Borland
VisiBroker™ 8.0
VisiBroker for .Net
Developer's Guide

Refer to the file deploy.html for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Microsoft, the .NET logo, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

Borland®

Contents

Chapter 11
Using the Transaction service   **107**

Chapter 12
Using the Security service   **121**

Chapter 13
Using VisiBroker for .NET with Partially
Trusted Applications   **131**

Chapter 14
Using VisiBroker for .NET with COM   **135**

Chapter 15
Using VisiBroker for .NET with Borland
GateKeeper   **143**

Appendix A
Compiler options   **149**

Appendix B
IDL to C# mapping   **155**

Appendix C
Java built-in type support 171

Index 179

1

Understanding the VisiBroker for .NET model

This chapter introduces the VisiBroker for .NET components, and it describes the technologies with which VisiBroker for .NET lets your applications interoperate.

What is VisiBroker for .NET?

The VisiBroker for .NET product provides high performance connectivity between the Microsoft .NET runtime and J2EE and CORBA components. This product allows you to build managed client-side and server-side applications developed for the .NET Framework (and ASP.NET applications) that can access heterogeneous server-side components via IIOP, the highly scalable, interoperable and secure communications protocol.

Figure 1.1 illustrates how a deployment with VisiBroker for .NET-powered applications might look. The left and right sides of the figure show two .NET application environments, the top ones running stand-alone .NET applications, and the others running ASP.NET hosted applications. In the middle of the diagram the J2EE and CORBA server environments are shown.

The functionality provided by VisiBroker for .NET, the client stubs, server skeletons, and the VisiBroker for .NET runtime, is displayed in the shaded areas. Note that there is no shading in the J2EE and CORBA server environments, indicating that VisiBroker for .NET does not need to be deployed into the server environment in order to interoperate with the .NET environment.

Figure 1.1   VisiBroker for .NET client-side deployment diagram
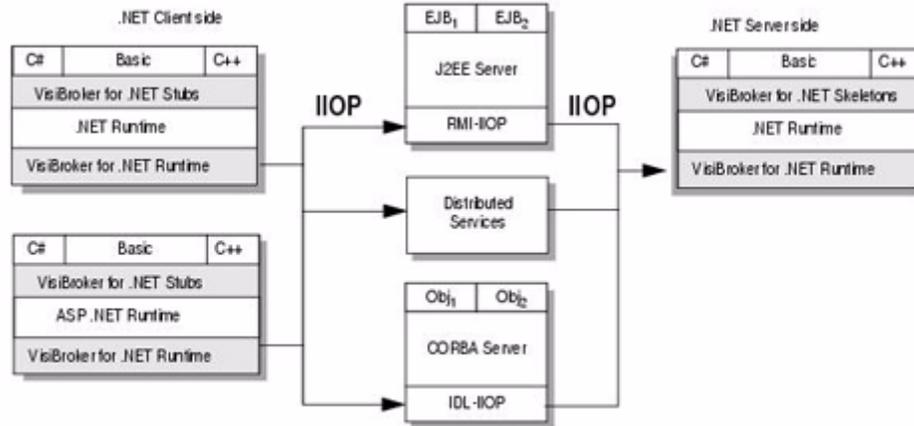
VisiBroker for .NET developer tools

Stubs and *skeletons* are required for VisiBroker for .NET-powered applications to invoke methods on J2EE and CORBA objects. Stubs and skeletons are interface-specific objects that provide parameter marshaling and communication for an application to invoke methods on an object that is running in a different execution environment. The VisiBroker for .NET developer tools provide you with compilers to generate the stubs and skeletons needed to communicate with your CORBA and J2EE server objects.

The J2EE-based compiler reads interfaces specified in Java Remote Method Invocation (RMI) files. The CORBA-based compiler reads interfaces specified in Interface Definition Language (IDL) files. The resulting stubs and skeletons target the .NET Common Type System (CTS), Microsoft's language-neutral type system. Although the compilers generate stubs and skeletons in the C# programming language, once the C# stub or skeleton is compiled into the Microsoft Common Intermediate Language (CIL) by a C# compiler it is usable from any .NET-compatible language.

VisiBroker for .NET runtime

The VisiBroker for .NET runtime is a collection of libraries and network resources that integrates within end-user applications, and allows your applications to locate and use objects. The runtime exposes the basic CORBA and J2EE APIs required for using remote objects. These APIs are compliant with the Microsoft Common Language System, and are therefore accessible to any .NET programming language.

The VisiBroker for .NET runtime provides the following capabilities:

– **Marshaling**—a high-performance, scalable engine for reading and writing IIOP packets.

– **Connection management**—controls the allocation of TCP connections and other communication resources.

– **Security**—encryption and authentication of messages based on the widely adopted standards: SSL, TLS, X.509, etc. (Note that this enables secure connectivity to any J2EE 1.3 compatible product.)

– **Objects-by-value**—allows arbitrarily complex data types to be passed across client-server boundaries (for J2EE 1.3 products).

– **Invocation context propagation**—provides the ability to augment IIOP packets with system-level data.

– **Portable interceptors**—provides the ability to augment IIOP packets with user- or system-level data. This is particularly important for products that provide distributed transaction support based on the OTS and XA specifications. (Note that interoperable transaction support is optional in J2EE 1.3, and is therefore only provided by a subset of J2EE vendors.)

VisiBroker for .NET features

The VisiBroker for .NET product provides the following capabilities:

– **High Performance:** VisiBroker for .NET provides binary data formatting by using IIOP for client-server networking.

– **Stateful services:** VisiBroker for .NET provides a full distributed object model, which can support arbitrary server-side components and arbitrary life-cycle requirements.

– **Advanced security support**: Encryption, authentication and authorization are all supported in VisiBroker for .NET, based on the latest security standards.

– **Support for complex data types:** Using VisiBroker for .NET, data conversions are handled automatically, which is both more efficient and less error prone.

– **Enterprise Quality of Service:** VisiBroker for .NET provides advanced QoS out of the box, including:

  – **Load balancing:** The ability to fan-out requests to a collection of service providers.

  – **Fault tolerance:** The ability to redirect requests from a failed server to an alternate provider.

  – **Transactions:** The ability to propagate two-phase-commit transaction contexts across application boundaries and start transactions on the client side.

  – **Scalability:** The ability to control the lifetime of connections, multiplex over connections, etc., for optimizing resource utilization.

What is .NET?

Microsoft .NET provides developers with a single approach to build both desktop applications and Web-based applications. It also enables developers to use the same tools and skills to develop software for a variety of systems, using a variety of programming languages, and it can minimize conflicts between applications by helping incompatible software components coexist.

The .NET Framework consists of the .NET Framework class library (FCL), for building .NET applications, and the *common language runtime* (CLR), for running them.

Common language runtime

The common language runtime (CLR) is the runtime engine in the Microsoft .NET Framework for executing applications. The CLR also provides *managed* applications with services such as cross-language integration, code access security, object lifetime management, and debugging and profiling support.

Programs can be written for the CLR in just about every language, including C#, C++, Microsoft Visual Basic, and JScript. The runtime simplifies programming by assisting with many mundane tasks of writing code. These tasks include memory management—which can be a big generator of bugs—security management, and error handling.

When it is compiled using a compiler in a .NET language, the code written in your programming language of choice is compiled into an assembly-like language called *common intermediate language* (CIL). The CIL is compiled down to executable code by the common language runtime at execution time.

.NET Framework class library

Programmers who write Windows applications are familiar with the Windows API, standard class libraries, and functions or classes of their own. The .NET Framework class library (FCL) includes prepackaged sets of functionality that developers can use to build applications that use the types, methods, and properties that target the common language runtime. Writing code using the types provided in the FCL is the surest way to have completely interoperable .NET applications.

Some of the features included in the FCL are:

– ASP.NET to help build Web applications and Web services.

– Windows Forms for client user interface development.

– ADO.NET to help connect applications to databases.

.NET Remoting

Distributed applications are traditionally based on DCOM, CORBA, and Java RMI remoting technologies using binary protocols, such as IIOP, that utilize network bandwidth efficiently. In contrast, much of .NET interoperability centers on XML and SOAP.

The VisiBroker for .NET runtime provides a *managed* code implementation of IIOP for the .NET Framework. VisiBroker for .NET allows you, the developer, to locate and call methods on remote objects using .NET Remoting-style calls, shielding you from having to learn how to write CORBA or Java RMI-style calls. See application development examples in "Developing VisiBroker for .NET client applications," and "Developing VisiBroker for .NET Remoting servers."

Managed vs. Unmanaged Applications

The .NET Framework supports what it calls *managed* and *unmanaged* applications. Managed applications are programs that you create using a supported .NET language, such as C#, and which adhere to various rules imposed by the Framework. All VisiBroker for .NET code is managed code.

Unmanaged applications are programs created in unsupported languages, or which do not completely adhere to .NET Framework rules. These applications, many of which are legacy applications, can still be run within a wrapper process provided by the .NET Framework.

What is J2EE?

Java 2 Platform, Enterprise Edition (J2EE) technology and its component based model simplifies enterprise development and deployment. The J2EE platform manages the infrastructure and supports the Web services to enable development of secure, robust and interoperable business applications. J2EE consists of several APIs to implement Enterprise JavaBeans, Java Servlets, Java Server Pages, and JDBC for database access, among many others.

J2EE simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically. J2EE takes advantage of CORBA technology for interaction with existing enterprise resources.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) technology gives developers the ability to model a wide range of objects useful in the enterprise by defining two distinct types of EJB components: Session Beans and Entity Beans. Session Beans represent behaviors associated with client sessions. Entity Beans represent collections of data, such as records in a database, and encapsulate operations on the data they represent. Entity

Beans are intended to be persistent, surviving as long as the data they're associated with.

Client applications communicate with EJBs using strictly standardized EJBHome and EJBObject interfaces to locate, instantiate, and invoke methods on remote objects. You can use the VisiBroker for .NET developer tools to generate all the code needed to communicate with the EJBs, from its Java RMI source to the .NET compatible C# language.

Java RMI

Java Remote Method Invocation (RMI) technology allows developers to work completely in the Java programming language to produce Java technology-based distributed applications. There is no separate Interface Definition Language (IDL) or mapping to learn. Java RMI technology that is run over Internet Inter-Orb Protocol (RMI-IIOP) delivers CORBA distributed computing capabilities to the J2EE platform.

Like CORBA, RMI-IIOP is based on open standards defined with the participation of hundreds of vendors and users in the Object Management Group. Like CORBA, RMI-IIOP uses IIOP as its communication protocol. IIOP eases legacy application and platform integration by allowing application components written in C++, C, COBOL, and other CORBA supported languages to communicate with components running on the Java platform.

What is CORBA?

Common Object Request Broker Architecture (CORBA) is an architectural specification that provides the capability for distributed applications to interoperate without understanding detailed communication requirements on one end or the other. CORBA is based on open standards defined with the participation of hundreds of vendors and users in the Object Management Group.

A common model of a CORBA application is a typical client-server model, with the exception that it uses a middle layer, known as *middleware*, or more specifically, an Object Request Broker (ORB). An ORB is a collection of services that manage interactions between distributed applications.

Interface Definition Language

The Interface Definition Language (IDL) is a descriptive language you use to describe your CORBA interfaces to remote objects. You use an IDL compiler to generate a client stub file and a server skeleton file in your implementation language, usually C++, Java, C#, or another high-level language. The Object Management Group (OMG) has defined specifications for language mappings to a variety of programming languages. VisiBroker for .NET provides a language mapping for IDL in C#. See "IDL to C# mapping," for more information.

You can write your IDL code in any IDE but you need an IDL compiler to generate .NET compatible stubs and skeletons. Using the VisiBroker for .NET developer tools, you can use one of the IDL compilers included to generate the C# client stub from an IDL file. The IDL compiler reads the IDL file and generates a class or other addressable object that includes stubs, which are general methods that accept a simple message request from an application. The stub passes the request to the object implementation, on the server for example, and, on receiving a response, decodes the response and returns the results to the calling application, or client.

The VisiBroker for .NET features comply with the CORBA specification (version 2.4) from the Object Management Group (OMG) and are interoperable with Borland AppServer.

CORBA and .NET Remoting

Much of .NET interoperability centers on XML and SOAP. While these technologies have their strengths, primarily in being able to use connectionless protocols, such as HTTP, they have serious drawbacks when it comes to synchronous communications.

In those cases, using peer-to-peer protocols, such as IIOP, are more efficient and secure. Additionally, using synchronous client-server communication allows you to pass binary data across a more tightly-coupled system, providing more data security and recovery capabilities.

VisiBroker for .NET allows you to bootstrap to the CORBA middleware, and locate objects using either CORBA-style calls or .NET Remoting calls in your client code. See examples of application development in "Developing VisiBroker for .NET client applications," and "Developing VisiBroker for .NET Remoting servers."

Microsoft Visual Studio .NET options

If you selected the Microsoft Visual Studio .NET component when you installed VisiBroker for .NET, your Visual Studio environment will have some extra elements to make your VisiBroker for .NET application development go smoothly.

To configure the VisiBroker for .NET options in Visual Studio:

1 Select the Tools menu and click Options

2 Select the Borland VisiBroker for .NET options group

The following configuration options are available:

– Installation directory—the directory where the VisiBroker for .NET components are installed.

– JRE directory—the directory where the Java Runtime Environment is installed.

– Supported file extensions—displays the VisiBroker for .NET compiler for each supported file extension.

– Defaults—allows you to configure default command line arguments for each of the VisiBroker for .NET compilers. For descriptions of command line arguments see "Compiler options."

2

Developing VisiBroker for .NET client applications

This chapter introduces the development process for creating .NET client applications that can access J2EE and CORBA server objects using the VisiBroker for .NET runtime. Simple examples are provided to illustrate the three different methods for making calls on remote objects.

VisiBroker for .NET provides you with three methods for developing client applications that communicate with distributed objects: .NET Remoting, CORBA, and J2EE. These three technologies each define a standard way of doing essentially the same steps: bootstrap the middleware, locate and instantiate remote objects, and invoke methods on them.

The syntax, APIs, and programming models are slightly different for each of the three technologies, but the following examples will prove that whichever way you write it you can accomplish the same result with each of them.

Where do I go from here?

If you are a Microsoft developer, already comfortable with .NET Remoting, or new to distributed technologies, start with "A simple .NET Remoting example". Developers familiar with J2EE should start with "A simple J2EE example", and those familiar with CORBA should start with "A simple CORBA example".

Some simple examples

The following sections show you some simple examples of the three methods you can use to bootstrap the middleware, locate and instantiate remote objects, and invoke methods on them.

A simple .NET Remoting example

If you are a Microsoft developer, already comfortable with .NET Remoting, or new to distributed technologies, you will be pleased to learn that you can develop .NET applications that interoperate with objects on both J2EE and CORBA servers using the .NET Remoting programming model.

The following three lines of code show how easily you can instantiate the remote object `MyServer` and call a `Method()` on it.

```
static void Main(string[] args) {
    RemotingConfiguration.Configure ("MyApplication.exe.config");
    MyServerHome myServerHome = new MyServerHomeRemotingProxy();
    MyServer myServer = myServerHome.Create();
    myServer.Method();
}
```

The information for establishing a connection with the server and locating the remote object are contained in an XML configuration file, as shown in ".NET Remoting configuration".

Let's walk through the example line by line:

The first line specifies the configuration file where the .NET Remoting is configured.

```
RemotingConfiguration.Configure ("MyApplication.exe.config");
```

The next line of code instantiates the *factory* object `MyServerHome`.

```
MyServerHome myServerHome = new MyServerHomeRemotingProxy();
```

A factory object is a lookup mechanism for locating and creating a remote object. You look it up first in order to locate and create an instance of the actual object you want to invoke methods on.

There is no concept of *narrowing* an object's type in .NET. You locate the object and cast it to its specific type all in one step.

The next line creates an instance of `myServer`.

```
MyServer myServer = myServerHome.Create();
```

You can now call methods on your instance of `myServer`.

```
myServer.Method();
```

It's that simple! If you want more information on configuring .NET Remoting using the VisiBroker for .NET protocol see ".NET Remoting configuration".

A simple J2EE example

VisiBroker for .NET provides a method for allowing developers familiar with writing calls to EJBs to do so in the .NET application.

Consider the following example.

```
static void Main(string[] args) {
    J2EE.Naming.Context root = new J2EE.Naming.InitialContext(args);
    string serverName = "location/of/my/server";
    object myServerHomeObject = root.Lookup(serverName);
    MyServerHome myServerHome = (MyServerHome)
        J2EE.Rmi.PortableRemoteObject.Narrow(myServerHomeObject,
            typeof(MyServerHome));
    MyServer myServer = myServerHome.Create();
    myServer.Method();
}
```

As you can see this is somewhat more complex than the .NET Remoting example. There is no configuration file in which to hide the details required for locating the objects.

Let's walk through the example line by line:

In the first line we establish the root context for the J2EE naming service.

```
J2EE.Naming.Context root = new J2EE.Naming.InitialContext(args);
```

The next two lines declare a variable to contain the location of the EJBHome object (`myServerHomeObject`) on the server, and look it up.

```
string serverName = "location/of/my/server";
```

```
object myServerHomeObject = root.Lookup(serverName);
```

The next line narrows `myServerHomeObject` to its type, `MyServerHome`.

```
MyServerHome myServerHome = (MyServerHome)
   J2EE.Rmi.PortableRemoteObject.Narrow(myServerHomeObject,
typeof(MyServerHome));
```

The next line creates an instance of `myServer`.

```
MyServer myServer = myServerHome.Create();
```

Finally we can invoke a method on `MyServer`.

```
myServer.Method();
```

A simple CORBA example

VisiBroker for .NET provides a method for allowing developers familiar with writing calls to CORBA objects to do so in the .NET application.

The following example shows the calls you might make.

```
static void Main(string[] args) {
   CORBA.ORB orb = CORBA.ORB.Init(args);
   CORBA.Object rootObject = orb.ResolveInitialReferences("NameService");
   CosNaming.NamingContextExt root =
      CosNaming.NamingContextExtHelper.Narrow(rootObject);
   string serverName = "location/of/my/server";
   CORBA.Object myServerHomeObject = root.ResolveStr(serverName);
   MyServerHome myServerHome = MyServerHomeHelper.Narrow(myServerHomeObject);
   MyServer myServer = myServerHome.Create();
   myServer.Method();
}
```

As you can see this is somewhat more complex than the .NET Remoting example. There is no configuration file in which to hide the details required for locating the objects.

Let's walk through the example line by line:

In the first line we initialize the ORB.

```
CORBA.ORB orb = CORBA.ORB.Init(args);
```

In the next two lines we obtain the root context for the CORBA naming service.

```
CORBA.Object rootObject = orb.ResolveInitialReferences("NameService");
CosNaming.NamingContextExt root =
CosNaming.NamingContextExtHelper.Narrow(rootObject);
```

The next two lines declare a variable to contain the location of the factory object (`myServerHomeObject`) on the server, and look it up.

```
string serverName = "location/of/my/server";
CORBA.Object myServerHomeObject = root.ResolveStr(serverName);
```

The next line narrows `myServerHomeObject` to its type, `MyServerHome`.

```
MyServerHome myServerHome = MyServerHomeHelper.Narrow(myServerHomeObject);
```

The next line creates an instance of `myServer`.

```
MyServer myServer = myServerHome.Create();
```

Finally we can invoke a method on `MyServer`.

```
myServer.Method();
```

.NET Remoting configuration

This section contains the details of the configuration file alluded to in the .NET example under "A simple .NET Remoting example".

Let's recall the .NET Remoting example:

```
static void Main(string[] args) {
    RemotingConfiguration.Configure ("MyApplication.exe.config");
    MyServerHome myServerHome = new MyServerHomeRemotingProxy();
    MyServer myServer = myServerHome.Create();
    myServer.Method();
}
```

The information for establishing a connection with the server and locating the remote object are hidden away in an XML configuration file. This technique is known as declarative activation in .NET.

A configuration file for our example might look like the following:

```
<configuration>
    <system.runtime.remoting>
        <application name="MyApplication">
            <client>
                <wellknown type="MyServerHomeRemotingProxy, MyApplicationAssembly"
                    url="janeva:corbaname:rir:#location/of/my/server/object"/>
            </client>
            <channels>
                <channel type="Janeva.Remoting.IiopChannel,
                    Borland.Janeva.Runtime"/>
            </channels>
        </application>
    </system.runtime.remoting>
</configuration>
```

Specifying the object location

When we instantiated `MyServerHome` in the first line of the example, we used the `new` operator on `MyServerHomeRemotingProxy()`. In order to locate the object on which to make the call, the example configuration file uses the `wellknown` element,

```
<wellknown type="MyServerHomeRemotingProxy, MyApplication"
    url="janeva:corbaname:rir:#location/of/my/server/object"/>
```

where `MyServerHomeRemotingProxy` is the type name and `MyApplication` is the name of the assembly where the type is defined.

Note:

MyServerHome is represented as a `wellknown` object (also known as Server-activated object or SAO). Any CORBA or EJB server object can be represented as an SAO. In addition, EJBs can be represented as Client Activated Objects (CAO). See "Client-activated objects vs. server-activated objects" for more information.

The .NET programming model requires that you locate the remote object with a URL. URLs are formed with two parts:

– The `janeva:` protocol prefix tells the application to use the IIOP channel (`Janeva.Remoting.IiopChannel`), specified in the `<channel>` element of the configuration file.

  Typically in .NET the first part of the URL contains the communication protocol. VisiBroker for .NET extends .NET Remoting with a new protocol: CORBA IIOP.

– `corbaname:rir:#location/of/my/server/object` is one of several CORBA ORB `string_to_object()` compatible URL schemes. See the following table in the next session for more examples and descriptions of the URL schemes.

URL schemes

To address the problem of bootstrapping and allow for more convenient exchange of human-readable object references, VisiBroker for .NET allows URLs in the formats listed in the following table to be converted into object references

Specifying the Remoting channel

URL scheme	Examples	Description

corbaname:	`janeva:corbaname:rir:#location/of/my/server/object`
	or
	`janeva:corbaname:rir:<NS_host>:<NS_port>#location/of/my/server/object`
		The `corbaname` URL scheme is most often used to resolve EJBs. It allows URLs to denote entries in a Naming Service. The host address is the location and listening port of the Naming Service and it can be formatted as <NS_host_name>:<NS_port> or <NS_ip_address>:<NS_port>. More details about the `corbaname` URL scheme are available in the OMG CORBA specification.

corbaloc:	`janeva:corbaloc:rir:<host>:<port>/object_key`	The `corbaloc` URL scheme provides direct access to server objects by location and object key. It is not often used because of the limited amount of addressing power. More details about the `corbaloc` URL scheme are available in the OMG CORBA specification.

osagent:	`janeva:osagent:poa:<poa_name>:<object_id>[:<server_host_name>]`
	or
	`janeva:osagent:repid:<interface_repository_id>[:<object_name>][:<server_host_name>]`
		The `osagent` scheme is a private feature for using with Borland VisiBroker CORBA server objects.
		To avoid ambiguity, all colons (:) in the <interface_repository_id> must be prefixed with the backslash (\) character, as follows:
		`janeva:osagent:repid:IDL\:com/semagroup/targys/servicelayer/corba/ServiceRootI\:1.0:SL_demo_server`

IOR:	`janeva:IOR:<stringified_object_reference>`	The `IOR` URL scheme allows you to look up an object by stringified object reference (IOR).
http:	janeva:http:<host_address>`/location/of/my/ior/file`	The HTTP URL scheme points to a text file containing the stringified object reference.
file:	`janeva:file:<host_address>`/location/of/my/ior/file`	The file URL scheme points to a text file containing the stringified object reference.

