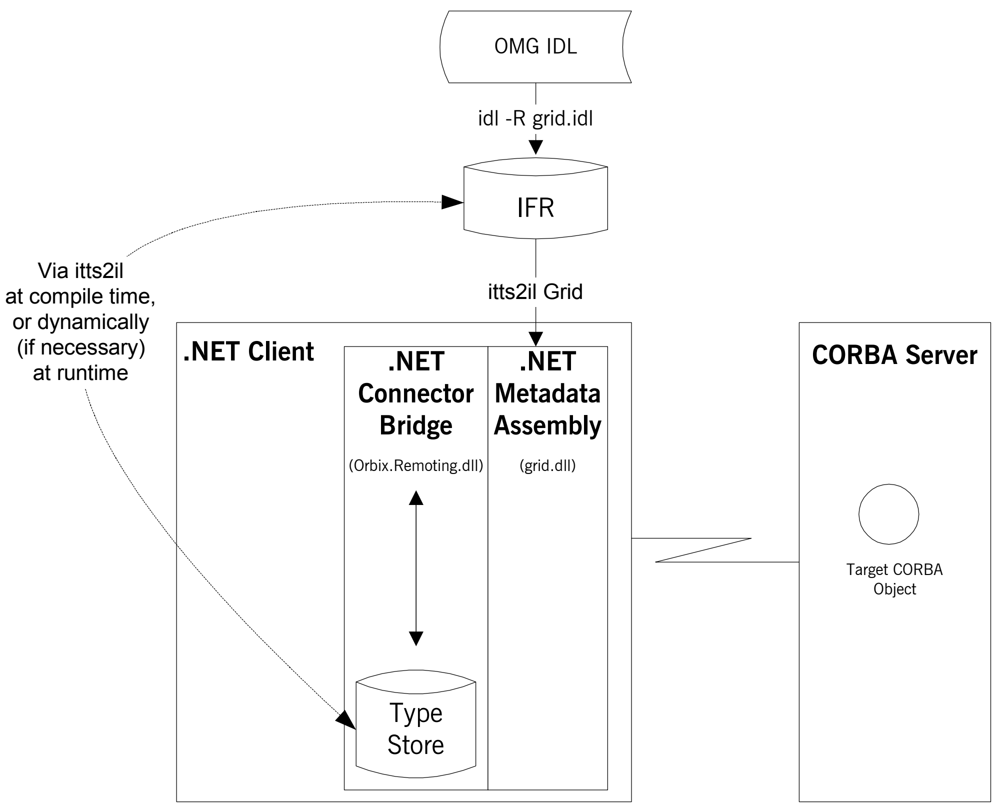

Figure 8: *.NET Metadata and Dynamic Type Information Usage*

Explanation

The .NET Connector uses two distinct stores of type information, both of which are required to enable a .NET client to communicate with a CORBA server, and both of which are managed via the `itts2il` utility. These stores of type information are:

- The .NET metadata assembly DLL.
- The type information in the type store.

As a starting point, on the CORBA side, the OMG IDL that defines the interfaces to your target CORBA objects must first be registered in an Interface Repository (IFR), using the `idl -R` filename command (where *`filename`* represents the OMG IDL filename). This is necessary because:

- The `itts2il` utility (at compile time) obtains the type information it needs from the IFR to generate .NET metadata and automatically prime the type store cache.
- The type store (at runtime) obtains from the IFR any required type information not currently in the type store cache.

The .NET metadata assembly DLL stores type information required by the .NET framework. .NET metadata must be generated from the OMG IDL defined for the target CORBA objects, so .NET clients have a .NET interface to those objects. At application runtime, the client uses the .NET metadata to make method calls on the remote target CORBA object. As far as the client is concerned, it is making a method call on a remote .NET object.

The type store cache stores type information in a format useful to CORBA. When a client makes a method call, the .NET Connector bridge (that is, the `Orbix.Remoting` remoting channel) intercepts the client request and attempts to obtain the type information corresponding to the client request from the type store cache. The bridge follows this pattern when attempting to obtain type information:

1. Check the type store memory cache, which is populated on application start-up with the most recently accessed type information in the type store.
2. If the type information is not in the memory cache, look for it in the type store disk cache.
3. If the type information is not in the disk cache, look for it in the IFR.