To communicate with remote objects a .NET client application has to create and register a Remoting *channel*. The channel provides a conduit for communication between a client and a remote object.

Instead of using the .NET Framework `Channels` types, VisiBroker for .NET provides the `Janeva.Remoting.IiopChannel` type for creating a channel on IIOP.

```
<channel type="Janeva.Remoting.IiopChannel, Borland.Janeva.Runtime"/>
```

The second argument is the VisiBroker for .NET runtime assembly name.

Client-activated objects vs. server-activated objects

VisiBroker for .NET supports both types of activation for remotable objects:

– **Server activation**. Server-activated objects (SAO) are created by the server only when they are needed. They are not created when the client proxy is created by calling `new` or `Activator.GetObject`, but rather when the client invokes the first method on that proxy. The previous sections in this chapter are examples of this object activation method.

– **Client activation**. Use client-activated objects when the application needs to retain state between method calls and also needs to pair each client with a unique object instance. Client-activated objects (CAO) are created on the server when the client calls `new` or `Activator.CreateInstance`.

Any kind of remote object supported by VisiBroker for .NET can be used on the client-side as an SAO. In addition, a J2EE server object can also be represented as a CAO.

The client activation in VisiBroker for .NET is based on the fact that many J2EE components follow the factory design pattern. Namely, any remotely accessible EJB (that is, stateful or stateless session or entity bean) exposes a home interface which is used to create or resolve the bean instance. For EJBs configured as CAOs, VisiBroker for .NET allows you to skip resolving the home interface and to create or resolve the bean instance simply by creating an instance of bean's proxy class.

For example, let's consider a simple EJB interface, `SimpleSession`, and its home interface, `SimpleSessionHome`:

```
public interface SimpleSession extends javax.ejb.EJBObject {
   public void ping() throws java.rmi.RemoteException;
}

public interface SimpleSessionHome extends javax.ejb.EJBHome {
   public SimpleSession create(String name);
}
```

The `SimpleSession` interface configured as an SAO can be accessed on the client side in C# as follows:

```
SimpleSessionHome home = new SimpleSessionHomeRemotingProxy();
SimpleSession session = home.Create("my name");
session.Method();
```

If the `SimpleSession` interface is represented as a CAO, the client code is a bit simpler:

```
SimpleSession session = new SimpleSessionRemotingProxy("my name");
session.Method();
```

Now, let's explore in detail how VisiBroker for .NET supports the client activation model for J2EE components.

First, the `java2cs` compiler has extended knowledge of the EJB home interface. The compiler maps some methods defined on the EJB home interface to constructors of the bean's Remoting Proxy class in the generated C# code. For the Session EJB home (stateful or stateless), these are any `create()` methods. For the Entity EJB home, this is the `findByPrimaryKey()` method. Also, the `java2cs` compiler preserves the parameters of the original `home` method in the generated proxy constructor. For example, the `SimpleSessionHome.create(String name)` method maps to the `SimpleSessionRemotingProxy(string name)` constructor in the generated C# code.

When a new instance of the CAO Remoting Proxy is created, the VisiBroker for .NET runtime does a few things under the covers. First, it resolves the bean's home interface based on the VisiBroker for .NET URL specified in the Remoting object configuration. Then, depending on whether this is a Session bean or an Entity bean, the runtime remotely calls either the corresponding Session's `create()` method, or the Entity's `findByPrimaryKey()` method. Lastly, the Remoting Proxy of the EJB instance, resulted by this call, becomes an object returned by the new statement.

While the VisiBroker for .NET CAO usage model resembles the original .NET Remoting CAO model quite closely, it is worth noting a few peculiarities:

1 Creating an EJB Remoting Proxy, configured as a CAO, does not always imply that a new EJB instance is created on the server side (the EJB container). While this is true for the Session beans, the Entity beans behave differently. For Entities, the CAO constructor call translates into the `findByPrimaryKey()` call, therefore an existing instance with the corresponding primary key must already exist, otherwise an exception will be thrown. Thus, the CAO representation of the Entity bean can be used only to resolve a bean instance, not to create one. To create a new Entity instance use the SAO model.

2 VisiBroker for .NET Client-activated objects do not support the lifetime lease model. This is due to the fact that the EJB models this concept. Moreover, the EJB life cycle is different depending on the EJB type. The client-side developer needs to understand these differences and explicitly call the `Remove()` method on the EJB interface or the home when the EJB instance is no longer needed.

A configuration file for a CAO example should look like the following:

```
<configuration>
   <system.runtime.remoting>
      <application name="MyApplication">
         <client url="janeva:corbaname:rir:#location/of/my/server/object">
            <activated type="SimpleSessionRemotingProxy,
               MyApplicationAssembly"/>
         </client>
         <channels>
            <channel type="Janeva.Remoting.IiopChannel,
               Borland.Janeva.Runtime"/>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Programmatic activation

An alternative to configuration files for writing your .NET calls to server side objects is to activate the Remoting channel and specify the location of the remote object directly in the code. The following code sample shows how this might look for an SAO.

```
static void Main(string[] args) {
   Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
   System.Runtime.Remoting.Channels.ChannelServices.RegisterChannel(channel);
   string objectUrl = "janeva:corbaname:rir:#" +
      "location/of/my/server/object";
   MyServerHome myServerHome = (MyServerHome)
      System.Activator.GetObject(typeof(MyServerHome), objectUrl);
   MyServer myServer = myServerHome.Create();
   myServer.Method();
}
```

The following code sample shows how this might look for a CAO.

```
static void Main(string[] args) {
   Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
   System.Runtime.Remoting.Channels.ChannelServices.RegisterChannel(channel);
   string objectUrl = "janeva:corbaname:rir:#" +
      "location/of/my/server/object";
   MyServer myServer = (MyServer) System.Activator.CreateInstance(
      typeof(MyServerRemotingProxy), new object[] {"my name"});
   myServer.Method();
}
```

The first two lines in each example deal with setting up the VisiBroker for .NET Remoting channel on IIOP.

```
Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
System.Runtime.Remoting.Channels.ChannelServices.RegisterChannel(channel);
```

The third line declares a variable to contain the location of the factory object
(`myServerHomeObject`) on the server, and look it up similar to the way it was done in the
J2EE and CORBA examples in the previous sections, except that there is no narrowing
in .NET.

3

Configuring properties

There are three ways to set VisiBroker for .NET properties. These are given below in order of priority, from highest to lowest.

1 Using command-line arguments

2 Setting properties programmatically

3 Using a configuration file

Note

The settings with higher priority override the settings with lower priority. For example, the properties set at the command-line override the properties defined programmatically.

Setting properties at the command-line

If you are running a VisiBroker for .NET application from a command prompt, then you may specify VisiBroker for .NET properties as space-delimited key-value pairs, and the key is preceded by a hyphen (-). For example:

```
Client -ORBInitRef NameService=corbaloc:iiop:1.2@host1:3075/NameService
```

In the application code, for developers who use the VisiBroker for .NET style API, the command line arguments can be passed into the corresponding version of the `Janeva.Remoting.IiopChannel()` constructor. For example:

```
static void Main(string[] args) {
  Janeva.Remoting.IiopChannel channel = new Janeva.Remoting.IiopChannel(args);
  …
}
```

For developers using the CORBA style API, pass these arguments to the static `ORB.Init()` constructor:

```
static void Main(string[] args) {
  CORBA.ORB orb = CORBA.ORB.Init(args);
  ...
}
```

For J2EE developers, VisiBroker for .NET supports an equivalent ORB initialization API using `J2EE.Naming.InitialContext()`. For example, suppose your J2EE server is running on the local host with a Naming Service listening to port 2809. Your client can use the `-ORBInitRef` style initialization to point to the Naming Service as follows:

```
Client -ORBInitRef NameService=corbaname:iiop:localhost:2809/NameService
```

In you application code, you simply pass these arguments to the static `J2EE.Naming.InitialContext` constructor:

```
static void Main(string[] args) {
  J2EE.Naming.Context context =
    J2EE.Naming.InitialContext(args);
...
}
```

Setting properties programmatically

You can store VisiBroker for .NET properties in a `System.Collections.Hashtable` object, and pass these to either `CORBA.ORB.Init()`, `J2EE.Naming.InitialContext()`, or `Janeva.Remoting.IiopChannel()`. This provides a cleaner approach to setting VisiBroker for .NET properties than the command-line approach and is useful when the command-line is not available.

The .NET Remoting developer may pass the `Hashtable` settings into the appropriate version of the `Janeva.Remoting.IiopChannel` constructor:

```
static void Main(string[] args) {
   System.Collections.Hashtable props = new System.Collections.Hashtable();
   props.Add("ORBInitRef",
     "NameService=corbaloc:iiop:1.2@host1:3075/NameService");
   props.Add("janeva.transaction", true);
   Janeva.Remoting.IiopChannel channel =
     new Janeva.Remoting.IiopChannel(args. props);

// other code here
   ...
}
```

The following CORBA example creates a `Hashtable` object and sets three properties:

```
static void Main(string[] args) {
   System.Collections.Hashtable props = new System.Collections.Hashtable();
   props.Add("ORBInitRef",
     "NameService=corbaloc:iiop:1.2@host1:3075/NameService");
   props.Add("janeva.transactions", true);
   CORBA.ORB orb = CORBA.ORB.Init(args, props);

// other code here
   ...
}
```

For J2EE developers, you may also use a `Hashtable` to initialize the application:

```
static void Main(string[] args) {
   System.Collections.Hashtable props = new System.Collections.Hashtable();
   props.Add("ORBInitRef",
     "NameService=corbaloc:iiop:1.2@host1:3075/NameService");
   props.Add("janeva.transactions", true);
   J2EE.Naming.InitialContext context = new J2EE.Naming.InitialContext(props);

// other code here
   ...
}
```

Setting properties within a configuration file

VisiBroker for .NET properties can be set by using a configuration file.

Important

The configuration file section <janeva> is renamed to <visinet> in VisiBroker for .NET 7.0. However, for backward compatibility with older versions, the section name <janeva> is still supported.

Properly named, the configuration file is loaded automatically. For ASP.NET applications, this is the Web.config file. For other applications, this is the <app_assembly_name>.exe.config file located in the same directory where the <app_assembly_name>.exe is.

Note

In Microsoft Visual Studio .NET you must add a file called app.config to your project to get the appropriately named XML configuration file included in your build.

The example below shows a sample configuration file.

```
<configuration>
  <configSections>
    <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
  </configSections>
  <visinet>
    <transactions enabled="true"/>
    <server defaultPort="10000">
      <remoting enabled="true"/>
    </server>
  </visinet>
</configuration>
```

Notice that all of the VisiBroker for .NET settings are grouped under the <visinet> section in the configuration file. Since the VisiBroker for .NET settings are not part of the standard .NET configuration XML, it is important to instruct the .NET runtime to read the <visinet> XML. This is achieved by adding the <configSections> section as it is demonstrated in the example above.

VisiBroker for .NET property descriptions

Each VisiBroker for .NET property has a counterpart setting in the configuration file. The following sections describe each VisiBroker for .NET property and the corresponding configuration file setting in detail.

Resolving the Naming Service

The following property is used to resolve the Naming Service.

ORBInitRef

Type: string

Default value: none

XML:

```
<naming url="NameService=URL" />
```

Each application server has its own URL syntax as shown in the following table.

Application server	Naming Service URL format
WebLogic 7 or 8	corbaloc::localhost:7001/NameService
IBM WebSphere 5	corbaname:iiop:localhost:2809/NameServiceServerRoot
Oracle's 0C4J:	corbaloc:iiop:1.2@localhost:5555/NameService
Sybase	corbaloc:iiop:1.2@localhost:9000/NameService

Note also that the default port number may vary for your deployment.

Note

Resolving the naming service for Borland AppServer is automatic (based on OSAgent), so this configuration is optional for Borland AppServer. Other application servers require this configuration.

Examples

To resolve the Naming Service using the command line the argument should be in the following format:

```
> client -ORBInitRef NameService=corbaloc::localhost:7001/NameService
```

The property setting in the configuration file would resemble the following example.

```
<visinet>
   <naming url="corbaloc::localhost:7001/NameService" />
</visinet>
```

Licensing property

This property is configured to enable the VisiBroker for .NET runtime to locate the license if necessary.

janeva.license.dir

Set the path to the directory where the VisiBroker for .NET license file is located. The path can be absolute or relative to the current directory.

Type: string

Default value: none

XML:

```
<license dir="path" />
```

Example

The following example sets the `janeva.license.dir` property in a configuration file.

```
<visinet>
   <license dir="C:\Program Files\Borland\Janeva" />
</visinet>
```

Transactions properties

These properties are configured to enable VisiBroker for .NET transaction support.

janeva.transactions

Set this property to `true` to enable support of the client-demarcated transactions. Keep in mind that it is impossible to start a new transaction without turning this feature on. Namely, the `orb.ResolveInitialReferences("TransactionCurrent")` call will fail if transactions are not enabled.

This feature is disabled by default, as, when enabled, it adds an additional performance overhead during a remote invocation.

Type: boolean [true|false]

Default value: false

XML:

```
<transactions enabled="value" />
```

Note

If the `<transactions>` section is present in the configuration file, and the `enabled` attribute is missing, the default VisiBroker for .NET behavior is to enable transactions.

Example

The following example configurations set the `janeva.transactions` property to `true`.

```
<visinet>
   <transactions enabled="true" />
</visinet>

<visinet>
   <transactions />
</visinet>
```

janeva.transactions.factory.url

This URL points to a transaction service Current factory.

Type: string

Default value: none

XML:

```
<transactions>
   <factory url="corbaloc::URL" />
</transactions>
```

Example

The following example configuration sets the `janeva.transactions.factory.url` property.

```
<visinet>
   <transactions enabled="true">
      <factory url="corbaloc::localhost:6666/TransactionFactory" />
   </transactions>
</visinet>
```

Server-side properties

These properties are used to configure VisiBroker for .NET server-side support.

janeva.server.defaultPort

This property sets the port on which a VisiBroker for .NET server listens to for IIOP requests. The value 0 (zero) means that the system will pick a random port number.

Type: integer

Default value: 0 (zero)

XML:

```
<server defaultPort="value">
```

janeva.server.remoting

This property is configured when using remoting-style callbacks and remoting-style VisiBroker for .NET servers. If set to `true`, then remoting-style callbacks and remoting-style VisiBroker for .NET servers are enabled.

This feature is disabled by default. When enabled, it adds an additional performance overhead during a remote invocation.

Type: boolean [true|false]

Default value: false

XML:

```
<server><remoting enabled="value" /></server>
```

Example

The following example sets the `janeva.server.port` and the `janeva.server.remoting` properties in a configuration file.

```
<visinet>
   <server defaultPort="2809">
      <remoting enabled="true" />
   </server>
</visinet>
```

Interoperability property

This property is used to configure various VisiBroker for .NET interoperability aspects.

janeva.interop.jvmType

This property controls how VisiBroker for .NET writes certain data types on the wire. It specifies the JVM on the receiving side of the outgoing communication. This is pertinent when communicating with a server running on Java. When communicating between a .NET client and .NET server this property must be set to the same value on both sides.

Type: integer [1|2|3]

Default value: 1

XML:

```
<interop jvmType="value" />
```

Note that the marshaling format for various data types evolves over time as the JDK changes. In order for VisiBroker for .NET to be able to write such changing data types, this flag can be used to indicate which type of VM you are inter-operating with.

Currently there are three valid setting for this flag:

1 A value of 1 indicates that you are using a version 1.1, 1.2 or 1.3 JVM.

2 A value of 2 indicates that you are using a version 1.4.0 or 1.4.1 JVM

3 A value of 3 indicates that you are using a version 1.4.2 or later JVM.

The main difference between JVM Type 1 and 2 is the format for writing an instance of:

```
java.lang.Random
java.math.BigDecimal
java.math.BigInteger
```

This format changed in JDK version 1.4.0, and if you need to send such data from a VisiBroker for .NET process to a Java process, you must set this flag appropriately.

The main difference between JVM Type 2 and 3 is the format for writing an instance of:

```
java.util.Vector
java.util.Stack
```

This format changed in JDK version 1.4.2, and if you need send such data from a VisiBroker for .NET process to a Java process, you must set this flag appropriately.

A few notes on JVM interoperability:

– The `janeva.interop.jvmType` property only affects the *write* side of VisiBroker for .NET.

– The VisiBroker for .NET *read* side always supports all JVMs. So, it is possible to receive `Random`, `Vector`, and `Stack` instances from J2EE applications running on any JVM irrespective of the setting for the `jvmType` flag. Only when the VisiBroker for .NET process needs to send such objects to a J2EE application will the `jvmType` need to be specified.

Example

The following example sets the `janeva.interop.jvmType` property in a configuration file.

```
<visinet>
   <interop jvmType="2"/>
</visinet>
```

Security properties

These properties are used to configure VisiBroker for .NET security support.

janeva.security

Set this property to `true` to enable VisiBroker for .NET security support.

This feature is disabled by default. When enabled, it adds an additional performance overhead during a remote invocation.

Type: boolean [true|false]

Default value: false

XML:

```
<security enabled="value"/>
```

Note

If the `<security>` section is present in the configuration file, and the `enabled` attribute is missing, the default VisiBroker for .NET behavior is to enable security.

janeva.security.username

This property configures the user name for the security identity passed to the server-side for authentication. This property is used in conjunction with the `janeva.security.password` property.

Type: string

Default value: none

XML:

```
<security><identity><username>value</username></identity></security>
```

janeva.security.password

Specifies the password in the clear text format.

Type: string

Default value: none

XML:

```
<security>
   <identity>
```

```
        <password>value</password>
    </identity>
</security>
```

janeva.security.realm

This is the authentication realm to be used in conjunction with the user name and password elements in the security identity configuration. By default, users belong to the security realm called default. This property should be set when using an authentication realm other than a realm called default.

Type: string

Default value: default

XML:

```
<security>
    <identity>
        <realm>value</realm>
    </identity>
</security>
```

janeva.security.certificate

This property sets the certificate used for authentication. The expected value is a string representing the friendly name of the certificate located in the Windows Certificate Store.

Type: string

Default value: none

XML:

```
<security><identity><certificate>value</certificate></identity></security>
```

Examples

The following example sets the janeva.security.username, janeva.security.password and janeva.security.realm properties for the security identity in a configuration file.

```
<visinet>
    <security enabled="true">
        <identity>
            <username>admin</username>
            <password>foobar</password>
            <realm>MyRealm</realm>
        </identity>
    </security>
</visinet>
```

The following example sets the janeva.security.certificate property for the security identity in a configuration file.

```
<visinet>
    <security enabled="true">
        <identity>
            <certificate>joeshopper</certificate>
        </identity>
    </security>
</visinet>
```

Server-side security properties

These properties are used to configure VisiBroker for .NET server-side security.

janeva.security.server

Set this property to `true` to enable VisiBroker for .NET server-side security support.

This feature is disabled by default, as, when enabled, it adds an additional performance overhead during a remote invocation.

Type: boolean [true|false]

Default value: false

XML:

```
<security>
   <server enabled="value"/>
</security>
```

Note

If the `<security><server>` section is present in the configuration file, and the `enabled` attribute is missing, the default VisiBroker for .NET behavior is to enable server-side security.

janeva.security.server.defaultPort

Configures the port to be used for SSL over IIOP.

Type: integer

Default value: none

XML:

```
<security>
   <server defaultPort="value"/>
</security>
```

janeva.security.server.certificate

This property specifies the friendly name of the certificate. If a certificate is specified in this section, then it will be used to identify the server peer of the SSL connection. Note, that if value for this setting is not provided, the VisiBroker for .NET runtime will try to use a certificate provided in the `janeva.security.certificate` setting. If neither of these settings is specified, VisiBroker for .NET runtime considers this as a bad configuration and fails to initialize.

Type: string

Default value: none

XML:

```
<security>
   <server>
      <certificate>value</certificate>
   </server>
</security>
```

Exampvole

The following example sets the server-side security properties in a configuration file.

```
<visinet>
   <security>
      <server enabled="true" defaultPort="15000">
         <certificate>Book Store</certificate>
      </server>
   </security>
</visinet>
```

Firewall property

This property is used to enable the VisiBroker for .NET firewall support.

janeva.firewall

Enables support of the high-level firewall gateway such as the Borland Gatekeeper.

This feature is disabled by default, as, when enabled, it adds an additional performance overhead during a remote invocation.

Type: boolean [true|false]

Default value: false

XML:

```
<firewall enabled="value"/>
```

Note

If the `<firewall>` section is present in the configuration file, and the `enabled` attribute is missing, the default VisiBroker for .NET behavior is to enable the firewall.

Example

The following example sets the `janeva.firewall` property in a configuration file.

```
<visinet>
   <firewall enabled="true"/>
</visinet>
```

Portable Interceptor property

This property is used to configure the portable interceptor.

janeva.orb.init

Specifies the portable interceptor that needs to be loaded by the ORB. If the portable interceptor is part of the same assembly containing the main class, then you can just specify the class name. If the portable interceptor is part of an assembly outside of the assembly containing the main class, then you need to specify the strongly-named assembly name. You may specify as many portable interceptors as you wish.

Type: string

Default value: none

XML:

```
<orb>
   <init type="value"/>
</orb>
```

Example

The following example sets the `janeva.orb.init` property in a configuration file.

```
<visinet>
   <orb>
      <init type="MyInterceptor, MyInterceptorAssembly, version=1.2.3.4,
        culture=neutral, publicKeyToken=xxxx"/>
      <init type="MyInterceptor2"/>
   </orb>
</visinet>
```

Format for Specifying IOR

Some VisiBroker properties are provided to the applications as means to specify some IOR. For examples,

```
vbroker.orb.gatekeeper.ior
vbroker.firewall.<custome_name>.ior
vbroker.orb.initRef
vbroker.orb.defaultInitRef
vbroker.orb.activationIor
```

The values of them are strings that will in turn can be passed as the only parameter to the ORB method string_to_object() and therfore what formats can be set as the values of those properties will really depend on what formats string_to_object() of that ORB can accept.

The followings are various formats that are acceptable as string_to_object() parameter of Visinet ORBs.

Format	Visinet

URL of a text file containing stringified IOR
 	
Examples:
`http://myhost:8088/ns.ior`
`ftp://myhost:13001/bank.ior`
`file:///C:/bank.ior`
	Supported: The protocols set is as per .NET system

Corbaloc: Corbaloc machinery depends on the locate-forward exception being thrown by a conventional IIOP listen point, whose host-port is specified in the string. This will not work in any of the following common situations: For security reason, all clear IIOP listener ports are disabled and all listeners are SSL. The required IIOP listen point is not directly reachable and some firewall traversal is required. Example: `corbaloc::myhost:33001/MyService`	Supported

Stringified IOR
 	
Example:
`IOR:000000000000001c49444c3a42616e`
`6b2f4163636f756e744d616e616765723a`
`312e30000000000100000000000000a800`
`0102000000000f3134332e3138362e3134`
`312e353800000a9e00000000002b00504d`
`43000000040000000f2f62616e6b5f706f`
`727461626c6500200000000b42616e6b4d`
`616e6167657200000000004564953300000`
`0010000023830000000100000000000000`
`0056495303000000500070801ff000000`
`000000000000000080000000056495300000`
`0000010000001800000000000100010000`
`00010501000100010109000000000`
	Supported

VisiBroker Smart Agent properties

These properties are configured when you are using the Smart Agent (OSAgent) for object registration and lookup.

janeva.agent

This property can be used to disable the Smart Agent.

Type: boolean [true|false]

Default value: false

XML:

```
<agent enabled="value"/>
```

janeva.agent.port

This property sets the port used by the Smart Agent.

Type: integer

Default value: 14000

XML:

```
<agent port="value"/>
```

janeva.agent.addr

This property specifies the physical location of the Smart Agent, either by IP address or hostname. If not provided, VisiBroker for .NET will look for any Smart Agent on the network with the proper port during the ping. Providing a host address will reduce network traffic, as VisiBroker for .NET will ping the Smart Agent on the provided host address and port.

Type: string

Default value: none

XML:

```
<agent addr="value"/>
```

Example

The following example configuration file sets the `janeva.agent janeva.agent.port` and `janeva.agent.addr` properties.

```
<visinet>
   <agent enabled="true" port="14001" addr="localhost.localdomain.com"/>
</visinet>
```

Setting VisiBroker properties

VisiBroker for .NET supports all of the properties originally introduced in the Borland VisiBroker line of products. Among these properties are the settings used to fine-tune the firewall support. In a configuration file you can specify the VisiBroker properties as key-value attributes in the `<vbroker>` section.

The following example show how to set some VisiBroker GateKeeper properties in a configuration file.

```
<visinet>
   <firewall enabled="true"/>
   <vbroker
      vbroker.orb.alwaysProxy="true"
      vbroker.orb.gatekeeper.ior="ior:…"
   />
```

```
</visinet>
```

4

Building and deploying VisiBroker for .NET applications

This chapter describes the process for building and deploying your VisiBroker for .NET-powered .NET applications. It contains the following topics:

- Generating VisiBroker for .NET stubs and skeletons
- Adding references to VisiBroker for .NET runtime libraries
- Deploying VisiBroker for .NET applications

Generating VisiBroker for .NET stubs and skeletons

The J2EE and CORBA technologies define object-level interfaces, and communication between your .NET applications and server objects is conducted exclusively through these interfaces. In CORBA these interfaces are defined in IDL, in J2EE they are defined in Java RMI.

The VisiBroker for .NET `java2cs` and `idl2cs` tools convert the interfaces from Java RMI or IDL into C#. VisiBroker for .NET adds features to the Microsoft Visual Studio .NET so that you can configure and use these tools in your IDE projects. You can also use the command line to compile the interfaces.

Visual Studio

To generate VisiBroker for .NET stubs and skeletons in Visual Studio .NET:

1   Add an IDL, JAR, or EAR to your Visual Studio project.

2   Select the file and confirm the VisiBroker for .NET properties as shown in Figure 4.1.

   Figure 4.1   Microsoft Visual Studio .NET VisiBroker for .NET properties

For an IDL file the compiler should be set to IDL2cs. A JAR or EAR file should use the Java2cs compiler. You can add compiler arguments and rename the output file in the properties dialog.

Important:

If you are generating the server skeleton code be sure to add the `-servant` compiler flag to the compiler arguments.

3 To compile just the interface file, right-click the file in the Solution Explorer and select Build and Browse.

If the compile is successful, it should generate a C# file and add it to your project.

command line

To use the compilers at the command line, make sure that the tools are available in your path so that it can be run from the command prompt. The compilers are located in the bin directory of the VisiBroker for .NET installation directory. To test whether the compilers are in your path, open a command prompt and type `idl2cs`. You should get a listing of compiler switches.

If you did not add it during the installation process, you can add `idl2cs` to your path from the command prompt by typing

```
prompt> set PATH=<VisiBroker Home>\VisBroker.NET\bin;%PATH%
```

Once you've confirmed that the compilers are in your path, you can use them:

```
prompt> idl2cs Example.idl
```

If the compile is successful, it should generate a C# file.

Adding references to VisiBroker for .NET runtime libraries

In order to take advantage of the VisiBroker for .NET runtime, applications must refer to the VisiBroker for .NET DLLs. The following sections describe how to add references to the VisiBroker for .NET runtime libraries in your applications.

Visual Studio

To add references to the VisiBroker for .NET runtime libraries in Visual Studio .NET:

1 Right-click the References node for your application in the Solution Explorer.

2 Select Add Reference.

Figure 4.2   Microsoft Visual Studio .NET Add Reference dialog
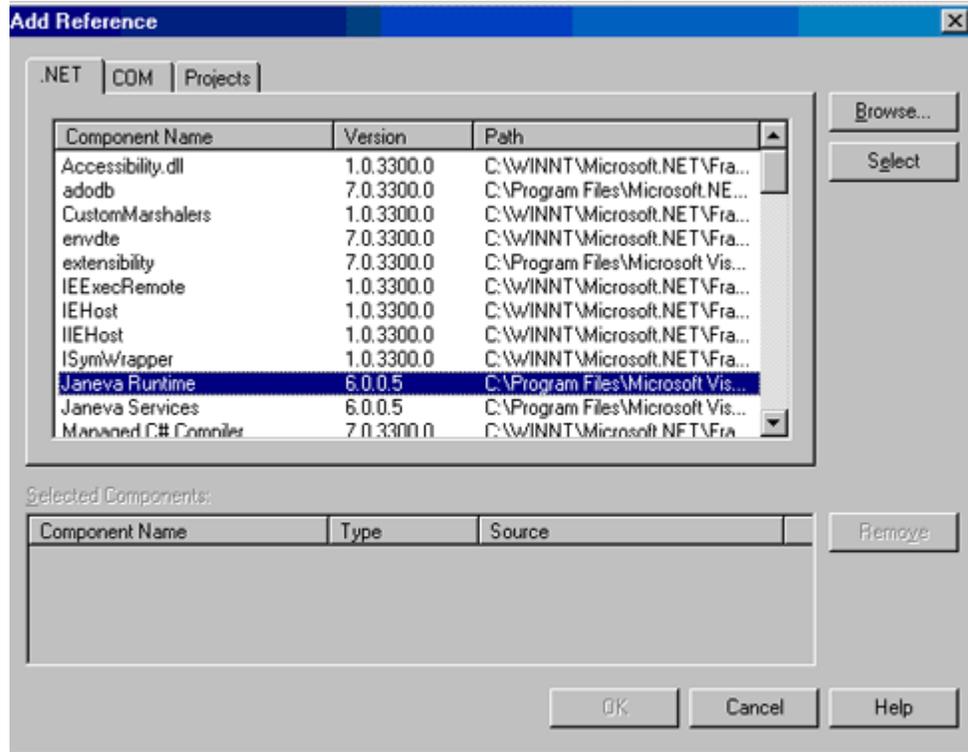
3 In the .NET tab select the appropriate VisiBroker for .NET reference and click Select.

If you are building a client application select only the VisiBroker for .NET Runtime reference. If you are building a server application, select both the VisiBroker for .NET Runtime and VisiBroker for .NET Services references.

4 Click OK.

If you selected the VisiBroker for .NET Runtime reference, the Borland.Janeva.Runtime should appear in your references list. If you selected the VisiBroker for .NET Services reference, the Borland.Janeva.Services should appear in your references list.

command line

To add the reference to the VisiBroker for .NET runtime library at compile time, invoke the C# command line compiler on the C# source code, including Borland.Janeva.Runtime.dll or Borland.Janeva.Services.dll as a reference.

```
prompt> csc /r:Borland.Janeva.Runtime.dll Client.cs
```

Deploying VisiBroker for .NET applications

To deploy applications using the VisiBroker for .NET technology you will need to include the following items:

– Microsoft .NET Framework Redistributable Package

– VisiBroker for .NET runtime libraries

– VisiBroker for .NET deployment license key

Microsoft .NET Framework Redistributable Package

VisiBroker for .NET is a .NET product. As such, it requires the .NET Framework Redistributable Package for execution, which is available as a free download from the Microsoft Web site, or it may be included with your IDE or operating system.

VisiBroker for .NET runtime libraries

For deployment, VisiBroker for .NET supports client applications on the front end or ASP.NET server applications. You must install the following VisiBroker for .NET runtime libraries on each machine that runs the VisiBroker for .NET-powered applications.

– Borland.Janeva.Runtime.dll

– Borland.Janeva.Runtime.Private.dll

– Borland.Janeva.NTD.dll

– Borland.BC.Bootstrap.dll

– Borland.BC.Rt.Core.dll

– Borland.BC.Jre.dll

The following two need to be installed only if services such as security, firewall, or transactions are being used:

– Borland.Janeva.Services.dll

– Borland.Janeva.Services.Private.dll

Depending on the application server being used, you will need to install one or more of the following:

– Borland.Janeva.[BES|Oracle|WebLogic|WebSphere].dll

You can install them in one of two ways:

– Install the VisiBroker for .NET runtime libraries on the target machine using the Borland VisiBroker CD

– Package the runtime libraries from your VisiBroker for .NET development installation in an application setup program

Clients (that make use of the VisiBroker for .NET runtime) on the same host can share the VisiBroker for .NET runtime libraries if you install them in the GAC.

VisiBroker for .NET deployment license key

The VisiBroker for .NET deployment license key is on your VisiBroker deployment CD-ROM in the License directory. You can use the license in one of three ways:

– Include the license as an embedded resource

– Copy the license to the application's virtual root (for ASP.NET deployment)

– Point to the license file location in the application configuration file

Important

Refer to your license agreement to determine what constraints exist on the number and types of machines on which you can use your deployment license.

Including the license as an embedded resource

The following procedures describe the steps to include the VisiBroker for .NET deployment license as an embedded resource in your application using Microsoft Visual Studio .NET.

To embed a resource using Visual Studio .NET:

1 Copy the license file (client.slip or server.slip) from the License directory on the VisiBroker deployment CD-ROM to your project directory.

2 Rename the SLIP file to borland.slip.

3 Click Show All Files in the Solution Explorer.

4 Right-click the license file, and select Include In Project.

5 Right-click the license file, and select Properties.

6 Change the Build Action property to Embedded Resource.

Copying the license to the application virtual root

To include the VisiBroker for .NET deployment license in the application root of an ASP.NET server application:

1 Copy the license file (client.slip or server.slip) from the License directory on the VisiBroker deployment CD-ROM to the application's root installation directory.

2 Rename the SLIP file to borland.slip.

Modifying the application configuration file

To include the location of the VisiBroker for .NET deployment license in the application configuration file:

1 Copy the license file (client.slip or server.slip) from the License directory on the VisiBroker deployment CD-ROM to a directory on your network.

2 Modify the XML to include the `<license>` element as shown in the following example.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
     <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections
   <visinet>
     <license dir="C:\Program Files\Borland\Janeva"/>
   </visinet>
</configuration>
```

The `<license>` `dir` value should be the absolute or relative path to the file containing the Borland license key.

5

Developing VisiBroker for .NET Remoting servers

This chapter explains the process for developing a VisiBroker for .NET Remoting server, and in particular it discusses how to implement a `MarshalByRefObject` object in VisiBroker for .NET.

Introduction

This section introduces the concepts of .NET Remoting server and a VisiBroker for .NET server.

About .NET Remoting

`MarshalByRefObject` objects are remote objects that run on the server and accept method calls from clients. .NET Remoting `MarshalByRefObjects` can be categorized into two groups:

– Server-activated objects (SAOs)

– Client-activated objects (CAOs)

SAOs can be marked as either Singleton or SingleCall. In the first case, one instance serves requests of all clients in a multi-threaded fashion. When using SAOs in `SingleCall` mode, a new object will be created for each request and destroyed afterwards. Both Singleton and SingleCall SAO modes are supported in VisiBroker for .NET. In addition to that, VisiBroker for .NET supports transient `MarshalByRefObject` objects that run either on a server, or on a client for server callback.

About VisiBroker for .NET Server

A VisiBroker for .NET server always starts from an IDL interface definition. An IDL interface defines the business logic that both the client and the server abide by. For example, the following example IDL file defines three interfaces:

– an `AccountManager` interface that follows the factory design pattern with an open method for opening new bank accounts.

- an `Account` interface that has operations to query the balance, as well as to do account debit and credit.

- a `Callback` interface for banking event notification.

```
// Bank.idl
module Bank {
   interface Callback {
      void notify(in string message);
   };
   interface Account {
      float balance();
      void credit(in float amount);
      void debit(in float amount);
   };
   interface AccountManager {
      Account open(in float balance, in Callback callback);
   };
};
```

A server will implement both the `AccountManager` interface and the `Account` interface. The client will provide the implementation for the `Callback` interface so that the bank server can call back to notify the client about all of the banking events.

The next two sections will walk through how to write the Bank server in .NET Remoting style, as well as how to add the callback implementation to the .NET Remoting style client.

Developing a server in .NET Remoting style

A server needs to implement the business logic. For the bank example, the bank server needs to provide implementation for both the `AccountManager` interface and the `Account` interface. The following code snippet shows the implementation of the `AccountManager` interface and the `Account` interface at the server side:

```
namespace Server {

   public class AccountImpl : MarshalByRefObject, Bank.Account {

      private float _balance;
      private Callback _callback;
      internal AccountImpl(float balance, Callback callback) {
         _balance = balance;
         _callback = callback;
         _callback.Notify("Created account with $" + _balance);
      }

      public float Balance() {
         _callback.Notify("Current balance is $" + _balance);
         return _balance;
      }

      public void Credit(float amount) {
         _callback.Notify("Crediting account with $" + amount);
         _balance += amount;
      }

      public void Debit(float amount) {
         if(amount <= _balance) {
            _callback.Notify("Debiting account by $" + amount);
            _balance -= amount;
         }
         else {
            _callback.Notify("Insufficient funds to debit $" + amount);
         }
```

```
      }
   }

   public class AccountManagerImpl : MarshalByRefObject, Bank.AccountManager {

      public AccountManagerImpl() {
         Console.WriteLine("AccountManager created on : " +
            System.DateTime.UtcNow.ToLongTimeString());
      }

      public Account Open(float balance, Callback callback) {
         Console.WriteLine("Opening a new account with balance = $" + balance);
         return new AccountImpl(balance, callback);
      }
   }
}
```

The `Open()` method of the `AccountManagerImpl` class takes in an initial balance and a `Callback` object reference that is passed in by the client, then creates a new instance of `AccountImpl` class.

The `Balance()` method of the `AccountImpl` class simply returns the balance to the client; the `Credit()` method credits the passed in amount to the account balance; the `Debit()` method debits the requested amount from the account balance. All of these three account operation events are notified to the client via the `Callback` object.

Now that the interface implementation is completed, the next step for the server is to register the `AccountManagerImpl` object either as a well known `SingleCall` service object or as a well known `Singleton` object to the .NET Remoting system. `AccountImpl` objects are transient as they do not outlive the process that created them.

Singleton object configuration

When a server implementation object is configured as a `Singleton` well known service type, only one instance of the server implementation object is created. It is this singleton instance that serves all requests coming from all clients. The configuration can be done either explicitly using .NET `RemotingConfiguration` APIs, or implicitly using a .NET Remoting configuration file.

Explicit registration

Singleton server implementation objects are explicitly registered to the Remoting system at the server side using the following statement:

```
RemotingConfiguration.RegisterWellKnownServiceType(
   typeof(<TheServerImplClass>),
      "<objectURI>", WellKnownObjectMode.Singleton);
```

For the bank example, the following code snippet explicitly registers an instance of `AccountManagerImpl` class as a well known `Singleton` service type with `AccountManager.iiop` as its end point URI:

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(
   Server.AccountManagerImpl),"AccountManager.iiop",
   WellKnownObjectMode.Singleton);
```

Implicit registration

Implicit registration of a server implementation object as a well known `Singleton` service type is done through the `<service>` property in the .NET Remoting configuration file as shown in the following example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="Singleton"
```

```
                        type="<namespace>.<implclassname>, <assembly>"
                        objectUri="<objectURI>"/>
              </service>
          </application>
      </system.runtime.remoting>
  </configuration>
```

and a call to .NET `RemotingConfiguration` to load in the configuration file:

```
RemotingConfiguration.Configure("<configfile>");
```

For the bank example, the complete configuration file of the server is shown below:

```
<configuration>
   <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections>
   <visinet>
      <agent port="24300" addr="localhost"/>
      <server defaultPort="10000">
         <remoting enabled="true"/>
      </server>
   </visinet>
   <system.runtime.remoting>
      <application name="Server">
         <channels>
            <channel type="Janeva.Remoting.IiopChannel,
               Borland.Janeva.Runtime"/>
         </channels>
         <service>
            <wellknown mode="Singleton"
                   type="Server.AccountManagerImpl, Server"
                   objectUri="AccountManager.iiop"/>
         </service>
      </application>
   </system.runtime.remoting>
   <janeva.runtime.remoting>
      <wellknown objectUri="AccountManager.iiop" jndi="a/b/c"/>
   </janeva.runtime.remoting>
</configuration>
```

For more information on `Janeva.Remoting.IiopChannel` type and its properties, see "Specifying the Remoting channel".

SingleCall object configuration

When a server object is configured as a well known `SingleCall` object, the server will create one instance per each client invocation of a method, execute the method and then destroy the object again. Similar to the `Singleton` mode, the configuration can be done either explicitly using .NET `RemotingConfiguration` APIs, or implicitly using .NET Remoting configuration file.

Explicit registration

To register a `SingleCall` server implementation object explicitly, use the following codes:

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(<TheServerImplClass>)
,
   "<objectURI>", WellKnownObjectMode.SingleCall);
```

Implicit registration

To register a `SingleCall` server implementation object implicitly, change the `<wellknown>` property's mode attribute to be `SingleCall` in the .NET Remoting configuration file:

```
<configuration>
   <configSections>
     <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
   </configSections>
   <visinet>
     <agent port="24300" addr="localhost"/>
     <server defaultPort="10000">
        <remoting enabled="true"/>
     </server>
   </visinet>
   <system.runtime.remoting>
     <application>
        <service>
           <wellknown mode="SingleCall"
                 type="<namespace>.<implclassname>, <assembly>"
                 objectUri="<objectURI>"/>
        </service>
     </application>
   </system.runtime.remoting>
</configuration>
```

If you compare the output of the bank server example between `Singleton` and `SingleCall` mode, you'll notice that in `Singleton` mode, the `AccountManagerImpl` class constructor is invoked only once no matter how many times a client tries to invoke the open method. While in `SingleCall` mode, the constructor is invoked once every time when the client invokes the open method.

Adding callbacks to a VisiBroker for .NET Remoting client

Adding callback objects to a VisiBroker for .NET remoting client is straight forward: implement the callback interface defined in the IDL file, then create an instance of the callback object and pass it as object reference to a server invocation method. Callback objects are transient objects in VisiBroker for .NET.

The following code listing shows a complete client implementation of the bank example:

```
using System;
using System.Runtime.Remoting;
using Bank;

namespace Client {

   public class CallbackImpl : MarshalByRefObject, Callback {

      public void Notify(string message) {
         Console.WriteLine("  Callback: " + message);
      }
   }

   public class Client {
      static void Main(string[] args) {
         try {
            RemotingConfiguration.Configure("Client.config");
            AccountManager manager = new AccountManagerRemotingProxy();
            Callback callback = new CallbackImpl();
            Account account = manager.Open(1000, callback);
            Console.WriteLine("Balance = $" + account.Balance());
            Console.WriteLine("Withdrawing $500");
```

```
                              account.Debit(500);
                              Console.WriteLine("balance = $" + account.Balance());
                              Console.WriteLine("Depositing $100");
                              account.Credit(100);
                              Console.WriteLine("Balance = $" + account.Balance());
                              Console.WriteLine("Withdrawing $700");
                              account.Debit(700);
                              Console.WriteLine("Balance = $" + account.Balance());
                          }
                          catch(Exception e) {
                              Console.WriteLine(e);
                          }
                          Console.WriteLine("Press enter key to stop the client...");
                          Console.ReadLine();
                      }
                  }
              }
```

The .NET remoting configuration file `Client.config` used by the bank client is listed below:

```
<configuration>
    <system.runtime.remoting>
        <application name="Client">
            <channels>
                <channel type="Janeva.Remoting.IiopChannel,
                    Borland.Janeva.Runtime"/>
            </channels>
            <client>
                <wellknown type="Bank.AccountManagerRemotingProxy, Client"
                        url="janeva:corbaloc::localhost:10000/AccountManager.iiop"/>
            </client>
        </application>
    </system.runtime.remoting>
</configuration>
```

Refer to ".NET Remoting configuration" for details on how write the Remoting section of the VisiBroker for .NET Remoting configuration file. See "Configuring properties," for information about configuring VisiBroker for .NET properties in a configuration file.

Properties

By default, VisiBroker for .NET Remoting server and callback feature is turned off. You will need to enable it explicitly for developing a VisiBroker for .NET Remoting server and/or add callback objects into your Remoting client. This is done by setting the `janeva.server.remoting` property to `true`. See "Configuring properties," for information about configuring VisiBroker for .NET properties in a configuration file.

6

Using hints and custom marshaling

This chapter explains how to use hints to control `java2cs` code generation for valuetypes in VisiBroker for .NET.

VisiBroker for .NET has a powerful mechanism that lets the user customize the code generation for Java valuetypes. Valuetypes are value classes that are implemented in Java (typically extending `java.io.Serializable` directly or indirectly). These classes have state and are intended to be marshaled over the wire as state.

VisiBroker for .NET code generation—an example

In order to fully understand the use of hints and how they affect `java2cs` code generation, the following example shows a simple Java type called User.