The bridge then converts the .NET client request to a CORBA request that it subsequently marshals across the network.

Enabling Advanced CORBA Features

Overview

This section describes how you can programatically enable advanced CORBA features, by simply defining in your .NET client code the configuration scope used by the custom remoting channel. This in turn provides a simple but dynamic means of enabling your .NET applications to avail of powerful CORBA client-side features, such as Secure Sockets Layer (SSL) security, Quality-of-Service (QOS), portable interceptors, and so on.

Graphical overview

Figure 9 shows how CORBA client-side features can be implemented as plug-ins to the .NET Connector bridge for use by .NET clients.

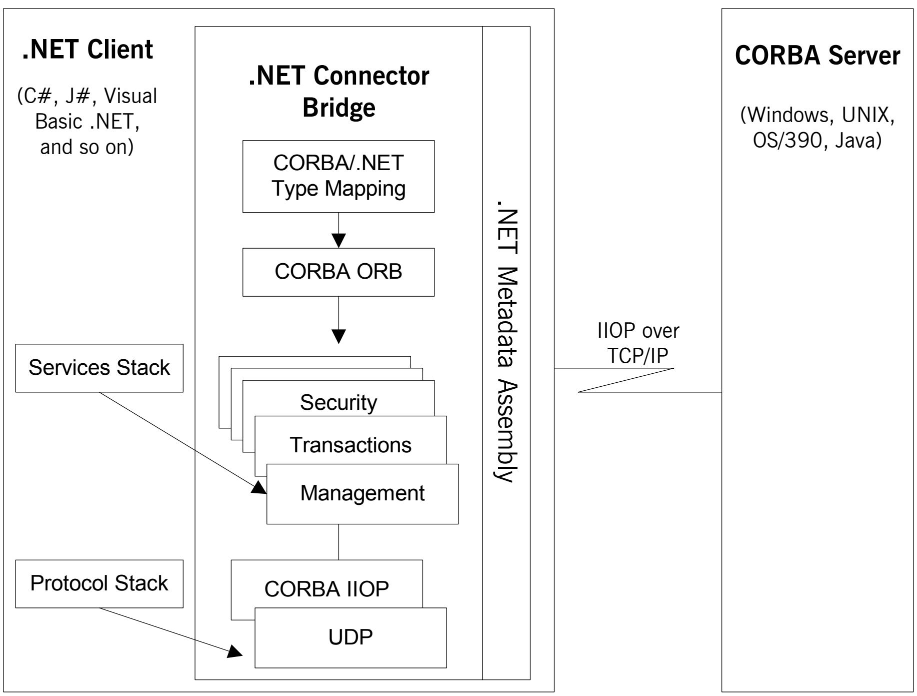

Figure 9: *CORBA Features as Plug-Ins to Remoting Channel*

Deploying advanced features

To deploy advanced CORBA features for use by your .NET clients:

1. Set up a configuration scope in your configuration domain that uses the `orb_plugins` and and `client_binding_list` configuration variables to specify the advanced plug-in features you want to use. For example:

   ```
   remoting_orbname
   {
        binding:client_binding_list = ["GIOP+GIOP_SNOOP+IIOP"];
        orb_plugins = ["iiop_profile", "giop", "giop_snoop",
           "iiop"];
        plugins:giop_snoop:verbosity = "4";
        plugins:giop_snoop:shlib_name = "it_giop_snoop";
   };
   ```

   The preceding configuration scope, for example, is called `remoting_orbname` and it enables the GIOP snoop feature to enable the output of diagnostic information on IIOP calls between .NET clients and CORBA servers.

2. In your .NET client code, pass the name of the relevant configuration scope as a string to the `OrbixRemotingChannel` constructor. The following is an example of how to do this in C#, based on the preceding configuration scope example:

   ```
   // C#
   ChannelServices.RegisterChannel(new OrbixClientChannel
      ("remoting_orbname"));
   ```

   The following is a corresponding example of how to do this in Visual Basic .NET:

   ```
   ' Visual Basic .NET
   ChannelServices.RegisterChannel(New OrbixClientChannel
      ("remoting_orbname"))
   ```

Index

Symbols
.NET clients
  implementing in C# 25
  implementing in Visual Basic .NET 22
  introduction to 12
.NET metadata, creating from OMG IDL 38