```
public class User implements java.io.Serializable {
   public String name;
   private String password;
   public User (String name, String password) {
      this.name = name;
      this.password = password;
   }
}
```

Obviously this example class is not realistic as it does not allow access to initialize or in any way use the private state of this object. However, we are skipping a real implementation of this object (with appropriate constructors and methods) for the sake of simplicity. For this discussion methods in Java classes are irrelevant.

Note:

We are generating a C# class corresponding to the Java class. The methods in Java classes are irrelevant because *porting* the methods would involve essentially reverse engineering the Java class, and so the porting of methods is not supported. If you would like to have the same or similar methods in your C# class corresponding to the Java version of your valuetype, you will have to implement the C# equivalent yourself. Later sections in this document will explain how that is done.

The important sections of the C# code that is generated from the example Java class, `User`, are shown below.

```
[System.Serializable] public class User {

   private string _Name;
```

```
public virtual string Name {
   get { return this._Name; }
   set { this._Name = value; }
}

private string _Password;

public virtual string Password {
   get { return this._Password; }
   set { this._Password = value; }
}

// Other common object methods omitted
}
```

The C# type `User` represents the Java class `User`. As is apparent, this is incorrect in a few ways.

– It provides public accessors to the private field (`password`, `_Password` in C#). This will happen regardless of whether the Java type provides the same accessors or not. As mentioned, the compiler will not look at the Java methods.

– This class demotes the access modifier of the field `name` (`_Name` in C#) from public to private, but a public property is provided for access.

– The C# object has no constructors or methods generated from the Java type.

In short, this class is not very usable. However, it provides you a starting point from which you can build your real valuetype. You can cut this code from the generated code, add it to your source, and add all the useful constructors and methods. We will show you later how to avoid generating this class again, and instead use your version.

ValueFactory class

Now let us look at the generated ValueFactory class for `User`. This class is responsible for creating and initializing an instance of the C# type `User` when it reads an instance of the Java class `User` from the network. it is also responsible for writing the correct data to the network when you pass an instance of the C# class `User` to a remote server. It is important to note that the ValueFactory is associated with the corresponding Java type. That is, each distinct Java type will have a distinct factory. This allows more than one Java type to map to a given C# type.

ValueFactory methods

The ValueFactory class has many methods, but the following example highlights the most significant ones that you will need to know.

```
public class UserValueFactory : CORBA.ValueFactory {

   public virtual CORBA.TypeCode GetTypeCode() {
      return UserHelper.GetTypeCode();
   }

   public virtual System.Type GetValueType() {
      return typeof(User);
   }

   public virtual User CreateObject() {
      return new User();
   }

   public virtual void InitObject(UserValueData src_data, User dst_object) {
      dst_object.Name = src_data.Name;
      dst_object.Password = src_data.Password;
   }

   public virtual void InitData(User src_object, UserValueData dst_data) {
      dst_data.Name = src_object.Name;
```

```
        dst_data.Password = src_object.Password;
    }
}
```

Note that `UserValueData` is a class that contains as public data members every instance member of the `User` class as shown in the following example.

```
public class UserValueData {
    public string Name;
    public string Password;
}
```

The following table describes the ValueFactory methods:

Method name	Description
GetValueType	Returns the type of the class that maps to the Java type MyValue.
CreateObject	Returns a new instance of the C# type corresponding to the Java type MyValue
InitObject	Used when reading a Java MyValue. The C# type created by `CreateObject` is passed to it as well as the ValueData class. When the call to `InitObject` is made, the data for MyValue has already been unmarshaled into the ValueData class. The `InitObject` merely assigns the fields from the ValueData class to the C# MyValue class. We will see the usefulness of this pattern later.
InitData	Used when writing the C# MyValue to the stream. This method merely transfers the state of the members of the C# MyValue to the ValueData class. The infrastructure will then marshal the state from the ValueData class.

Based on the above table, you see that the `ValueData` class represents the data that is marshaled on the wire, irrespective of how the data is stored or maintained in the C# type.

Notice that the ValueFactory created the object in one step (`CreateObject`) and read the data in another step (`InitObject`). There is a good reason for this. When unmarshaling or marshaling a type that is inherited from other stateful types, each type's factory is normally responsible for marshaling and unmarshaling only the state at its level in the hierarchy. To achieve this, the infrastructure will first instantiate an instance of the type that is being unmarshaled, but will pass it to the factory corresponding to each type in the hierarchy, starting from the base, to unmarshal the relevant state and work its way up the hierarchy. When writing, the same process is repeated, this time using the `InitData` methods.

An introduction to hints

The hints file is an XML file that provides hints to the `java2cs` compiler allowing the user to customize the code that is generated.

The following is an example of a simple hints.xml file.

```
<?xml version="1.0" ?>
<hints>
    <hint>
        <java-class>User</java-class>
        <cs-impl-type>UserData</cs-impl-type>
    </hint>
</hints>
```

To run the `java2cs` compiler with the above hints file, enter the following at the command line:

```
java2cs -hint_file hints.xml -o User.cs User
```

Supplying the implementation of a value type

Running the compiler with the following hint will cause the compiler to stop generating the `User` class.

```
<?xml version="1.0" ?>
<hints>
   <hint>
      <java-class>User</java-class>
      <cs-impl-type>User</cs-impl-type>
   </hint>
</hints>
```

You can now write your implementation of the `User` class as desired and compile it with the generated code.

Replacing the default implementation with a custom implementation of a different name

Running the compiler with the following hint will change the name of the C# type from `User` to `UserData`.

```
<?xml version="1.0" ?>
<hints>
   <hint>
      <java-class>User</java-class>
      <cs-impl-type>UserData</cs-impl-type>
   </hint>
</hints>
```

Using the above hint, the compiler no longer generates the `User` class or the `UserData` class. However, all of the other classes are generated with the assumption that you will implement the `UserData` class.

Notice that after generating code using the example hints file, the ValueFactory no longer refers to the `User` class. Rather, it refers to the `UserData` class.

```
public virtual System.Type GetValueType() {
   return typeof(UserData);
}

public virtual UserData CreateObject() {
   return new UserData();
}

public virtual void InitObject(UserValueData src_data,
         UserData dst_object) {
   dst_object.Name = src_data.Name;
   dst_object.Password = src_data.Password;
}

public virtual void InitData(UserData src_object,
           UserValueData dst_data) {
   dst_data.Name = src_object.Name;
   dst_data.Password = src_object.Password;
}
```

You could now write a `UserData` class (as shown in the following example) and use it with the generated code.

```
[System.Serializable] public class UserData {

   private string _Name;
   private string _Password;

   public UserData() {
   }
```

```
public UserData(string name, string password) {
   _Name = name;
   _Password = password;
}

internal void Init(string name, string password) {
   _Name = name;
   _Password = password;
}

public string Name {
   get {
      return _Name;
   }
}

public string Password {
   get{
      return _Password;
   }
}
}
```

You cannot use this class as is. In order for this class to compile, you will need to expose visible properties (or fields) to `InitObject` and `InitData` called `Name` and `Password` (See the code for `InitObject` and `InitData` in the generated `ValueFactory` class).

To fix this you can either add visible properties or change the field names to be `Name` and `Password` and make them visible to the generated code.

While this is straightforward, you may not want to expose the class fields. Rather you might want to keep your class as shown above. This means you need to take over the work of `InitObject` and `InitData` and rewrite the hints file.

```
<?xml version="1.0" ?>
<hints>
   <hint>
      <java-class>User</java-class>
      <cs-impl-type>UserData</cs-impl-type>
      <mode>custom</mode>
   </hint>
</hints>
```

The only difference between this hint file and the previous one is that the mode is set to `custom`. The generated code changes very little. In fact the only difference is in the `InitObject` and `InitData` methods. They are generated as follows:

```
public abstract class UserValueFactory : CORBA.ValueFactory {

   public abstract void InitObject(UserValueData src_data,
              UserData dst_object);

   public abstract void InitData(UserData src_object,
              UserValueData dst_data);
}
```

Notice that the class and these methods are no longer concrete. You will need to provide a factory for the `User` type now, but it is a trivial implementation:

```
public class UserFactory: UserValueFactory {

   public override void InitObject(UserValueData src_data,
              UserData dst_object) {
      dst_object.Init(src_data.Name, src_data.Password);
   }

   public override void InitData(UserData src_object,
              UserValueData dst_data) {
      dst_data.Name = src_object.Name;
```

```
                    dst_data.Password = src_object.Password;
               }
          }
```

This ValueFactory will automatically be registered as the ValueFactory for the `User` Java class as long as one of the Helper classes in the generated code is used. To explicitly register a ValueFactory you can either call `ORB.RegisterValueFactory()`, or you can call `ORB.RegisterAssembly()` which will register all of the factories in the provided assembly.

Mapping interfaces with methods

Consider the Java interface:

```
public interface Principal {
   public String getUserName();
}
```

and the Java class:

```
public class User implements Principal, java.io.Serializable {
   private String name;
   private String password;

   public User(String name, String password) {
      this.name = name;
      this.password = password;
   }

   public String getUserName() {
      return name;
   }
}
```

Running the compiler on this interface and class, without hints for both the interface and the class, will result in the following warning:

```
java2cs: (warning)The type Principal requires a mapping hint to be fully valid
(e.g., method signatures will be ignored).
```

```
java2cs: (warning)The type User requires a mapping hint to be fully valid
(e.g., method signatures will be ignored).
```

This warning indicates that the interface (which is not a remote interface) has methods that are ignored by the `java2cs` compiler. The compiler ignores these methods as it is not possible for the compiler to map methods that are not designed to be invoked remotely. This is due to the fact that the parameters that such methods take may be valid only in the local contexts. If you look at the generated code, the compiler will generate the following code for `Principal`:

```
public interface Principal {
}
```

and the following code for User:

```
[System.Serializable] public class User : Principal {
    . . .
}
```

The compiler ignored the generating the code for the `getUserName` method. The compiler warnings suggest that this is most likely not what is expected, and therefore you must use a hint to map this to an appropriate .NET interface.

Let's say that we use the following hint file (note that we are not providing a hint for `User`):

```
<hints>
   <hint>
      <java-class>Principal</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
```

```
         </hint>
      </hints>
```

This maps the interface `Principle` to the C# interface `IPrinciple` (which the compiler will not generate). Let us say we also add the `IAuthenticatable` to our .NET code as follows (note that you could use an existing interface, such as `System.Security.Principals.IPrincipal`):

```
public interface IPrincipal {
   string GetName();
}
```

Now, this works better. The generated `User` extends `IPrincipal`:

```
[System.Serializable] public class User : IPrincipal {
   ...
}
```

The compiler would have still generated the warning:

```
java2cs: (warning)The type User requires a mapping hint to be fully valid
(e.g., method signatures will be ignored).
```

Now it is obvious why this warning is generated. The `User` class that is generated cannot possibly know that the `IPrincipal` has a method called `GetName` that needs to be implemented. And even if it did, it could not possibly know how the method was implemented.

The rule here, therefore, is that whenever the compiler generates a value class, which it knows contains methods that need to be implemented, it will generate the warning.

Using signature type to hide implementation details

In the above case the `User` type implemented an interface. There are many cases where we develop classes that implement interfaces but our classes are private implementations that are never exposed to the user. For example, consider an Iterator of any collection. While the `Iterator` interface is public, all implementations of it are typically hidden and are never exposed to the user.

For example, if `User` were one such type, you do not want your ValueFactories actually exposing the type in its signatures because ValueFactories are public classes. To avoid this you can use the signature type in the hint to control what is exposed by the ValueFactory.

The following hint:

```
<hints>
   <hint>
      <java-class>Principal</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
   </hint>
   <hint>
      <java-class>User</java-class>
      <cs-sig-type>IPrincipal</cs-sig-type>
      <cs-impl-type>UserData</cs-impl-type>
      <mode>custom</mode>
   </hint>
</hints>
```

generates the following ValueFactory:

```
public abstract class UserValueFactory : CORBA.ValueFactory {

   public virtual System.Type GetValueType() {
      return typeof(UserData);
   }

   public virtual IPrincipal CreateObject() {
      return new UserData();
```

```
        }
        public abstract void InitObject(UserValueData src_data,
                        IPrincipal dst_object);
        public abstract void InitData(IPrincipal src_object,
                        UserValueData dst_data);
    }
```

Note that while the implementation that the factory uses is `UserData`, all of the signatures use `IPrincipal`.

Explicit factory code

Sometimes it is just convenient to write all the factory code yourself. To do this, use the following hints:

```
<hints>
    <hint>
        <java-class>Principal</java-class>
        <cs-sig-type>IPrincipal</cs-sig-type>
    </hint>
    <hint>
        <java-class>User</java-class>
        <cs-sig-type>UserData</cs-sig-type>
        <mode>custom</mode>
    </hint>
</hints>
```

The only changes from the previously generated code are the `GetValueType` and `CreateObject` methods which are also abstract now.

```
public abstract System.Type GetValueType();

public abstract UserData CreateObject();
```

The key here is that `cs-sig-type` element is used in the hint rather than `cs-impl-type`. This instructs the compiler to exclude all references to the implementation class.

Notice that you can still tweak the other aspects of the hints to change other code generation aspects. For example the following hint:

```
<hints>
    <hint>
        <java-class>Principal</java-class>
        <cs-sig-type>IPrincipal</cs-sig-type>
    </hint>
    <hint>
        <java-class>User</java-class>
        <cs-sig-type>UserData</cs-sig-type>
    </hint>
</hints>
```

still results in the `InitObject` and `InitData` methods being generated as shown below:

```
public virtual void InitObject(UserValueData src_data,
            UserData dst_object) {
    dst_object.Name = src_data.Name;
    dst_object.Password = src_data.Password;
}

public virtual void InitData(UserData src_object,
            UserValueData dst_data) {
    dst_data.Name = src_object.Name;
    dst_data.Password = src_object.Password;
}
```

Immutables

Consider the earlier example of the `UserData` class with one slight modification. In the following example we removed the `init` method and the default void constructor:

```
[System.Serializable] public class UserData {

    private string _Name;
    private string _Password;

    public UserData(String name, string password) {
        _Name = name;
        _Password = password;
    }

    public string Name {
        get {
            return _Name;
        }
    }

    public string Password {
        get{
            return _Password;
        }
    }
}
```

This is an example of a class that cannot be created without initializing its fields. Also notice that once created there is no way to initialize its fields. There are no methods to set the `Name` and `Password` fields, but here we are reading the state of the object from the network and we need to set the object's state to the exact values we read.

However, our ValueFactory creates the object in the `CreateObject` method and initializes it in another step (`InitObject`). This obviously will not work for us. To support this case, we provide the `immutable` mode in the hint.

Using this hint:

```
<hints>
    <hint>
        <java-class>Principal</java-class>
        <cs-sig-type>IPrincipal</cs-sig-type>
    </hint>
    <hint>
        <java-class>User</java-class>
        <cs-sig-type>IPrincipal</cs-sig-type>
        <cs-impl-type>UserData</cs-impl-type>
        <mode>immutable</mode>
    </hint>
</hints>
```

results in the following signature for `InitObject`:

```
public abstract IPrincipal InitObject(UserValueData src_data);
```

Also, the `CreateObject` call is no longer generated (abstract or otherwise).

Notice here how the `InitObject` returns an `IPrincipal` rather than receiving one as argument. This allows you to write a ValueFactory that creates a `UserData` with the value data that has already been unmarshaled and return it.

Such a ValueFactory might look like this:

```
public class UserFactory: UserValueFactory {

    public override IPrincipal InitObject(UserValueData src_data);
        return new UserData(src_data.Name, src_data.Password);
    }
```

```
        public override void InitData(UserData src_object,
                    UserValueData dst_data) {
            dst_data.Name = src_object.Name;
            dst_data.Password = src_object.Password;
        }
    }
```

Be aware that with the `immutable` mode you are responsible for using all the state in the data object (which will include all the data for all of the base classes as well) to initialize your immutable object as appropriate.

Custom marshaling

When writing passwords to the network you may want to encrypt them to prevent passwords from being sent in the clear. To do this you should have the Java class use custom marshaling.

Consider the following version of the Java `User` class:

```
public class User implements Principal, java.io.Serializable {
    private String name;
    transient private String password;

    public User(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public String getUserName() {
        return name;
    }

    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException {
        s.defaultWriteObject();
        s.writeObject(encrypt(password));
    }

    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        s.defaultReadObject();
        password = encrypt((String) s.readObject());
    }

    private String encrypt(String val) {
        char[] result = new char[val.length()];
        for (int 1 = 0; i < val.length(); i++) {
            result[i] = (char) (((byte) val.charAt(i)) ^ 0x77);
        }
        return new String(result);
    }
}
```

This is a custom marshaled Java `Serializable` class. The default code generation for this class (with no hints) shows some changes. The value class is no longer generated. This is because the compiler knows that your class is custom marshaled, so it cannot possibly generate the appropriate fields in your class. However, it does know to generate the ValueData class, as that represents the fields (the nontransient fields) that would be marshaled if the class used default marshaling. As show in the code sample above, the class also marshals some additional data.

The ValueData is generated as follows:

```
public class UserValueData {
    public string Name;
}
```

The ValueFactory is generated as follows:

```
public abstract System.Type GetValueType();

public abstract User CreateObject();

public abstract void ReadObject(UserValueData data,
            CORBA.ObjectInputStream input,
            User obj);

public abstract void WriteObject(User obj,
            UserValueData data,
            CORBA.ObjectOutputStream output);

public static void DefaultReadValueData(UserValueData data,
            CORBA.ObjectInputStream input) {
  ...
}

public static void WriteValueData(UserValueData data,
            CORBA.ObjectOutputStream output) {
  ...
}
```

Notice that the `GetValueType` and `CreateObject` methods are now abstract. The compiler requires you to provide the implementation for these as it does not know the name of your C# class. Second, note that you no longer have the `InitObject` and `InitData` methods. Instead, you have two new methods: `ReadObject` and `WriteObject`. You will have to implement these methods to provide the appropriate custom marshaling logic. As you can see, the ValueData object and the value class are still passed to the method, but in addition a Stream is also passed. This allows the custom marshaling logic to be written. And finally some additional methods (`DefaultReadValueData` and `WriteValueData`) are generated to allow the user to read or write default marshaled data.

In Java, a common use of custom marshaling is to lazy-compute serializable fields at the time of marshaling and to lazy-initialize transient fields at the time of unmarshaling. The actual marshaling remains identical. Sometimes, the custom marshaling reads and writes the default fields but adds some additional data at the end of the stream.

A sample value factory for the above Java class is shown below, using this implementation of `UserData`.

```
[System.Serializable] public class UserData {

  private string _Name;
  private string _Password;

  public UserData() {
  }

  public UserData(string name, string password) {
    _Name = name;
    _Password = password;
  }

  internal Init(string name, string password) {
    _Name = name;
    _Password = password;
  }

  public string Name {
    get {
      return _Name;
    }
  }

  public string Password {
    get{
      return _Password;
    }
```

```
        }
    }
```

The ValueFactory:

```
public class UserFactory : UserValueFactory {
    public override System.Type GetValueType() {
        return typeof(UserData);
    }
    public override UserData CreateObject() {
        return new UserData();
    }
    public string Encrypt(string val) {
        char[] ersult = new char[val.Length];
        for(int i = 0; i < val.Length; i++) {
            result[i] = (char) (((byte) val[i] ^ 0x77);
        }
        return new string(result);
    }
    public override void ReadObject(UserValueData data,
                CORBA.ObjectInputStream input,
                User obj) {
        DefaultReadValueData(data, input);
        obj.Init(data.Name, Encrypt(input.ReadString()));
    }
    public override void WriteObject(User obj,
                UserValueData data,
                CORBA.ObjectOutputStream output) {
        data.Name = obj.Name;
        DefaultWriteValueData(data, output);
        output.WriteObject(Encrypt(obj.Password));
    }
}
```

As shown earlier, you may modify the name of the value object and change the signature that is exposed using the other hint techniques. You may also write additional data after the `DefaultWriteValueData` and read the same addition after the `DefaultReadValueData`. In addition, calling `DefaultWrite`/`ReadValueData` is not required.

Hints file schema

The hints file schema is as follows:

```
<?xml version="1.0" ?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="hints">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="hint" type="hintType" minOccurs="1"
                    maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:complexType name="hintType">
        <xsd:sequence>
            <xsd:element name="java-class" type="xsd:string"/>
```

```
            <xsd:element name="cs-sig-type" type="xsd:string" minOccurs="0"/>
            <xsd:element name="cs-impl-type" type="xsd:string" minOccurs="0"/>
            <xsd:element name="mode" type="modeType" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="modeType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="automatic"/>
            <xsd:enumeration value="custom"/>
            <xsd:enumeration value="immutable"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

One-to-many marshaling precedence

VisiBroker for .NET has a set of built-in value factories, that have a predetermined precedence. When there is an ambiguity about how to marshal a particular type, the default behavior is as follows:

```
// we need a deterministic ordering for value factories,
// so that the user knows how types are marshaled by
// default. The marshaling preference is based on
// registration order, with highest priority
// going to the last factory registered...

CORBA.ValueFactory[] factories = {

    // Lowest priority goes to JDK 1.4 types, since these
    // are meaningless to older JDKs...
    new J2EE.Factories.LinkedHashMapValueFactory(),
    new J2EE.Factories.LinkedHashSetValueFactory(),

    // Next in priority are the JDK 1.0 and 1.1 types,
    // which are no longer in fashion...
    new J2EE.Factories.HashtableValueFactory(),
    new J2EE.Factories.PropertiesValueFactory(),
    new J2EE.Factories.StackValueFactory(),
    new J2EE.Factories.VectorValueFactory(),

    // Finally, we have the JDK 1.2 types (note that there
    // are no relevant JDK 1.3 types)...
    // First we have the "less popular" types...
    new J2EE.Factories.LinkedListValueFactory(),
    new J2EE.Factories.TreeMapValueFactory(),
    new J2EE.Factories.TreeSetValueFactory(),

    // And finally we have the "most popular" types...
    new J2EE.Factories.HashMapValueFactory(),
    new J2EE.Factories.HashSetValueFactory(),

    // And finally ArrayList wins to overall popularity contest!
    new J2EE.Factories.ArrayListValueFactory(),
};

foreach(CORBA.ValueFactory factory in factories) {
    orb.RegisterValueFactory(factory);
}
```

Items lower in the array take precedence over items higher in the array. Of course, that may not be what you requires. In cases where you require a different precedence, you need to manually override the default behavior. The simplest way to do this is explicitly register your preferred ValueFactories in your main program. If you want

java.util.Hashtable to take precedence over competing types (such as java.util.HashMap), then your main program would contain:

```
CORBA.ORB orb = CORBA.ORB.Init();
orb.RegisterValueFactory(J2EE.Util.HashtableValueFactory.GetSingleton());
```

The ORB.Init is setting up all the default ORB behavior, including doing the ValueFactory registration shown above. This default has the HashMap ValueFactory taking precedence over the Hashtable ValueFactory. But then after initializing the ORB, we explicitly register the Hashtable ValueFactory, which will cause this to take precedence over all the previous ValueFactory registrations.

7

Using Quality of Service

Quality of Service (QoS) utilizes policies to define and manage the connection between your client applications and the servers to which they connect.

Understanding Quality of Service

Quality of Service policy management is performed through operations accessible in the following contexts:

– The ORB level policies are handled by a locality constrained `PolicyManager`, through which you can set Policies and view the current `Policy` overrides. Policies set at the ORB level override system defaults.

– Thread level policies are set through `PolicyCurrent`, which contains operations for viewing and setting `Policy` overrides at the thread level. Policies set at the thread level override system defaults and values set at the ORB level.

– Object level policies can be applied by accessing the base Object interface's quality of service operations. Policies applied at the Object level override system defaults and values set in at the ORB or thread level.

Setting policies per CORBA object

Use the CORBA.ObjectOperations methods in order to set QoS policies per CORBA object. To set QoS policies per CORBA object, one needs to cast the CORBA object to CORBA.ObjectOperations and call the method `SetPolicyOverrides_()` as shown in the following example.

```
// Set exclusive connection policy
bool deferBind = true;
Any policyValue = orb.CreateAny();
policyValue.InsertBoolean(deferBind);
Policy policies =
    orb.CreatePolicy(EXCLUSIVE_CONNECTION_POLICY_TYPE.Value, policyValue);
```

```
Calc.VisiCalc calc = Calc.VisiCalcHelper.Narrow(
   ((CORBA.ObjectOperations)objRef).SetPolicyOverrides_(
      new Policy [] {orb.CreatePolicy(
         QoSExt.EXCLUSIVE_CONNECTION_POLICY_TYPE.Value, policyValue)},
         SetOverrideType.SET_OVERRIDE));
```

Policy overrides and effective policies

The effective policy is the policy that would be applied to a request after all applicable policy overrides have been applied. The effective policy is determined by comparing the Policy as specified by the IOR with the effective override. The effective Policy is the intersection of the values allowed by the effective override and the IOR-specified Policy. If the intersection is empty a CORBA.INV_POLICY exception is raised.

QoS interfaces

The following interfaces are used to get and set QoS policies.

Object

VisiBroker for .NET extends CORBA.Object to provide additional QoS support as defined in the OMG Messaging specification. This means that there are two exposed Object interfaces.

Object methods

The CORBA.Object **interface** contains the following methods used to get the effective policy and get or set the policy override.

GetClientPolicy_

```
CORBA.Policy GetClientPolicy_(int type)
```

Returns the effective overriding Policy for the object reference. The effective override is obtained by first checking for an override of the given PolicyType at the Object scope, then at the Current scope, and finally at the ORB scope. If no override is present for the requested PolicyType, the system-dependent default value for that PolicyType is used. Portable applications are expected to set the desired defaults at the ORB scope since default Policy values are not specified.

The effective Policy is the one that would be used if a request were made. This Policy is determined first by obtaining the effective override for the PolicyType as returned by GetClientPolicy_.

The effective override is then compared with the Policy as specified in the IOR. The effective Policy is the intersection of the values allowed by the effective override and the IOR-specified Policy. If the intersection is empty, the system exception INV_POLICY is raised. Otherwise, a Policy with a value legally within the intersection is returned as the effective Policy. The absence of a Policy value in the IOR implies that any legal value may be used. Invoking NonExistent_ or ValidateConnection_ on an object reference prior to GetPolicy_ ensures the accuracy of the returned effective Policy.

If GetPolicy_ is invoked prior to the object reference being bound, the returned effective Policy is implementation dependent. In that situation, a compliant implementation may do any of the following: raise the exception CORBA.BAD_INV_ORDER, return some value for that PolicyType which may be subject to change once a binding is performed, or attempt a binding and then return the effective Policy.

Note that if the RebindPolicy has a value of TRANSPARENT, VB_TRANSPARENT, or VB_NOTIFY_REBIND, the effective Policy may change from invocation to invocation due to transparent rebinding.

Parameter	Description
type	The type of policy requested

GetPolicy_

```
CORBA.Policy GetPolicy_(int policy_type)
```

Returns the effective policy for an object reference—a Policy object of the type specified by the policy_type parameter.

Parameter	Description
policy_type	The type of policy to obtain

GetPolicyOverrides_

```
CORBA.Policy[] GetPolicyOverrides_(int[] types)
```

Returns the list of Policy overrides (of the specified policy types) set at the Object scope. If the specified sequence is empty, all Policy overrides at this scope will be returned. If none of the requested PolicyTypes are overridden at the Object scope, an empty sequence is returned.

Parameter	Description
types	The policy types queried for

SetPolicyOverrides_

```
CORBA.Object SetPolicyOverrides_(CORBA.Policy[] policies, CORBA.SetOverrideType set_add)
```

Returns a new Object with the given policies either replacing any existing policies in this Object or with the given policies added to the existing ones, depending on the value of the given SetOverrideType object.

This method works in a way similar to the CORBA.PolicyManager method of the same name. However, it updates the current set of policies of an Object, thread, or ORB with the requested list of Policy overrides. In addition, this method returns a CORBA.Object whereas other methods of the same name return void.

Parameter	Description
policies	an array of Policy objects containing the policies to be added or to be used as replacements
set_add	either SetOverrideType.SET_OVERRIDE, indicating that the given policies will replace any existing ones, or SetOverrideType.ADD_OVERRIDE, indicating that the given policies should be added to any existing ones

ValidateConnection_

```
bool ValidateConnection_(out CORBA.Policy[] inconsistent_policies)
```

Returns a boolean value based on whether the current effective policies for the object will allow an invocation to be made. It returns the value TRUE if the current effective policies for the Object allow an invocation to be made. If the object reference is not yet bound, a binding occurs as part of this operation. If the object reference is already bound, but current policy overrides have changed or for any other reason the binding is no longer valid, a rebind is attempted regardless of the setting of any RebindPolicy override.

The ValidateConnection_ operation is the only way to force such a rebind when implicit rebinds are disallowed by the current effective RebindPolicy. The attempt to bind or rebind may involve processing GIOP LocateRequests by the ORB. Returns the value

FALSE if the current effective policies would cause an invocation to raise the system exception INV_POLICY.

If the current effective policies are incompatible, the out parameter inconsistent_policies contains those policies causing the incompatibility. This returned list of policies is not guaranteed to be exhaustive. If the binding fails due to some reason unrelated to policy overrides, the appropriate system exception is raised.

Parameter	Description
inconsistent_policies	out parameter that returns the list of inconsistent policies that prevent the invocation from being made

PolicyManager

The CORBA.PolicyManager interface provides methods for getting and setting Policy overrides at the ORB level.

PolicyManager methods

GetPolicyOverrides

    CORBA.Policy[] GetPolicyOverrides(int[] ts)

This method returns a PolicyList sequence of all the overridden policies for the requested PolicyTypes. If the specified sequence is empty (that is, if the length of the list is zero), all Policy overrides at the current context level will be returned. If none of the requested PolicyTypes are overridden at the target PolicyManager, an empty sequence is returned.

SetPolicyOverrides

    void SetPolicyOverrides(CORBA.Policy[] policies, CORBA.SetOverrideType set_add)

This method modifies the current set of policy overrides with the requested list of Policy overrides. Invoking SetPolicyOverrides with an empty sequence of policies and a mode of SET_OVERRIDE removes all overrides from a PolicyManager.

The first input parameter, policies, is a sequence of references to Policy objects. The second parameter, set_add, of type CORBA.SetOverrideType indicates whether these policies should be added onto any other overrides that already exist in the PolicyManager using ADD_OVERRIDE, or they should be added to a PolicyManager that doesn't contain any overrides using SET_OVERRIDES.

Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Should you attempt to override policies that do not apply to your client, a CORBA.NO_PERMISSION exception will be raised. If the request would cause the specified PolicyManager to be in an inconsistent state, no policies are changed or added, and an InvalidPolicies exception is raised. There is no evaluation of compatibility with policies set within other PolicyManagers.

Parameter	Description
policies	A sequence of references to Policy objects.
set_add	A parameter of type CORBA.SetOverrideType that indicates whether these policies should be added (ADD_OVERRIDE) to any other overrides that already exist in the PolicyManager, or added to a clean PolicyManager free of any other overrides (SET_OVERRIDE). If the request would cause the specified PolicyManager to be in an inconsistent state, no policies are changed or added, and an InvalidPolicies exception is raised.

PolicyCurrent

The CORBA.PolicyCurrent interface derives from PolicyManager and Current without adding new methods. Therefore all operations on the PolicyManager interface are also available in PolicyCurrent. See "PolicyManager" for a description of these methods.

PolicyCurrent provides access to the policies overridden at the thread level. A reference to a thread's PolicyCurrent is obtained by invoking ResolveInitialReferences and specifying an identifier of PolicyCurrent.

DeferBindPolicy

The QoSExt.DeferBindPolicy determines if the ORB will attempt to contact the remote object when it is first created, or to delay this contact until the first invocation is made. By default, the ORB connects to the (remote) object when on a Bind or a StringToObject call.

The valid values for DeferBindPolicy are TRUE and FALSE. If DeferBindPolicy is set to TRUE, all binds will be deferred until the first invocation of a binding instance. The default value is FALSE.

If you create a client object, and DeferBindPolicy is set to true, you may delay the server startup until the first invocation. This option existed before as an option to the Bind method on the generated helper classes.

DeferBindPolicy properties

Value

```
bool Value
```

Returns the current setting of the DeferBindPolicy.

Example

The code sample below illustrates an example for creating a DeferBindPolicy and setting the policy on the ORB.

```
public class DeferBindClient {
   static void Main(string[] args) {
      try {
         CORBA.ORB orb = CORBA.ORB.Init(args);

         // Initialize the flag and the references
         bool deferMode = true;
         Any policyValue = orb.CreateAny();
         policyValue.InsertBoolean(deferMode);

         Policy policies =
            orb.CreatePolicy(DEFER_BIND_POLICY_TYPE.Value, policyValue);

         // Get a reference to the thread manager
         PolicyManager orbManager =
            PolicyManagerHelper.Narrow(
               orb.ResolveInitialReferences("ORBPolicyManager"));

         // Set the policy on the ORB level
         orbManager.SetPolicyOverrides(new Policy[] {policies},
            SetOverrideType.SET_OVERRIDE);

         // Get the binding method
         byte[] managerId = orb.StringToObjectId("BankManager");

         Bank.AccountManager manager =
            Bank.AccountManagerHelper.Bind("/qos_poa", managerId);

         // use Jack B. Quick as the account name.
         string name = "Jack B. Quick";

         // Request the account manager to open a named account.
         Bank.Account account = manager.Open(name);

         // Get the balance of the account.
         float balance = account.Balance();
```

```
                    // Print out the balance.
                    Console.WriteLine(
                        "\n The balance in " + name + "'s account is $" + balance);
                }
                catch (Exception e) {
                    Console.WriteLine(e);
                }
            }
        }
```

ExclusiveConnectionPolicy

The `QoSExt.ExclusiveConnectionPolicy` is a VisiBroker for .NET-specific policy that gives you the ability to establish an exclusive (non-shared) connection to the specified server object. This policy can have a boolean value of `TRUE` or `FALSE`. If the policy is `TRUE`, connections to the server object are exclusive. If the policy is `FALSE`, existing connections are reused if possible, and a new connection is opened only if reuse is not possible. The default value is `FALSE`.

ExclusiveConnectionPolicy properties

Value

```
    bool Value
```

Returns the current setting of the `ExclusiveConnectionPolicy`.

RelativeConnectionTimeoutPolicy

The `QoSExt.RelativeConnectionTimeoutPolicy` indicates a timeout after which attempts to connect to an object using one of the available endpoints is aborted. The timeout situation is likely to happen with objects protected by firewalls, where HTTP tunneling is the only way to connect to the object.

Note

This Policy is not enforced for in-process communications.

The policy value of type `unsigned long long` specifies the timeout in 100s of nanoseconds. It is applied to every endpoint that the ORB tries to connect to. Therefore, if multiple connection attempts are made, the elapsed time will be a multiple of the configured timeout. **The accuracy is also limited by the Java virtual machine implementation.**

RelativeConnectionTimeoutPolicy methods

RelativeExpiry

```
    long RelativeExpiry()
```

Gets the timeout in multiples of 100 nanoseconds.

Example

The following code examples illustrates how to create `RelativeConnectionTimeoutPolicy`.

```
    public class ConnClient {
        static void Main(string [] args) {
            try {
                // Initialize the ORB.
                ORB orb = ORB.Init(args);
```

```
// Get the manager Id
byte[] managerId = orb.StringToObjectId("BankManager");

string name = "Jack B. Quick";

// Specify the timeout in 100s of Nanosecs.
// To set a timeout of 20 secs, set 20 * 10^7 nanosecs
int connTimeout = 20;
Any ctopolicyValue =  orb.CreateAny();

ctopolicyValue.InsertUlonglong(connTimeout * 10000000);

Policy ctoPolicy = orb.CreatePolicy(
   RELATIVE_CONN_TIMEOUT_POLICY_TYPE.Value, ctopolicyValue);

PolicyManager orbManager = PolicyManagerHelper.Narrow(
   orb.ResolveInitialReferences("ORBPolicyManager"));

orbManager.SetPolicyOverrides(new Policy [] {ctoPolicy},
   SetOverrideType.SET_OVERRIDE);

// Locate an account manager. Give the full POA name and
// the servant ID.
AccountManager source =
   AccountManagerHelper.Bind("/qos_poa", managerId);

Account account = source.Open(name);
float balance = account.Balance();
Console.WriteLine("The balance in {0}'s account is {1}$", name,
balance);
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
   }
}
```

RebindPolicy

The `Messaging.RebindPolicy` determines how the client-side ORB handles closed connections, GIOP location-forward messages and object failures. The ORB handles fail-overs, rebinds, and reconnections by looking at the effective policy at the `CORBA.Object` instance.

The OMG-defined Policy values determine whether the ORB may transparently rebind once it is successfully bound to a target server. The extended policy values determine whether the ORB may transparently failover once it is successfully bound to a target Object.

The `RebindPolicy` is a client-side-only policy.

Note

The `RebindPolicy` is enforced only after being successfully bound to an object. For GIOP-based protocols an object reference is considered bound once it is in a state where a `LocateRequest` message would result in a `LocateReply` message with status `OBJECT_HERE`.

The `RebindPolicy` is set only on the client side. It can have one of six values that determines the behavior in the case of a disconnection, an object forwarding request, or an object failure. The `RebindPolicy` accepts the following constants to define the behavior of the client when rebinding.

The currently supported values are:

– `Messaging.TRANSPARENT`—allows the ORB to silently handle object-forwarding and necessary reconnections during the course of making a remote request.

- – `Messaging.NO_REBIND`—allows the ORB to silently handle reopening of closed connections while making a remote request, but prevents any transparent object-forwarding that would cause a change in client-visible effective QoS policies. When `RebindMode` is set to `NO_REBIND`, only explicit rebind is allowed.

- – `Messaging.NO_RECONNECT`—prevents the ORB from silently handling object-forwards or the reopening of closed connections. You must explicitly rebind and reconnect when `RebindMode` is set to `NO_RECONNECT`.

- – `QoSExt.VB_TRANSPARENT`—is the default policy. It extends the functionality of `TRANSPARENT` by allowing transparent rebinding with both implicit and explicit binding.

- – `QoSExt.VB_NOTIFY_REBIND`—throws an exception if a rebind is necessary. The client catches this exception, and binds on the second invocation.

- – `QoSExt.VB_NO_REBIND`—does not enable failover. It only allows the client ORB to reopen a closed connection to the same server; it does not allow object forwarding of any kind.

Note

Be aware that if the effective policy for your client is `VB_TRANSPARENT` and your client is working with servers that hold state data, `VB_TRANSPARENT` could connect the client to a new server without the client being aware of the change of server, any state data held by the original server will be lost.

The following table lists the behavior of the different `RebindMode` types.

RebindMode type	Reestablish closed connection to the same object?	Allow object forwarding?	Object failover?
`NO_RECONNECT`	No, throws `REBIND` exception.	No, throws `REBIND` exception.	No
`NO_REBIND`	Yes	Yes, if policies match. No, throws `REBIND` exception.	No
`TRANSPARENT`	Yes	Yes	No
`VB_NOTIFY_REBIND`	Yes	Yes	Yes. `VB_NOTIFY_REBIND` throws an exception after failure detection, and then tries a failover on subsequent requests.
`VB_TRANSPARENT`	Yes	Yes	Yes, transparently.

[1]The appropriate CORBA exception will be thrown in the case of a communication problem or an object failure.

Example

The following example creates a `RebindPolicy` of type `TRANSPARENT` and sets the policy on the ORB, thread, and object levels.

```
using System;
using System.IO;
using CORBA;
using QoSExt;
using Messaging;
using Bank;

public class TransparentClient {
    static void Main(string[] args) {
        try {
            short rebindMode = Messaging.TRANSPARENT.Value;

            // initialize the ORB
```

```
CORBA.ORB orb = CORBA.ORB.Init(args);

// get the object Id
byte[] managerId = orb.StringToObjectId("BankManager");

// locate an account manager; give the full POA name and the object Id
Bank.AccountManager manager =
   Bank.AccountManagerHelper.Bind("/qos_poa", managerId);
string s = orb.ObjectToString(manager);
CORBA.Object obj = orb.StringToObject(s);

// Create the client side policy so that we can receive TRANSIENT
// exception thrown by the server side orb.
Any policyValue = orb.CreateAny();
RebindModeHelper.Insert(policyValue, rebindMode);

Policy myRebindPolicy =
   orb.CreatePolicy(REBIND_POLICY_TYPE.Value, policyValue);

// Set the policy on the AccountManager object.
Bank.AccountManager manager = Bank.AccountManager.Narrow(
   ((CORBA.ObjectOperations)obj.SetPolicyOverrides_(
      new Policy [] {orb.CreatePolicy(
         QoSExt.EXCLUSIVE_CONNECTION_POLICY_TYPE.Value, policyValue)},
         SetOverrideType.SET_OVERRIDE));

//get a reference to the ORB policy manager
PolicyManager orbManager = null;

try {

   orbManager =
      PolicyManagerHelper.Narrow(orb.ResolveInitialReferences(
         "ORBPolicyManager"));
}
catch (CORBA.ORBNS.InvalidName e) {
}

//get a reference to the per-thread manager
CORBA.PolicyManager current = null;
try {
   current =
      PolicyManagerHelper.Narrow(orb.ResolveInitialReferences(
         "PolicyCurrent"));
}
catch (CORBA.ORBNS.InvalidName e) {
}

//set the policy on the orb level
try {
   orbManager.SetPolicyOverrides(new Policy[] {myRebindPolicy},
      SetOverrideType.SET_OVERRIDE);
}
catch (CORBA.InvalidPolicies e) {
}

// set the policy on the Thread level
try {
   current.SetPolicyOverrides(new Policy[] {myRebindPolicy},
      SetOverrideType.SET_OVERRIDE);
}
catch (CORBA.InvalidPolicies e) {
}

CORBA.Object oldObjectReference =
   Bank.AccountManagerHelper.Bind("/qos_poa", managerId);
```

```
                  CORBA.Object newObjectReference =
                    ((CORBA.ObjectOperations)oldObjectReference).SetPolicyOverrides_(
                      new Policy [] {myRebindPolicy}, SetOverrideType.SET_OVERRIDE);
            }
         catch (Exception e) {
            Console.WriteLine(e);
         }
      }
   }
```

RebindForwardPolicy

The `QoSExt.RebindForwardPolicy` determines whether the client ORB attempts to rebind in the case of a failure to connect during a `LOCATION_FORWARD`. When the client is forwarded to a new object, an attempt is made to connect to a new destination object. If this attempt fails, the ORB transparently connects back to the original object (the source of the forward), under the following circumstances:

– The total number of forwards at this point have not exceeded the value for `forward_count` specified in this policy.

– This is not the second consecutive attempt to connect to the same destination object.

The `vbroker.orb.rebindForward` property sets the value for `forward_count` at the ORB level. You can override the value for `forward_count` at the ORB, thread or object level programmatically, as in any QoS policy. The default value of `0` (zero) for the property indicates that no limit has been specified.

RebindForwardPolicy methods

ForwardCount

```
   short ForwardCount()
```

Returns the current setting for `forward_count` of the `RebindForward` policy.

RelativeRequestTimeoutPolicy

The `Messaging.RelativeRequestTimeoutPolicy` indicates the relative amount of time which a Request or its responding Reply may be delivered. After this amount of time, the Request is canceled. This policy applies to both synchronous and asynchronous invocations. Assuming the request completes within the specified timeout, the Reply will never be discarded due to timeout. Timeout value is specified in 100s of nanoseconds.

Example

The following code illustrates how to create `RelativeRequestTimeoutPolicy`.

```
   public class RequestTimeoutClient {
      static void Main(string[] args) {
         try {

            CORBA.ORB orb = CORBA.ORB.Init(args);

            // get the object Id
            byte[] managerId = orb.StringToObjectId("BankManager");

            // locate an account manager; give the full POA name and the object Id
            Bank.AccountManager manager =
               Bank.AccountManagerHelper.Bind("/qos_poa", managerId);

            string s = orb.ObjectToString(manager);

            // Specify the timeout in 100s of Nanosecs.
```

```
                 // To set a timeout of 50 secs, set 50 * 10^7 nanosecs
                 int reqTimeout = 20;
                 CORBA.Any policyValue =  orb.CreateAny();
                 policyValue.InsertUlonglong(reqTimeout * 10000000);

                 //set the RelativeRequestTimeoutPolicy
                 CORBA.Policy reqPolicy = orb.CreatePolicy(
                    RELATIVE_REQ_TIMEOUT_POLICY_TYPE.Value, policyValue);

                 // Get a reference to the thread manager
                 PolicyManager orbManager = PolicyManagerHelper.Narrow(
                    orb.ResolveInitialReferences("ORBPolicyManager"));

                 //Set the policy on the ORB level
                 orbManager.SetPolicyOverrides(new Policy[] {reqPolicy},
                    SetOverrideType.SET_OVERRIDE);
            }
         catch (Exception e) {
            Console.WriteLine(e);
         }
      }
   }
```

RelativeRoundTripTimeoutPolicy

The `Messaging.RelativeRoundtripTimeoutPolicy` specifies the relative amount of time for which a Request or its corresponding Reply may be delivered. If a response has not yet been delivered after this amount of time, the Request is canceled. Also, if a Request had already been delivered and a Reply is returned from the target, the Reply is discarded after this amount of time. This policy applies to both synchronous and asynchronous invocations. Assuming the request completes within the specified timeout, the Reply will never be discarded due to timeout. Timeout value is specified in 100s of nanoseconds.