A
abstract interfaces (in OMG IDL) 77
any type (in OMG IDL) 80
  CORBA-to-.NET mapping 120
array type (in OMG IDL) 90
attributes (in OMG IDL) 65

B
basic types (in OMG IDL) 79
  CORBA-to-.NET mapping 101
bitwise operators 97
bridge, introduction to 11
built-in types (in OMG IDL) 79

C
C#, writing clients in 25
caching mechanism 42
callbacks 29
  generating stub code for 34
  implementing 31
char type (in OMG IDL) 80
clients. *See* .NET clients
command options 131
commands
  itts2il 132
  ittypeman 135
configuration domain 126
configuration repository 126
configuration variables 125–??
  TYPEMAN_CACHE_FILE 127
  TYPEMAN_DISK_CACHE_SIZE 127
  TYPEMAN_IFR_IOR_FILENAME 128
  TYPEMAN_IFR_NS_NAME 129
  TYPEMAN_MEM_CACHE_SIZE 128
  TYPEMAN_READONLY 129
constant definitions (in OMG IDL) 94
constant expressions (in OMG IDL) 97
constant fixed types (in OMG IDL) 84
  CORBA-to-.NET mapping 122
context clause (in OMG IDL) 120
CORBA complex types 106
CORBA servers, introduction to 12
CORBA-to-.NET mapping
  basic types 101
  constants 122
  exceptions 113
  interfaces 103
  modules 121
  object references 120
  strings 102
  structs 107
  unions 109

D
data types, defining in OMG IDL 93
decimal fractions 84
disk cache 43

E
empty interfaces (in OMG IDL) 67
enum type
  in OMG IDL 86
  ordinal values of 86
exceptions (in OMG IDL) 66
  See also system exceptions, user exceptions
  CORBA-to-.NET mapping 113
extended built-in types (in OMG IDL) 82

F
fixed type (in OMG IDL) 83
floating point type (in OMG IDL) 79
forward declaration of interfaces (in OMG IDL) 73

I

IDL
  abstract interfaces 77
  arrays 90
  attributes 65
  built-in types 79
  constant definitions 94
  constant expressions 97
  creating .NET metadata from 38
  empty interfaces 67
  enum type 86
  exceptions 66
  extended built-in types 82
  forward declaration of interfaces 73
  inheritance redefinition 72
  interface inheritance 68
  local interfaces 74
  modules and name scoping 57
  multiple inheritance 69
  object interface inheritance 71
  operations 62
  pseudo object types 92
  registering 44
  sequence type 91
  struct type 87
  structure 56
  union type 88
  valuetypes 76
IIOP 10
implementing
  callbacks 31
  server for client callbacks 36
inheritance (in OMG IDL)
  CORBA-to-.NET mapping 105
inheritance redefinition (in OMG IDL) 72
interface (in OMG IDL)
  CORBA-to-.NET mapping 103
interface inheritance (in OMG IDL) 68
itts2il
  location of 37
  options 132
ittypeman
  location of 37
  options 135

L

local interfaces in IDL 74
local object pseudo-operations 75
long double type in IDL 83
long long type in IDL 82

M

memory cache 43
module (in OMG IDL)
  CORBA-to-.NET mapping 121
modules and name scoping in IDL 57
multiple inheritance in IDL 69

O

object interface inheritance in IDL 71
object references
  CORBA-to-.NET mapping 120
octet type (in OMG IDL) 80
OMG IDL *See* IDL
operations (in OMG IDL) 62

P

protocols, introduction to 10
pseudo object types (in OMG IDL) 92

S

sequence type (in OMG IDL) 91
servers, implementing for client callbacks 36
string type (in OMG IDL) 80
  CORBA-to-.NET mapping 102
struct type (in OMG IDL) 87
  CORBA-to-.NET mapping 107

T

type store
  adding OMG IDL to 44
  caching mechanism 42
  central role of 40
  creating .NET metadata from 38
  deleting contents of 46
  dumping contents of 47
  priming 44

U

union type (in OMG IDL) 88
  CORBA-to-.NET mapping 109

V
valuetypes (in OMG IDL) 76
Visual Basic .NET, writing clients in 22

W
wchar type (in OMG IDL) 83
wstring type (in OMG IDL) 83