Example

The following code illustrates how to create `RelativeRoundTripTimeoutPolicy`.

```
public class RoundtripTimeoutClient {
   static void Main(string[] args) {
      try {

         CORBA.ORB orb = CORBA.ORB.Init(args);

         // get the object Id
         byte[] managerId = orb.StringToObjectId("BankManager");

         // locate an account manager; give the full POA name and the object Id
         Bank.AccountManager manager =
            Bank.AccountManagerHelper.Bind("/qos_poa", managerId);

         string s = orb.ObjectToString(manager);

         // Specify the timeout in 100s of Nanosecs.
         // To set a timeout of 20 secs, set 20 * 10^7 nanosecs
         int rttTimeout = 50;
         Any policyValue = orb.CreateAny();
         policyValue.InsertUlonglong(rttTimeout * 10000000);

         // Create Policy
         CORBA.Policy rttPolicy =
            orb.CreatePolicy(RELATIVE_RT_TIMEOUT_POLICY_TYPE.Value,
               policyValue);

         // Get a reference to the thread manager
         PolicyManager orbManager =
```

```
            PolicyManagerHelper.Narrow(
                orb.ResolveInitialReferences("ORBPolicyManager"));

        // Set the policy on the ORB level
        orbManager.SetPolicyOverrides(new Policy[] {rttPolicy},
            SetOverrideType.SET_OVERRIDE);
    }
    catch (Exception e) {
        Console.WriteLine(e);
    }
  }
}
```

SyncScopePolicy

The `Messaging.SyncScopePolicy` defines the level of synchronization for a request with respect to the target. This interface is a local object derived from `CORBA.Policy`.

Values of type `SyncScope` are used in conjunction with a `SyncScopePolicy` to control the behavior of one-way operations. It is applied to one-way operations to indicate the synchronization scope with respect to the target of that operation request. It is ignored when any non-one-way operation is invoked.

This policy is also applied when the DII is used with a flag of `INV_NO_RESPONSE` since the implementation of the DII is not required to consult an interface definition to determine if an operation is declared one way.

The default `SyncScopePolicy` is `SYNC_WITH_TRANSPORT`.

Applications must explicitly set an ORB-level `SyncScopePolicy` to ensure portability across ORB implementations. When instances of `SyncScopePolicy` are created, a value of type `Messaging.SyncScope` is passed to `CORBA.ORB.CreatePolicy`. This policy is only applicable as a client-side override.

The following table lists the behavior of the different `SyncScope` values:

SyncScope type	Description
SYNC_WITH_TRANSPORT	Default. The ORB returns control to the client only after the transport has accepted the request message. There is no guarantee that the request will be delivered, but provides a useful degree of assurance given knowledge of the characteristics of the transport. Since no reply is returned from the server, no location-forwarding can be done with this level of synchronization.
SYNC_NONE	The ORB returns control to the client (e.g. returns from the method invocation) before passing the request message to the transport protocol. The client is guaranteed not to block. Since no reply is returned from the server, no location-forwarding can be done with this level of synchronization.

SyncScope type	Description
SYNC_WITH_SERVER	The server-side ORB is to send a reply before invoking the target implementation. If a reply of NO_EXCEPTION is sent, any necessary location-forwarding has already occurred. Upon receipt of this reply, the client-side ORB shall return control to the client application. The client blocks until all location-forwarding has been completed. For a server using a POA, the reply would be sent after invoking any ServantManager, but before delivering the request to the target Servant.
SYNC_WITH_TARGET	Equivalent to a synchronous, non-one-way operation in CORBA 2.2. The server-side ORB will only send the reply message after the target has completed the invoked operation. Note that any LOCATION_FORWARD reply will already have been sent prior to invoking the target and that a SYSTEM_EXCEPTION reply may be sent at anytime (depending on the semantics of the exception). Even though it was declared one way, the operation actually behaves like a synchronous operation. This form of synchronization guarantees that the client knows that the target has seen and acted upon a request. As with CORBA 2.2, only with this highest level of synchronization can the OTS be used. Any operations invoked with lesser synchronization precludes the target from participating in the client's current transaction.

QoS exceptions

– CORBA.INV_POLICY is raised when there is an incompatibility between Policy overrides.

– CORBA.REBIND is raised when the RebindPolicy has a value of NO_REBIND, NO_RECONNECT, or VB_NOTIFY_REBIND and an invocation on a bound object references results in an object-forward or location-forward message.

– CORBA.PolicyError is raised when the requested Policy is not supported.

– CORBA.InvalidPolicies can be raised when an operation is passed a PolicyList sequence. The exception body contains the policies from the sequence that are not valid, either because the policies are already overridden within the current scope, or are not valid in conjunction with other requested policies.

8

Using the dynamically managed types

The `DynAny` interface provides a way to dynamically create basic and constructed data types at runtime. It also allows information to be interpreted and extracted from an `Any` object, even if the type it contains was not known to the server at compile-time. The use of the `DynAny` interface enables you to build powerful client and server applications that create and interpret data types at runtime.

DynAny types

A `DynAny` object has an associated value that may either be a basic data type (such as `bool`, `int`, or `float`) or a constructed data type. The `DynAny` interface provides methods for determining the type of the contained data as well as for setting and extracting the value of primitive data types.

Constructed data types are represented by the following interfaces, which are all derived from `DynAny`. Each of these interfaces provides its own set of methods that are appropriate for setting and extracting the values it contains.

Interface	TypeCode	Description
DynArray	_tk_array	An array of values with the same data type that has a fixed number of elements.
DynEnum	_tk_enum	A single enumeration value.
DynFixed	_tk_fixed	Not supported.
DynSequence	_tk_sequence	A sequence of values with the same data type. The number of elements may be increased or decreased.
DynStruct	_tk_struct	A structure.
DynUnion	_tk_union	A union.
DynValue	_tk_value	Not supported.

Usage restrictions

A `DynAny` object may only be used locally by the process which created it. Any attempt to use a `DynAny` object as a parameter on an operation request for a bound object or to

externalize it using the `ObjectToString` method will cause a `MARSHAL` exception to be raised.

Furthermore, any attempt to use a `DynAny` object as a parameter on DII request will cause a `NO_IMPLEMENT` exception to be raised.

Creating a DynAny

A `DynAny` object is created by invoking an operation on a `DynAnyFactory` object. First obtain a reference to the `DynAnyFactory` object, and then use that object to create the new `DynAny` object.

Initializing and accessing the value in a DynAny

The `DynAny.Insert<Type>` methods in VisiBroker for .NET allow you to initialize a `DynAny` object with a variety of basic data types, where `<Type>` is `bool`, `octet`, `char`, and so on. Any attempt to insert a type that does not match the `TypeCode` defined for the `DynAny` will cause an `TypeMismatch` exception to be raised.

The `DynAny.Get<Type>` methods in VisiBroker for .NET allow you to access the value contained in a `DynAny` object, where `<Type>` is `bool`, `octet`, `char`, and so on. Any attempt to access a value from a `DynAny` component which does not match the `TypeCode` defined for the `DynAny` will cause a `TypeMismatch` exception to be raised.

The `DynAny` interface also provide methods for copying, assigning, and converting to or from an `Any` object.

Constructed data types

The following types are derived from the `DynAny` interface and are used to represent constructed data types.

Traversing the components in a constructed data type

Several of the interfaces that are derived from `DynAny` actually contain multiple components. The `DynAny` interface provides methods that allow you to iterate through these components. The `DynAny`-derived objects that contain multiple components maintain a pointer to the current component.

DynAny method	Description
Rewind	Resets the current component pointer to the first component. Has no effect if the object contains only one component.
Next	Advances the pointer to the next component. If there are no more components or if the object contains only one component, `false` is returned.
CurrentComponent	Returns a `DynAny` object, which may be narrowed to the appropriate type, based on the component's TypeCode.
Seek	Sets the current component pointer to the component with the specified, zero-based index. Returns `false` if there is no component at the specified index. Sets the current component pointer to –1 (no component) if specified with a negative index.

DynEnum

This interface represents a single enumeration constant. Methods are provided for setting and obtaining the value as a string or as an integral value.

DynStruct

This interface represents a dynamically constructed `struct` type. The members of the structure can be retrieved or set using a sequence of `NameValuePair` objects. Each `NameValuePair` object contains the member's name and an `Any` containing the member's type and value.

You may use the `Rewind`, `Next`, `CurrentComponent`, and `Seek` methods to traverse the members in the structure. Methods are provided for setting and obtaining the structure's members.

DynUnion

This interface represents a `union` and contains two components. The first component represents the discriminator and the second represents the member value.

You may use the `Rewind`, `Next`, `CurrentComponent`, and `Seek` methods to traverse the components. Methods are provided for setting and obtaining the union's discriminator and member value.

DynSequence and DynArray

A `DynSequence` or `DynArray` represents a sequence of basic or constructed data types without the need of generating a separate `DynAny` object for each component in the sequence or array. The number of components in a `DynSequence` may be changed, while the number of components in a `DynArray` is fixed.

You may use the `Rewind`, `Next`, `CurrentComponent`, and `Seek` methods to traverse the members in a `DynArray` or `DynSequence`.

9

Using Portable Interceptors

This chapter provides an overview of Portable Interceptors. Portable Interceptor example code is available in your VisiBroker for .NET installation.

Portable Interceptors overview

VisiBroker for .NET provides a set of interfaces known as *interceptors* which provide a framework for plugging in additional ORB behavior such as security, transactions, or logging. These interceptor interfaces are based on a *callback* mechanism. For example, using the interceptors, you can be notified of communications between clients and servers, and modify these communications if you wish, effectively altering the behavior of the ORB.

At its simplest usage, the interceptor is useful for tracing through code. Because you can see the messages being sent between clients and servers, you can determine exactly how the ORB is processing requests.

If you are building a more sophisticated application such as a monitoring tool or security layer, interceptors give you the information and control you need to enable these lower-level applications. For example, you could develop an application that monitors the activity of various servers and performs load balancing.

Types of Portable Interceptors

There are two kinds of Portable Interceptors defined by the OMG specification:

– **Request Interceptors** can enable the ORB services to transfer context information between clients and servers. Request Interceptors are further divided into *Client Request Interceptors* and *Server Request Interceptors*.

– An **IOR interceptor** is used to enable an ORB service to add information in an IOR describing the server's or object's ORB-service-related capabilities. For example, a security service (like SSL) can add its tagged component into the IOR so that clients recognizing that component can establish the connection with the server based on the information in the component.

Portable Interceptor classes and interfaces

All Portable Interceptors implement one of the following base interceptor API classes which are defined and implemented by VisiBroker for .NET:

– `ClientRequestInterceptor`

– `ServerRequestInterceptor`

– `IORInterceptor`

Interceptor class

All the interceptor classes mentioned above are derived from a common class: `Interceptor`. This `Interceptor` class has defined common methods that are available to its inherited classes.

Request Interceptor

A *request interceptor* is used to intercept flow of a request/reply sequence at specific interception points so that services can transfer context information between clients and servers. For each interception point, the ORB gives an object through which the Interceptor can access request information. There are two kinds of request interceptors and their respective request information interfaces:

– `ClientRequestInterceptor` and `ClientRequestInfo`

– `ServerRequestInterceptor` and `ServerRequestInfo`

ClientRequestInterceptor

`ClientRequestInterceptor` has its interception points implemented on the client side. There are five interception points defined in `ClientRequestInterceptor` by OMG as shown in the following table.

Interception Points	Description
SendRequest	Lets a client-side Interceptor query a request and modify the service context before the request is sent to the server.
SendPoll	Lets a client-side Interceptor query a request during a Time-Independent Invocation (TII)[1] polling get reply sequence.
ReceiveReply	Lets a client-side Interceptor query the reply information after it is returned from the server and before the client gains control.
ReceiveException	Lets a client-side Interceptor query the exception's information, when an exception occurs, before the exception is sent to the client.
ReceiveOther	Lets a client-side Interceptor query the information which is available when a request result other than normal reply or an exception is received.

[1]TII is not implemented in VisiBroker for .NET. As a result, the SendPoll( ) interception point will never be invoked.

Client-side rules

The following are the client-side rules:

– The starting interception points are: `SendRequest` and `SendPoll`. On any given request/reply sequence, one and only one of these interception points is called.

– The ending interception points are: `ReceiveReply`, `ReceiveException` and `ReceiveOther`.

– There is no intermediate interception point.

– An ending interception point is called if and only if `SendRequest` or `SendPoll` runs successfully.

– A `ReceiveException` is called with the system exception `BAD_INV_ORDER` with a minor code of 4 (ORB has shutdown) if a request is canceled because of ORB shutdown.

– A `ReceiveException` is called with the system exception `TRANSIENT` with a minor code of 3 if a request is canceled for any other reason.

Successful invocations	`SendRequest` is followed by `ReceiveReply` — a start point is followed by an end point
Retries	`SendRequest` is followed by `ReceiveOther` — a start point is followed by an end point

ServerRequestInterceptor

`ServerRequestInterceptor` has its interception points implemented on the server-side. There are five interception points defined in `ServerRequestInterceptor`. The following table shows the `ServerRequestInterceptor` Interception points.

Interception Points	Description
`ReceiveRequestServiceContexts`	Lets a server-side Interceptor get its service context information from the incoming request and transfer it to `PortableInterceptor.Current`'s slot.
`ReceiveRequest`	Lets a server-side Interceptor query request information after all information, including operation parameters, is available.
`SendReply`	Lets a server-side Interceptor query reply information and modify the reply service context after the target operation has been invoked and before the reply is returned to the client.
`SendException`	Lets a server-side Interceptor query the exception's information and modify the reply service context, when an exception occurs, before the exception is sent to the client.
`SendOther`	Lets a server-side Interceptor query the information which is available when a request result other than normal reply or an exception is received.

Server-side rules

The server-side rules are listed as below:

– The starting interception point is: `ReceiveRequestServiceContexts`. This interception point is called on any given request/reply sequence.

– The ending interception points are: `SendReply`, `SendException` and `SendOther`. On any given request/reply sequence, one and only one of these interception points is called.

– The intermediate interception point is `ReceiveRequest`. It is called after `ReceiveRequestServiceContexts` and before an ending interception point.

– On an exception, `ReceiveRequest` may not be called.

– An ending interception point is called if and only if `ReceiveRequestServiceContext` runs successfully.

– A `SendException` is called with the system exception `BAD_INV_ORDER` with a minor code of 4 (ORB has shutdown) if a request is canceled because of ORB shutdown.

– A `SendException` is called with the system exception `TRANSIENT` with a minor code of 3 if a request is canceled for any other reason.

Successful invocations	The order of interception points: `ReceiveRequestServiceContexts`, `ReceiveRequest`, `SendReply` — a start point is followed by an intermediate point which is followed by an end point .

IORInterceptor

`IORInterceptor` gives applications the ability to add information describing the server's or object's ORB service related capabilities to object references to enable the ORB service implementation in the client to function properly. This is done by calling the interception point, `EstablishComponents`. An instance of `IORInfo` is passed to the interception point.

PortableInterceptor (PI) Current

The `PortableInterceptor.Current` object (hereafter referred to as `PICurrent`) is a table of slots that can be used by Portable Interceptors implementations to associate thread-specific information with the currently active request context. Use of PICurrent is optional, and would typically be used if a client's thread-specific information is required within an Interceptor.

`PICurrent` is obtained through a call to:

```
ORB.ResolveInitialReferences("PICurrent");
```

Codec

The `Codec` provides a mechanism for interceptors to transfer components between their IDL data types and their CDR encapsulation representations.

CodecFactory

This class is used to create a `Codec` object by specifying the encoding format, the major and minor versions. `CodecFactory` can be obtained a call to:
```
ORB.ResolveInitialReferences("CodecFactory");
```

Creating a Portable Interceptor

The generic steps to create a Portable Interceptor are:

1 The Interceptor must be inherited from one of the following Interceptor interfaces:

  - `ClientRequestInterceptor`

  - `ServerRequestInterceptor`

  - `IORInterceptor`

1 The Interceptor implements one or more interception points that are available to the Interceptor.

2 The Interceptor can be named or anonymous. All names must be unique among all Interceptors of the same type. However, any number of anonymous Interceptors can be registered with the ORB.

Registering Portable Interceptors

Portable Interceptors must be registered with the ORB before they can be used. To register a Portable Interceptor the `janeva.orb.init` property is provided.

```
-janeva.orb.init pi_class_name[,assembly_name]
```

Note, that it is possible to specify a list of `janeva.pi.init` settings to configure multiple Portable Interceptors:

```
-janeva.orb.init pi_1 -janeva.orb.init pi_2 -janeva.orb.init pi_n
```

Each `janeva.orb.init` instance does not overwrite the previous one, but adds it to a Portable Interceptor list.

VisiBroker for .NET extensions to Portable Interceptors

POA scoped Server Request Interceptors

Portable Interceptors specified by OMG are scoped globally. VisiBroker for .NET has defined "POA scoped Server Request Interceptor", a public extension to the Portable Interceptors, by adding a new module call `PortableInterceptorExt`. This new module holds a local interface, `IORInfoExt`, which is inherited from `PortableInterceptor.IORInfo` and has additional methods to install POA scoped server request interceptor.

IORInfoExt Interface

```
using PortableInterceptor;

namespace PortableInterceptorExt {
   public interface IORInfoExt : IORInfo {
      void AddServerRequestInterceptor(
         ServerRequestInterceptor interceptor);
      string FullPoaName();
   }
}
```

Limitations of the Portable Interceptors Implementation

The following are limitations of the Portable Interceptor implementation.

ClientRequestInfo:

– `Arguments`, `Result`, `Exceptions`, `Contexts`, and `OperationContexts` are only available for DII invocations.

– `ReceivedException` and `ReceivedExceptionId` will always return a `CORBA.UNKNOWN` exception and its respective repository id if a user exception is thrown by the application.

ServerRequestInfo:

– `Exceptions` does not return any value; it will raise a `CORBA.NO_RESOURCES` exception in both dynamic invocations and static stub based invocation.

– `Contexts` returns the list of contexts that are available during operation invocation.

– `SendingException` returns the correct user exception only in the case of dynamic invocation (provided the user exception can be inserted into an `Any` or its `TypeCode` information is available).

– `Arguments`, `Result`, `Contexts`, and `OperationContexts` are only available for DSI invocations.

10

Using Portable Object Adapters

What is a Portable Object Adapter?

The Portable Object Adapter (POA) is a service used to take incoming requests from clients and map those requests to the appropriate object implementations. For J2EE developers, it might be useful to think of a POA as being similar to an EJB Container in that it is responsible for mapping invocations to the set of objects it logically contains.

As with any container, you can think of the POA as having an external perspective and an internal perspective. The internal model of the POA is in terms of *Servant* objects: these are the objects that implement the user's business logic. The external model of the POA is in terms of *Object References*, which are references that can be used in distributed system invocations (for example, these Object References are analogous to instances of `java.rmi.Remote` in RMI/J2EE terminology, CORBA Object References in CORBA terminology, or instances of `MarshalByRefObject` in .NET Remoting terminology). The task of the POA is to map between external Object References and internal Servant objects.

The POA is an intermediary between the implementation of an object and the ORB. In its role as an intermediary, a POA routes requests to *Servants* and, as a result may cause Servants to run and create child POAs if necessary.

Servers can support multiple POAs. At least one POA must be present, which is called the *Root POA*. The Root POA is created automatically for you. The set of POAs is hierarchical; all POAs have the Root POA as their ancestor.

Servant Managers locate and assign Servants to objects for the POA. When an Object Reference is assigned to a Servant, it is called an *active object* and the Servant is said to *incarnate* the active object. Every POA has one *Active Object Map* which keeps track of the object IDs of active objects and their associated active Servants.

POA terminology

Contained in the following table are definitions of some terms with which you should become more familiar as you read through this section.

Steps for creating and using POAs

Term	Term
Active Object Map	Table that maps active Object References (through their object IDs) to Servants. There is one Active Object Map per POA.
Adapter activator	Object that can create a POA on demand when a request is received for a child POA that does not exist.
etherealize	Remove the association between a Servant and an Object Reference.
incarnate	Associate a Servant with an Object Reference
ObjectID	Way to identify an Object Reference within the object adapter. An ObjectID can be assigned by the object adapter or the application and is unique only within the object adapter in which it was created. Servants are associated with Object References through ObjectIDs
Persistent Object	Object References that live beyond the server process that created them.
POA Manager	Object that controls the state of the POA; for example, whether the POA is receiving or discarding incoming requests
Policy	Object that controls the behavior of the associated POA and the objects the POA manages.
Root POA	Each ORB is created with one POA called the Root POA. You can create additional POAs (if necessary) from the Root POA.
Servant	Any code that implements the methods of an Object Reference, but is not the Object Reference itself.
Servant Manager	An object responsible for managing the association of objects with Servants, and for determining whether an object exists. More than one Servant Manager can exist.
Transient Object	An Object Reference that lives only within the process that created it.

Although the exact process can vary, the basic steps that occur during a POA life cycle are:

1 Define the POA's policies.

2 Create the POA.

3 Activate the POA through its POA manager.

4 Create and activate Servants.

5 Create and use Servant Managers.

6 Use adapter activators.

Depending on your needs, some of these steps may be optional. For example, you only have to activate the POA if you want it to process requests.

POA policies

Each POA has a set of policies that define its characteristics. When creating a new POA, you can use the default set of policies or use different values to suit your requirements. You can only set the policies when creating a POA; you can not change the policies of an existing POA. POAs do not inherit the policies from their parent POA.

The following sections list the POA policies, their values, and the default value (used by the Root POA).

Thread policy

The thread policy specifies the threading model to be used by the POA. The valid values for the thread policy are described in the following table.

Value	Description
ORB_CTRL_MODEL	(Default) The default POA threading model is multi-threaded, meaning that concurrent invocations are dispatched to multiple threads concurrently. Note that this means that Servant implementations must be thread-safe. If Servants are not thread safe, they must either be made so (using appropriate locking) or a different threading policy must be used for non-thread-safe Servants.
SINGLE_THREAD_MODEL	The POA processes requests sequentially. In a multi-threaded environment, all calls made by the POA to Servants and Servant Managers are thread-safe.
MAIN_THREAD_MODEL	Calls are processed on a distinguished *main* thread. Requests for all main-thread POAs are processed sequentially. In a multi-threaded environment, all calls processed by all POAs with this policy are thread-safe. The application programmer designates the main thread by calling ORB.Run() or ORB.PerformWork(). For more information about these methods, see "Activating objects".

Lifespan policy

The lifespan policy specifies the lifespan of the objects implemented in the POA. The valid values for the lifespan policy are listed in the following table.

Value	Description
TRANSIENT	(Default) A transient object activated by a POA cannot outlive the POA that created it. Once the POA is deactivated, an OBJECT_NOT_EXIST exception occurs if an attempt is made to use any object references generated by the POA.
PERSISTENT	A persistent object activated by a POA can outlive the process in which it was first created. Requests invoked on a persistent object may result in the implicit activation of a process, a POA and the Servant that implements the object.

Object ID Uniqueness policy

The Object ID Uniqueness policy allows a single Servant to be shared by many Object References. The valid values for the Object ID Uniqueness policy are listed in the following table.

Value	Description
UNIQUE_ID	(Default) Activated Servants support only one Object ID.
MULTIPLE_ID	Activated Servants can have one or more Object IDs. The Object ID must be determined within the method being invoked at run time.

ID Assignment policy

The ID assignment policy specifies whether object IDs are generated by server applications or by the POA. The valid values for the ID Assignment policy are listed in the following table.

Value	Description
USER_ID	Objects are assigned object IDs by the application.
SYSTEM_ID	(Default) Objects are assigned object IDs by the POA. If the PERSISTENT policy is also set, object IDs must be unique across all instantiations of the same POA.

Typically, USER_ID is for persistent objects, and SYSTEM_ID is for transient objects. If you want to use SYSTEM_ID for persistent objects, you can extract them from the Servant or Object Reference.

Servant Retention policy

The Servant Retention policy specifies whether the POA retains active Servants in the Active Object Map. The valid values for the Servant Retention policy are listed in the following table.

Value	Description
RETAIN	(Default) The POA tracks Object Reference activations in the Active Object Map. RETAIN is usually used with ServantActivators or explicit activation methods on POA.
NON_RETAIN	The POA does not retain active Servants in the Active Object Map. NON_RETAIN must be used with ServantLocators.

ServantActivators and ServantLocators are types of Servant Managers. For more information on Servant Managers, see "Using Servants and Servant Managers".

Request Processing policy

The Request Processing policy specifies how requests are processed by the POA. The valid values for the Request Processing policy are listed in the following table.

Value	Description
USE_ACTIVE_OBJECT_MAP_ONLY	(Default) If the Object ID is not listed in the Active Object Map, an OBJECT_NOT_EXIST exception is returned. The POA must also use the RETAIN policy with this value.
USE_DEFAULT_SERVANT	If the Object ID is not listed in the Active Object Map or the NON_RETAIN policy is set, the request is dispatched to the default Servant. If no default Servant has been registered, an OBJ_ADAPTER exception is returned. The POA must also use the MULTIPLE_ID policy with this value.
USE_SERVANT_MANAGER	If the Object ID is not listed in the Active Object Map or the NON_RETAIN policy is set, the Servant Manager is used to obtain a Servant.

Implicit Activation policy

The Implicit Activation policy specifies whether the POA supports implicit activation of Servants. The valid values for the Implicit Activation policy are listed in the following table.

Value	Description
IMPLICIT_ACTIVATION	The POA supports implicit activation of Servants. There are two ways to activate the Servants as follows: ■ Converting them to an Object Reference with `PortableServer.POA.ServantToReference()`. ■ Invoking `This_()` on the Servant. The POA must also use the SYSTEM_ID and RETAIN policies with this value.
NO_IMPLICIT_ACTIVATION	(Default) The POA does not support implicit activation of Servants.

Bind Support policy

The Bind Support policy (a VisiBroker-specific policy) controls the registration of POAs and active objects with the VisiBroker Smart Agent (osagent). If you have several thousands objects, it is not feasible to register all of them with the osagent. Instead, you can register the POA with the osagent. When a client request is made, the POA name and the object ID is included in the bind request so that the osagent can correctly forward the request. The valid values for the Bind Support policy are listed in the following table.

Value	Description
BY_INSTANCE	All active objects are registered with the osagent. The POA must also use the PERSISTENT and RETAIN policy with this value.
BY_POA	(Default) Only POAs are registered with the osagent. The POA must also use the PERSISTENT policy with this value.
NONE	Neither POAs nor active objects are registered with the smart agent.

Note:

The Root POA is created with the NONE activation policy.

Creating POAs

To implement objects using the POA, at least one POA object must exist on the server. To ensure that a POA exists, a Root POA is provided during the ORB initialization. This POA uses the default POA policies described earlier in this section.

Once the Root POA is obtained, you can create child POAs that implement a specific server-side policy set.

POA naming convention

Each POA keeps track of its name and its full POA name (the full hierarchical path name.) The hierarchy is indicated by a slash (/). For example, /A/B/C means that POA C is a child of POA B, which in turn is a child of POA A. The first slash indicates the Root POA. If the BindSupport:BY_POA policy is set on POA C, then /A/B/C is registered and the client binds with /A/B/C.

If your POA name contains escape characters or other delimiters, VisiBroker for .NET precedes these characters with a double back slash (\\) when recording the names internally.

Obtaining the Root POA

The following code sample illustrates how a server application can obtain its Root POA.

```
// Initialize the ORB.
CORBA.ORB orb = CORBA.ORB.Init(args);

// get a reference to the Root POA
PortableServer.POA rootPOA =
      POAHelper.Narrow(orb.ResolveInitialReferences("RootPOA"));
```

Note:

The `ResolveInitialReferences` method returns a value of type `CORBA.Object`. You are responsible for narrowing the returned object reference to the desired type, which is PortableServer.POA in the previous example.

Setting the POA policies

Policies are not inherited from the parent POA. If you want a POA to have a specific characteristic, you must identify all the policies that are different from the default value. For more information about POA policies, see "POA policies".

```
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(LifespanPolicyValue.PERSISTENT),
   rootPOA.CreateRequestProcessingPolicy(
      RequestProcessingPolicyValue.USE_DEFAULT_SERVANT),
   rootPOA.CreateIdUniquenessPolicy(IdUniquenessPolicyValue.MULTIPLE_ID)
};
```

Creating and activating the POA

A POA is created using `CreatePOA` on its parent POA. You can name the POA anything you like; however, the name must be unique with respect to all other POAs with the same parent. If you attempt to give two POAs the same name, a CORBA exception (`AdapterAlreadyExists`) is raised.

To create a new POA, use `CreatePOA` as follows:

```
CreatePOA("ThePOAName", thePOAManager, thePolicyList);
```

The POA manager (`<POAManager>`) controls the state of the POA (for example, whether it is processing requests). If null is passed to `CreatePOA` as the POA manager name, a new POA manager object is created and associated with the POA. Typically, you will want to have the same POA manager for all POAs. For more information about the POA manager, see "Managing POAs with the POA manager".

POA managers (and POAs) are not automatically activated once created. Use `Activate()` to activate the POA manager associated with your POA. The following code sample is an example of creating a POA and activating the POA manager.

```
// Create policies for our persistent POA
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(LifespanPolicyValue.PERSISTENT)
};

// Create myPOA with the right policies
PortableServer.POA myPOA =
   rootPOA.CreatePOA("bank_agent_poa",
      rootPOA.ThePOAManager,
      policies);

// Activate the POA manager
rootPOA.ThePOAManager.Activate();
```

Activating objects

When Object References are associated with an active Servant, if the POA's Servant Retention Policy is `RETAIN`, the associated object ID is recorded in the Active Object Map and the object is activated. Activation can occur in one of several ways:

– **Explicit activation**—The server application itself explicitly activates objects by calling `ActivateObject` or `ActivateObjectWithId`.

– **On-demand activation**—The server application instructs the POA to activate objects through a user-supplied Servant Manager. The Servant Manager must first be registered with the POA through `SetServantManager`.

– **Implicit activation**—The server activates objects solely by in response to certain operations. If a Servant is not active, there is nothing a client can do to make it active (for example, requesting for an inactive object does not make it active.)

– **Default Servant**—The POA uses a single Servant to implement all of its objects.

Activating objects explicitly

By setting `IdAssignmentPolicy.SYSTEM_ID` on a POA, objects can be explicitly activated without having to specify an object ID. The server invokes `ActivateObject` on the POA which activates, assigns and returns an object ID for the object. This type of activation is most common for transient objects. No Servant Manager is required since neither the object nor the Servant is needed for very long.

Objects can also be explicitly activated using object IDs. A common scenario is during server initialization where the user invokes `ActivateObjectWithId` to activate all the objects managed by the server. No Servant Manager is required since all the objects are already activated. If a request for a nonexistent object is received, an `OBJECT_NOT_EXIST` exception is raised. This has obvious negative effects if your server manages large numbers of objects.

This code sample is an example of explicit activation using `ActivateObjectWithId`.

```
// Create the account manager Servant.
Servant managerServant = new AccountManagerImpl(rootPoa);

// Activate the newly created Servant.
byte[] managerId = orb.StringToObjectId("BankManager");
testPoa.ActivateObjectWithId(managerId, managerServant);

// Activate the POAs
testPoa.ThePOAManager.Activate();
```

Activating objects on demand

On-demand activation occurs when a client requests an object that does not have an associated Servant. After receiving the request, the POA searches the Active Object Map for an active Servant associated with the object ID. If none is found, the POA invokes `Incarnate` on the Servant Manager which passes the object ID value to the Servant Manager. The Servant Manager can do one of three things:

– Find an appropriate Servant which then performs the appropriate operation for the request.

– Raise an `OBJECT_NOT_EXIST` exception that is returned to the client.

– Forward the request to another object.

The POA policies determine any additional steps that may occur. For example, if `RequestProcessingPolicy.USE_SERVANT_MANAGER` and `ServantRetentionPolicy.RETAIN` are enabled, the Active Object Map is updated with the Servant and object ID association. if `RequestProcessingPolicy.USE_SERVANT_MANAGER` and `ServantRetentionPolicy.RETAIN` are enabled, the Active Object Map is updated with the Servant and object ID association.

Activating objects implicitly

A Servant can be implicitly activated by certain operations if the POA has been created with `ImplicitActivationPolicy.IMPLICIT_ACTIVATION`, `IdAssignmentPolicy.SYSTEM_ID`, and `ServantRetentionPolicy.RETAIN`. Implicit activation can occur with:

– `POA.ServantToReference` method

– `POA.ServantToId` method

– `This_()` Servant method

If the POA has `IdUniquenessPolicy.UNIQUE_ID` set, implicit activation can occur when any of the above operations are performed on an inactive Servant.

If the POA has `IdUniquenessPolicy.MULTIPLE_ID` set, `ServantToReference` and `ServantToId` operations always perform implicit activation, even if the Servant is already active.

Activating with the default Servant

Use the `RequestProcessing.USE_DEFAULT_SERVANT` policy to have the POA invoke the same Servant no matter what the object ID is. This is useful when little data is associated with each object.

This is an example of activating all objects with the same Servants

```
using System;
using System.IO;
using PortableServer;
using CORBA;

public class Server {
   static void Main(string [] args) {
      try {
         // initialize the ORB
         ORB orb = ORB.Init(args);

         // get a reference to the root POA
         POA rootPOA =
            POAHelper.Narrow(orb.ResolveInitialReferences("RootPOA"));

         // create policies for our persistent POA
         Policy[] policies = {
            rootPOA.CreateLifespanPolicy(
               LifespanPolicyValue.PERSISTENT),
            rootPOA.CreateRequestProcessingPolicy(
               RequestProcessingPolicyValue.USE_DEFAULT_SERVANT),
            rootPOA.CreateIdUniquenessPolicy(
               IdUniquenessPolicyValue.MULTIPLE_ID)
         };

         // create myPOA with the right policies
         POA myPOA = rootPOA.CreatePOA("bank_default_servant_poa",
            rootPOA.ThePOAManager,
            policies );

         // create the servant
         AccountManagerImpl managerServant = new AccountManagerImpl();
         myPOA.SetServant(managerServant);

         // Activate the POA manager
         rootPOA.ThePOAManager.Activate();

         // Generate the reference and write it out. One for each
         // Checking and Savings account type. Note that we are not
         // creating any servants here and just manufacturing a
         // reference which is not yet backed by a servant.
```

```
                      // Write out checking object ID
                      try {
                         CORBA.Object objref = myPOA.CreateReferenceWithId(
                            orb.StringToObjectId("CheckingAccountManager"),
                            "IDL:Bank/AccountManager:1.0");

                         StreamWriter writer = new StreamWriter("cref.dat");
                         writer.WriteLine(orb.ObjectToString(objref));
                         writer.Close();
                      }
                      catch (Exception e) {
                         Console.WriteLine("Error writing the IOR for
                            CheckingAccountManager to file");
                         Console.WriteLine(e);
                      }

                      try {
                         // Write out savings object ID
                         CORBA.Object objref = myPOA.CreateReferenceWithId(
                            orb.StringToObjectId("SavingsAccountManager"),
                            "IDL:Bank/AccountManager:1.0");
                         StreamWriter writer = new StreamWriter("sref.dat");
                         writer.WriteLine(orb.ObjectToString(objref));
                         writer.Close();

                      }
                      catch (Exception e) {
                         Console.WriteLine("Error writing the IOR for
                            SavingsAccountManager to file");
                         Console.WriteLine(e);
                      }

                      Console.WriteLine("DefaultServantServer is ready.");
                      // Wait for incoming requests
                      orb.Run();
                   }
                   catch(Exception e) {
                      Console.WriteLine(e);
                   }
                }
             }
```

Deactivating objects

A POA can remove a Servant from its Active Object Map. This may occur, for example, as a form of garbage-collection scheme. When the Servant is removed from the map, it is deactivated. You can deactivate an object using `DeactivateObject()`. When an object is deactivated, it doesn't mean this object is lost forever. It can always be reactivated at a later time.

Using Servants and Servant Managers

Servant Managers perform two types of operations: find and return a Servant, and deactivate a Servant. They allow the POA to activate objects when a request for an inactive object is received. Servant Managers are optional. For example, Servant Managers are not needed when your server loads all objects at startup. Servant Managers may also inform clients to forward requests to another object using the `ForwardRequest` exception.

A Servant is an active instance of an implementation. The POA maintains a map of the active Servants and the object IDs of the Servants. When a client request is received, the POA first checks this map to see if the object ID (embedded in the client request)

has been recorded. If it exists, then the POA forwards the request to the Servant. If the object ID is not found in the map, the Servant Manager is asked to locate and activate the appropriate Servant. This is only an example scenario; the exact scenario depends on what POA policies you have in place.

There are two types of Servant Managers: *Servant Activator* and *Servant Locator*. The type of policy already in place determines which type of Servant Manager is used. For more information on POA policy, see "POA policies". Typically, a Servant Activator activates persistent objects and a Servant Locator activates transient objects.

To use Servant Managers, `RequestProcessingPolicy.USE_SERVANT_MANAGER` must be set as well as the policy which defines the type of Servant Manager (`ServantRetentionPolicy.RETAIN` for Servant Activator or `ServantRetentionPolicy.NON_RETAIN` for Servant Locator.)

ServantActivators

ServantActivators are used when `ServantRetentionPolicy.RETAIN` and `RequestProcessingPolicy.USE_SERVANT_MANAGER` are set.

Servants activated by this type of Servant Manager are tracked in the Active Object Map.

The following events occur while processing requests using Servant Activators:

1 A client request is received (client request contains POA name, the object ID, and a few others.)

2 The POA first checks the Active Object Map. If the object ID is found there, the operation is passed to the Servant, and the response is returned to the client.

3 If the object ID is not found in the Active Object Map, the POA invokes `Incarnate` on a Servant Manager. `Incarnate` passes the object ID and the POA in which the object is being activated.

4 The Servant Manager locates the appropriate Servant.

5 The Servant ID is entered into the active object map, and the response is returned to the client.

Note:

The `Etherealize` and `Incarnate` method implementations are user-supplied code.

At a later date, the Servant can be deactivated. This may occur from several sources, including the `DeactivateObject` operation, deactivation of the POA manager associated with that POA, and so forth. More information on deactivating objects is described in "Deactivating objects".

The following is the implementation of the ServantActivator.

```
using System;
using System.Threading;
using System.Collections;

public class
   AccountManagerActivator : PortableServer.ServantActivator {

   private Hashtable _objectMap = new Hashtable();

   public AccountManagerActivator() {
      Console.WriteLine("AccountManagerActivator() called.");
      // Populate the Object Map.
      _objectMap.Add("SavingsAccountManager",
         new SavingsAccountManagerImpl());
      _objectMap.Add("CheckingAccountManager",
         new CheckingAccountManagerImpl());
   }

   public PortableServer.Servant Incarnate(byte[] oid,
      PortableServer.POA adapter) {
```

```
      try {
         Console.WriteLine(
            "AccountManagerActivator.Incarnate() called.");

         string accountType = CORBA.ORB.Init().ObjectIdToString(oid);
         Console.WriteLine("\nAccountManagerActivator.Incarnate()
            called with ID = " + accountType);

         new ObjectDeactivator(adapter, oid);
         return (PortableServer.Servant) _objectMap[accountType];
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
      return null;
   }

   public void Etherealize(byte[] oid,
      PortableServer.POA adapter,
      PortableServer.Servant serv,
      bool cleanupInProgress,
      bool remainingActivations) {

      Console.WriteLine("Etheralize() called.");

      try {
         string accountType = CORBA.ORB.Init().ObjectIdToString(oid);
         Console.WriteLine("\nAccountManagerActivator.Etherealize()
            called with ID = " + accountType);
      }
      catch (Exception e) {
         Console.WriteLine(e);
      }
   }

   private const int ONE_SECOND = 1000;

   private class ObjectDeactivator {
      private PortableServer.POA _adapter;
      private byte[] _oid;

      public ObjectDeactivator(PortableServer.POA adapter, byte[] oid) {
         _adapter = adapter;
         _oid = oid;
         new Thread(new ThreadStart(Deactivate)).Start();
      }

      public void Deactivate() {
         Console.WriteLine("Deactivate() called.");
         try {
            Thread.Sleep(ONE_SECOND * 15);
            Console.WriteLine("\nDeactivating the object with ID = " +
            CORBA.ORB.Init().ObjectIdToString(_oid));
            _adapter.DeactivateObject(_oid);
         }
         catch (Exception e) {
            Console.WriteLine(e);
         }
      }
   }
}
```

The following is a server implementation similar to the code example in "Activating with
the default Servant". In this example we highlight the differences for activating Servants
with the ServantActivator.

```
// create policies for our persistent POA
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(
      LifespanPolicyValue.PERSISTENT),
   rootPOA.CreateRequestProcessingPolicy(
      RequestProcessingPolicyValue.USE_SERVANT_MANAGER)
};

// create myPOA with the right policies
POA myPOA =
   rootPOA.CreatePOA("bank_servant_activator_poa",
      rootPOA.ThePOAManager,
      policies );

// Create the servant activator servant and get its reference
ServantActivator sa = new AccountManagerActivator(orb);

// Set the servant activator on our POA
myPOA.SetServantManager(sa);

// Activate the POA manager
rootPOA.ThePOAManager.Activate();
```

ServantLocators

In many situations, the POA's Active Object Map could become quite large and consume memory. To reduce memory consumption, a POA can be created with `RequestProcessingPolicy.USE_SERVANT_MANAGER` and `ServantRetentionPolicy.NON_RETAIN`, meaning that the Servant-to-object association is not stored in the Active Object Map. Since no association is stored, Servant Locator Servant Managers are invoked for each request.

The following events occur while processing requests using Servant Locators:

1 A client request, which contains the POA name and the object id, is received.

2 Since `ServantRetentionPolicy.NON_RETAIN` is used, the POA does not search the Active Object Map for the object ID.

3 The POA invokes `Preinvoke` on a Servant Manager. `Preinvoke` passes the object ID, the POA in which the object is being activated, and a few other parameters.

4 The Servant Locator locates the appropriate Servant.

5 The operation is performed on the Servant and the response is returned to the client.

6 The POA invokes `Postinvoke` on the Servant Manager.

Note:

The `Preinvoke` and `Postinvoke` method implementations are user-supplied code.

The following is the implementation of the ServantLocator.

```
using System;
using CORBA;
using PortableServer;
using PortableServer.ServantLocatorNS;

public class AccountManagerLocator : ServantLocator {
   private ORB _orb;

   public AccountManagerLocator(ORB orb) {
      _orb = orb;
   }

   public Servant Preinvoke (byte[] oid, POA adapter,
      string operation, out object theCookie) {
```

```
            string accountType = _orb.ObjectIdToString(oid);
            theCookie = null;

            Console.WriteLine("\nAccountManagerLocator.preinvoke
               called with ID = {0}\n", accountType);
            if (accountType.Equals("SavingsAccountManager")) {
               return new SavingsAccountManagerImpl();
            }

            return new CheckingAccountManagerImpl();
         }

      public void Postinvoke (byte[] oid,
         POA adapter,
         string operation,
         object theCookie,
         Servant theServant) {
         string id = _orb.ObjectIdToString(oid);
         Console.WriteLine("\nAccountManagerLocator.postinvoke
            called with ID = {0}\n", id);
      }
   }
}
```

The following is a server implementation similar to the code example in "Activating with the default Servant". In this example we highlight the differences for activating Servants using the ServantLocator.

```
// Create policies for our POA. We need persistence life
// span, use servant manager request processing policies and
// non retain retention policy. This non retain policy will let
// us use the servant locator instead of servant activator
CORBA.Policy[] policies = {
   rootPOA.CreateLifespanPolicy(
      LifespanPolicyValue.PERSISTENT),
   rootPOA.CreateServantRetentionPolicy(
      ServantRetentionPolicyValue.NON_RETAIN),
   rootPOA.CreateRequestProcessingPolicy(
      RequestProcessingPolicyValue.USE_SERVANT_MANAGER)
};

// create myPOA with the right policies
POA myPOA = rootPOA.CreatePOA("bank_servant_locator_poa",
   rootPOA.ThePOAManager, policies);

// Create the servant locator servant and get its reference
ServantLocator sl = new AccountManagerLocator(orb);

// Set the servant activator on our POA
myPOA.SetServantManager(sl);

// Activate the POA manager
rootPOA.ThePOAManager.Activate();
```

Managing POAs with the POA manager

A POA manager controls the state of the POA (whether requests are queued or discarded), and can deactivate the POA. Each POA is associated with a POA manager object. A POA manager can control one or several POAs.

A POA manager is associated with a POA when the POA is created. You can specify the POA manager to use, or specify null to have a new POA manager created.

The following is an example of naming the POA and its POA manager:

```
POA myPOA = rootPOA.CreatePOA("MyPOA",
   rootPOA.ThePOAManager, policies);
```

```
POA myPOA = rootPOA.CreatePOA("MyPOA", null, policies);
```

A POA manager is "destroyed" when all its associated POAs are destroyed.

A POA manager can have the following four states:

– `Holding`

– `Active`

– `Discarding`

– `Inactive`

These states in turn determine the state of the POA. They are each described in detail in the following sections.

Getting the current state

To get the current state of the POA manager, use

```
State state = manager.GetState();
```

Holding state

By default, when a POA manager is created, it is in the `Holding` state. When the POA manager is in the `Holding` state, the POA queues all incoming requests.

Requests that require an adapter activator are also queued when the POA manager is in the `Holding` state.

To change the state of a POA manager to `Holding`, use

```
manager.HoldRequests(waitForCompletion);
```

`waitForCompletion` is `Boolean`. If `false`, this operation returns immediately after changing the state to `Holding`. If `true`, this operation returns only when all requests started prior to the state change have completed or when the POA manager is changed to a state other than `Holding`. `AdapterInactive` is the exception raised if the POA manager was in the `Inactive` state prior to calling this operation.

Note:

POA managers in the `Inactive` state cannot change to the `Holding` state.

Any requests that have been queued but not yet started will continue to be queued during the `Holding` state.

Active state

When the POA manager is in the `Active` state, its associated POAs process requests.

To change the POA manager to the `Active` state, use

```
manager.Activate();
```

`AdapterInactive` is the exception raised if the POA manager was in the `Inactive` state prior to calling this operation.

Note:

POA managers currently in the `Inactive` state can not change to the `Active` state.

Discarding state

When the POA manager is in the `Discarding` state, its associated POAs discard all requests that have not yet started. In addition, the adapter activators registered with the associated POAs are not called. This state is useful when the POA is receiving too many requests. You need to notify the client that their request has been discarded and to resend their request. There is no inherent behavior for determining if and when the

POA is receiving too many requests. It is up to you to set up thread monitoring if so desired.

To change the POA manager to the `Discarding` state, use

```
manager.DiscardRequests(waitForCompletion);
```

The `waitForCompletion` option is `Boolean`. If `false`, this operation returns immediately after changing the state to `Holding`. If `true`, this operation returns only when all requests started prior to the state change have completed or when the POA manager is changed to a state other than `Discarding`. `AdapterInactive` is the exception raised if the POA manager was in the `Inactive` state prior to calling this operation.

Note:

POA managers currently in the `Inactive` state can not change to the `Discarding` state.

Inactive state

When the POA manager is in the `Inactive` state, its associated POAs reject incoming requests. This state is used when the associated POAs are to be shut down.

Note:

POA managers in the `Inactive` state cannot change to any other state.

To change the POA manager to the `Inactive` state, use

```
manager.Deactivate(etherealizeObjects, waitForCompletion);
```

After the state changes, if `etherealizeObjects` is `true`, then all associated POAs that have `ServantRetentionPolicy.RETAIN` and `RequestProcessingPolicy.USE_SERVANT_MANAGER` set call `Etherealize` on the Servant Manager for all active objects. If `etherealizeObjects` is `false`, then `Etherealize` is not called. The `waitForCompletion` option is `Boolean`. If `false`, this operation returns immediately after changing the state to `Inactive`. If `true`, this operation returns only when all requests started prior to the state change have completed or `Etherealize` has been called on all associated POAs (that have `ServantRetentionPolicy.RETAIN` and `RequestProcessingPolicy.USE_SERVANT_MANAGER`). `AdapterInactive` is the exception raised if the POA manager was in the `Inactive` state prior to calling this operation.

Listening and Dispatching: Server Engines, Server Connection Managers, and their properties

Note:

Policies that cover listener and dispatcher features are not supported by POAs. In order to provide these features, a VisiBroker for .NET-specific policy (`ServerEnginePolicy`) can be used.

VisiBroker for .NET provides a very flexible mechanism to define and tune endpoints for VisiBroker for .NET servers. An endpoint in this context is a destination for a communication channel for clients to communicate with servers. A *Server Engine* is a virtual abstraction for connection endpoint provided as a configurable set of properties.

A Server Engine abstraction can provide control in terms of:

– types of connection resources

– connection management

– threading model and request dispatching

Server Engine and POAs

A POA on VisiBroker for .NET can have many-to-many relationship with a Server Engine. A POA can be associated with many Server Engines and vice versa. The

manifestation of this fact is that a POA, and hence the Object References on the POA, can support multiple communication channels.

The simplest case is where POAs have their own unique single server engine. Here, requests for different POAs arrive on different ports. A POA can also have multiple server engines. In this scenario, a single POA supports requests coming from multiple input ports.

Notice that POAs can share server engines. When server engines are shared, the POAs listen to the same port. Even though the requests for (multiple) POAs arrive at the same port, they are dispatched correctly because of the POA name embedded in the request. This scenario occurs, for example, when you use a default server engine and create multiple POAs (without specifying a new server engine during the POA creation).

Server Engines are identified by a name and is defined the first time its name is introduced. By default VisiBroker for .NET defines three Server Engine names. They are:

– `iiop_tp`: TCP transport with thread pool dispatcher

– `iiop_ts`: TCP transport with thread per session dispatcher

– `iiop_tm`: TCP transport with main thread dispatcher

Associating a POA with a Server Engine

The default Server Engine associated with POA can be changed by using the property `vbroker.se.default`. For example, setting

```
vbroker.se.default=MySE
```

defines a new server engine with the name MySE. The Root POA and all child POAs created will be associated with this Server Engine by default.

A POA can also be associated with a particular ServerEngine explicitly by using the `SERVER_ENGINE_POLICY_TYPE` POA policy. For example:

```
// create ServerEngine policy value
Any seAny = orb.CreateAny();
StringSequenceHelper.Insert(seAny, new String [] {"mySE"});
Policy sePolicy = orb.CreatePolicy(
    PortableServerExt.SERVER_ENGINE_POLICY_TYPE.Value, seAny);

// create POA policies
Policy [] policies = {
    rootPOA.CreateLifespanPolicy(LifespanPolicyValue.PERSISTENT),
    sePolicy
};

// create POA with policies
POA myPOA = rootPOA.CreatePOA("bank_se_policy_poa",
    rootPOA.ThePOAManager,
    policies);
```

The POA has an IOR template, profiles for which, are obtained from the Server Engines associated with it.

If you don't specify a server engine policy, the POA assumes a server engine name of `iiop_tp` and uses the following default values:

```
vbroker.se.iiop_tp.host=null
vbroker.se.iiop_tp.proxyHost=null
vbroker.se.iiop_tp.scms=iiop_tp
```

To change the default server engine policy, enter its name using the `vbroker.se.default` property and define the values for all the components of the new server engine. For example:

```
vbroker.se.default=abc,def
vbroker.se.abc.host=cob
vbroker.se.abc.proxyHost=null
```

```
vbroker.se.abc.scms=cobscm1,cobscm2
vbroker.se.def.host=gob
vbroker.se.def.proxyHost=null
vbroker.se.def.scms=gobscm1
```

Defining Hosts for Endpoints for the Server Engine

Since Server Engines help define a connection's endpoints, the following properties are provided to specify their hosts:

– `vbroker.se.<se-name>.host=<host-URL>` (`vbroker.se.mySE.host=host.borland.com`, for example.)

– `vbroker.se.<se-name>.proxyHost=<proxy-host-URL-or-IP-address>` (`vbroker.se.mySE.proxyHost=proxy.borland.com`, for example.)

The `proxyHost` property can also take an IP address as its value. Doing so replaces the default hostname in the IOR with this IP address.

The endpoint abstraction of a Server Engine is further fine-grained in terms of configurable set of entities referred to as Server Connection Managers (SCM). A Server Engine can have multiple SCMs. SCMs are not shareable between Server Engines. SCMs are also identified using a name and are defined for a Server Engine using:

```
vbroker.se.<se-name>.scms=<SCM-name>[,<SCM-name>,...]
```

Server Connection Managers

The Server Connection Manager defines the configurable components of an endpoint. Its responsibilities are connection resource management, listening for requests, and dispatching requests to its associated POA. Three logical entities, defined through property groups, are provided by the SCM to fulfill these responsibilities:

– Manager

– Listener

– Dispatcher

Each SCM has one Manager, Listener, and Dispatcher. All three, when defined, form a single endpoint definition allowing clients to contact servers.

Manager

Manager is a set of properties defining the configurable portions of a connection resource. VisiBroker for .NET provides a manager of type `Socket`.

```
vbroker.se.<se-name>.scm.<scm-name>.manager.type=Local|Socket
```

You can specify the maximum number of concurrent connections acceptable to the server endpoint using the `connectionMax` property:

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMax=<integer>
```

Setting `connectionMax` to 0 (zero) indicates that there is no restriction on the number of connections, which is the default setting.

You specify the maximum number of idle seconds using the `connectionMaxIdle` property:

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMaxIdle=<seconds>
```

Setting `connectionMaxIdle` to 0 (zero) indicates that there is no timeout, which is the default setting.

Garbage collection time can also be specified for the Manager to garbage-collect idled connections. (Connections can idle after the `connectionMaxIdle` time until they are garbage-collected.) You can use the `garbageCollectTimer` property to specify the period of garbage collection in seconds:

```
vbroker.se.<se-name>.scm.<scm-name>.manager.garbageCollectTimer=<seconds>
```

Garbage collection time is specified through the following property:

```
vbroker.orb.gcTimeout=<seconds>
```

A value of 0 (zero) means that the connection will never be garbage collected.

Listener

The Listener is the SCM component that determines how and where the SCM listens for messages. Like the Manager, the Listener is also a set of properties. VisiBroker for .NET defines a IIOP listener for the TCP connections.

Since listeners are close to the actual underlying transport mechanism, their properties are not portable across listener types. Each listener type has its own set of properties, defined below.

IIOP listener properties

IIOP listeners need to define a port and (if desired) a proxy port in conjunction with their hosts. These are set using the `port` and `proxyPort` properties, as follows:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.port=<port>
vbroker.se.<se-name>.scm.<scm-name>.listener.proxyPort=<proxy-port>
```

Note:

If you do not set the `port` property (or set it to 0 [zero]), a random port will be selected. A 0 value for the `proxyPort` property means that the IOR will contain the actual port (defined by the `listener.port` property or selected by the system randomly). If it is not required to advertise the actual port, set the proxy port to a nonzero (positive) value.

Setting properties to define standard TCP socket options is also supported for send| receive buffer sizes, socket lingering time, and whether or not to keep inactive sockets alive. The following properties are provided for these mechanisms:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.rcvBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.sendBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.socketLinger=<seconds>
vbroker.se.<se-name>.scm.<scm-name>.listener.keepAlive=true|false
```

If for any reason you wish to simply use your system's defaults for the TCP socket properties, simply set the appropriate property to a value of 0 (zero).

VisiBroker for .NET additionally supports a property allowing you to specify your GIOP version:

```
vbroker.se.<se-name>.scm.<scm-name>.listener.giopVersion=<version>
```

Dispatcher

The Dispatcher defines a set of properties that determine how the SCM dispatches requests to threads. Three types of dispatchers are provided: `ThreadPool`, `ThreadSession`, and `MainThread`. You set the dispatcher type with the type property:

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.type=ThreadPool|ThreadSession|
MainThread
```

Further control is provided through the SCM for the `ThreadPool` dispatcher type. The `ThreadPool` defines the minimum and maximum number of threads that can be created in the thread pool, as well as the maximum time in seconds after which an idled thread is destroyed. These values are controlled with the following properties:

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMin=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMax=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMaxIdle=<seconds>
```

When to use these properties

There are many times where you need to change some of the server engine properties. The method for changing these properties depends on what you need. For example, suppose you want to change the port number. You could accomplish this by:

– Changing the default `listener.port` property

– Creating a new server engine

Changing the default `listener.port` property is the simplest method, but this affects all POAs that use the default server engine. This may or may not be what you want.

If you want to change the port number on a specific POA, then you'll have to create a new server engine, define the properties for this new server engine, and then reference the new server engine when creating the POA.

The previous sections show how to update the server engine properties. The following code shows how to create a POA with a user-defined server engine policy:

```
using System;
using System.IO;
using PortableServer;
using CORBA;

public class Server {
   static void Main(string [] args) {
      try {
         // initialize the ORB
         ORB orb = ORB.Init(args);

         // get a reference to the root POA
         POA rootPOA =
            POAHelper.Narrow(orb.ResolveInitialReferences("RootPOA"));

         // Create our server engine policy
         Any seAny = orb.CreateAny();
         StringSequenceHelper.Insert(seAny, new String [] {"mySe"});
         Policy sePolicy = orb.CreatePolicy(
            PortableServerExt.SERVER_ENGINE_POLICY_TYPE.Value, seAny);

         // create policies for our persistent POA
         Policy [] policies = {
            rootPOA.CreateLifespanPolicy(
               LifespanPolicyValue.PERSISTENT), sePolicy
         };

         // create myPOA with the right policies
         POA myPOA = rootPOA.CreatePOA("bank_se_policy_poa",
            rootPOA.ThePOAManager, policies);

         // create the servant
         AccountManagerImpl managerServant = new AccountManagerImpl();

         // Decide on the ID for the servant
         byte [] managerId = orb.StringToObjectId("BankManager");

         // Activate the servant
         myPOA.ActivateObjectWithId(managerId, managerServant);

         // Obtain the reference
         CORBA.Object objRef = myPOA.ServantToReference(managerServant);

         // Now write out the IOR
         try {
            StreamWriter writer = new StreamWriter("ior.dat");
            writer.WriteLine(orb.ObjectToString(objRef));
            writer.Close();
         }
```

```
            catch (Exception e) {
               Console.WriteLine("Error writing the IOR to file ior.dat");
               Console.WriteLine(e);
            }
            // Activate the POA manager
            rootPOA.ThePOAManager.Activate();

            Console.WriteLine("{0} is ready.", objRef);

            // Wait for incoming requests
            orb.Run();
        }
        catch(Exception e) {
            Console.WriteLine(e);
        }
        Console.ReadLine();
    }
}
```

Adapter activators

Adapter activators are associated with POAs and provide the ability to create child POAs on-demand. This can be done during the `FindPOA` operation, or when a request is received that names a specific child POA.

An adapter activator supplies a POA with the ability to create child POAs on demand, as a side-effect of receiving a request that names the child POA (or one of its children), or when `FindPOA` is called with an activate parameter value of `true`. A server that creates all its needed POAs at the beginning of execution does not need to use or provide an adapter activator; it is necessary only for the case in which POAs need to be created during request processing.

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA (or any descendant POAs) will be queued. This serialization allows the adapter activator to complete any initialization of the new POA before requests are delivered to that POA.

Processing requests

Requests contain the Object ID of the target object and the POA that created the target object reference. When a client sends a request, the ORB first locates the appropriate server, or starts the server if needed. It then locates the appropriate POA within that server.

Once the ORB has located the appropriate POA, it delivers the request to that POA. How the request is processed at that point depends on the policies of the POA and the object's activation state. For information about object activation states, see "Activating objects".

– If the POA has `ServantRetentionPolicy.RETAIN`, the POA looks at the Active Object Map to locate a Servant associated with the Object ID from the request. If a Servant exists, the POA invokes the appropriate method on the Servant.

– If the POA has `ServantRetentionPolicy.NON_RETAIN` or has `ServantRetentionPolicy.RETAIN` but did not find the appropriate Servant, the following may take place:

– If the POA has `RequestProcessingPolicy.USE_DEFAULT_SERVANT`, the POA invokes the appropriate method on the default Servant.

– If the POA has `RequestProcessingPolicy.USE_SERVANT_MANAGER`, the POA invokes `Incarnate` or `Preinvoke` on the Servant Manager.

– If the POA has `RequestProcessingPolicy.USE_OBJECT_MAP_ONLY`, an exception is raised.

If a Servant Manager has been invoked but can not incarnate the object, the Servant Manager can raise a `ForwardRequest` exception.

11

Using the Transaction service

This chapter describes how to use transactions with VisiBroker for .NET. For more
details on each of the APIs see the Borland VisiBroker for .NET API documentation.

Configuring VisiBroker for .NET for transactions

To run with transactions, you must do the following steps:

1   Add a Reference to the Services DLL in your application. This is also required in
    order to access the `CosTransations` namespace, which is defined in that DLL.

2   When executing your application set the `janeva.transactions` property to `true`.

Creating VisiBroker for .NET-managed transactions

With VisiBroker for .NET-managed transactions you are using the `Current` interface for
all transaction management. You are beginning transactions using `Current` and you are
using `Current` for the implicit transaction propagation. This means that you will always
originate your transactions using `Current.Begin()`.

`Current` is an object that is valid for the entire process and manages the association of
each thread's transaction context. Each thread has its own independent, isolated
association with a transaction context.

In VisiBroker for .NET-managed transactions, transaction participants share the same
transaction context because the transaction service transparently forwards the
transaction context to each participant. This means that the state of a transaction is
maintained as the originator calls on other objects to perform actions, which may in turn
call other objects.

Obtaining a Current object reference

To gain access to a VisiBroker for .NET-managed transaction, you must obtain an
object reference to the `Current` object. The `Current` object reference is valid throughout
the process. The following steps describe the general process for obtaining a reference
to a `Current` object, and are include code examples.

1   Call the `orb.ResolveInitialReferences()` method. This method obtains a reference to
    the `Current` object.

2 Narrow the returned object to a `CosTransactions.Current` object.

For example:

```
CORBA.ORB orb = ...;
CosTransactions.Current current = CosTransactions.CurrentHelper.Narrow(
    orb.ResolveInitialReferences("TransactionCurrent"));
```

When you narrow to `CosTransactions.Current`, you specify your use of the original set of methods provided by the `CosTransactions` module.

Looking at the CosTransactions module

The CosTransactions module is the Transaction Service IDL that conforms to the final OMG Transaction Service document. This is the module to use to restrict yourself strictly to CORBA-compliant methods. The IDL for this module is contained in the file CosTransactions.idl.

Transaction service classes and interfaces

Current interface

The Current interface defines methods to:

– Enable a program to manage transactions.

– Use implicit transaction propagation.

– Obtain information about the current transaction.

– Register Resources and Synchronization objects.

Current methods

The following sections describe the important Current methods. For more details, see the Borland VisiBroker for .NET API documentation.

Begin

This method creates a new transaction. Because nested transactions are not supported, this is always a top-level transaction.

The transaction context of the client thread is modified so that the thread is associated with the new transaction. If the client thread is already associated with a transaction, the `CosTransactions.SubtransactionsUnavailable` exception is raised.

Commit

This method commits the transaction associated with the client thread. The effect of this method is equivalent to calling the `Commit` method on the corresponding Terminator object.

If this transaction has been marked for rollback, or any `Resource` votes for Rollback, this call raises `CORBA.TRANSACTION_ROLLEDBACK`. If there is no current transaction, a `CosTransactions.NoTransaction` exception is raised. If the caller is not the transaction originator, `Commit` raises the exception `CORBA.NO_PERMISSION`.

Checks are made to ensure checked behavior.

On return from this method, the client thread is no longer associated with a transaction. Any attempt to use `Current`, as if there were a transaction, will raise an exception, such as `NoTransaction` or `CORBA.TRANSACTION_REQUIRED`, or will return a null object reference.

This method does not return until the transaction is complete, and all related `Synchronization` objects have been notified.

GetControl

This method returns a `Control` object reference that represents the transaction context currently associated with the client thread.

If the client thread is not associated with a transaction, a null object reference is returned.

GetStatus

This method returns an enumerated value (`enum Status`) that represents the status of the transaction associated with the client thread.

Calling this method is equivalent to calling the `GetStatus` method on the corresponding `Coordinator` object. If there is no transaction associated with the current thread, then the method returns `CosTransactions.StatusNoTransaction`.

The possible return values are:

Return value	Description
StatusActive	A transaction is associated with the target object and it is in the active state. The transaction service returns this status after a transaction has been started and prior to a `Coordinator` issuing any prepare statements, unless the transaction has been marked for rollback or timed out.
StatusMarkedRollback	A transaction is associated with the target object and has been marked for rollback, perhaps as the result of the `RollbackOnly` method.
StatusPrepared	A transaction is associated with the target object and has been prepared.
StatusCommitted	A transaction is associated with the target object and has been committed. It is likely that heuristics exist, otherwise the transaction would have been quickly destroyed and `StatusNoTransaction` returned.
StatusRolledBack	A transaction is associated with the target object and the outcome has been determined as rollback. It is likely that heuristics exist, otherwise the transaction would have been quickly destroyed and `StatusNoTransaction` returned.
StatusUnknown	A transaction is associated with the target object, but the transaction service cannot determine its current status. This is a transient condition, and a subsequent invocation will ultimately return a different status.
StatusNoTransaction	No transaction is currently associated with the target object. This will occur after a transaction has completed.
StatusPreparing	A transaction is associated with the target object and it is in the process of preparing. The transaction service returns this status if the transaction has started preparing, but has not yet completed the process, perhaps because it is waiting for responses to prepare from one or more `Resources`.
StatusCommitting	A transaction is associated with the target object and is in the process of committing. The transaction service returns this status if the transaction has begun to commit, but has not yet completed the process, perhaps because it is waiting for responses from one or more `Resources`.
StatusRollingBack	A transaction is associated with the target object and it is in the process of rolling back. The transaction service returns this status if the transaction is being rolled back, but has not yet completed the process, perhaps because it is waiting for responses from one or more `Resources`.

GetTransactionName

This method returns a printable string that is a descriptive name for the transaction. This method is intended to assist in diagnostics and debugging.

The effect of this method is equivalent to calling the `GetTransactionName` method on the corresponding `Coordinator` object. If there is no transaction associated with the client thread, an empty string is returned.

Resume

Associates the client thread with the specified transaction. Typically, this is used to either

– Associate a transaction context with a thread for use in implicit transaction propagation, or

– Resume a transaction that was previously suspended by a `Suspend` method.

The client thread becomes associated with the specified transaction. If the client thread was already associated with a transaction, the previous transaction context is forgotten. If `Resume` is invoked with a NULL control, no transaction is associated with the current thread, and the transaction context is forgotten.

Caution

Any transaction context you set via `Resume` is propagated back to the invoking object.

Rollback

Rolls back the transaction associated with the client thread. This is equivalent to calling the `Rollback` method on the corresponding `Terminator` object. This method does not return until the transaction is complete, and all related `Synchronization` objects have been notified. On return from this method, the client thread is no longer associated with a transaction. Any attempt to use `Current`, as if there were a transaction, will raise an exception, such as `CosTransactions.NoTransaction` or `CORBA.TRANSACTION_REQUIRED`, or return a null object reference. If a heuristic occurs, this method will not throw a heuristic-related exception.

If the caller is not the transaction originator, `Rollback` raises the exception `CORBA.NO_PERMISSION`.

RollbackOnly

The method modifies the transaction associated with the client thread so that Rollback is the only possible transaction outcome. The effect of this request is equivalent to calling the `RollbackOnly` method on the corresponding `Coordinator` object. A client that is restricted from performing the `Rollback` operation, can nonetheless call `RollbackOnly`.

SetTimeout

This method establishes a new time-out for transactions started by subsequent calls to the `Current.Begin` method in all threads within this program.

To establish a new time-out, use these values of the seconds parameter:

Value	Effect
= 0	Sets any subsequent transaction that is begun to the default transaction time-out for the transaction service instance that it uses.
> 0	Sets the new time-out to the specified number of seconds. If the `seconds` parameter exceeds the maximum time-out valid for a transaction service instance being used, then the new time-out is set to that maximum, to bring it in range.

Note:

When a transaction, created by a subsequent call to `Begin` in any thread in the process, takes longer to start transaction completion than the established time-out, it will be rolled back. If the time-out occurs before the transaction enters the completion stage (begins two-phase or one-phase processing) the transaction will be rolled back. Otherwise, the time-out is ignored.

Suspend

This method suspends the transaction currently associated with the client thread and returns a `Control` object for that transaction. If the client thread is not associated with a transaction, a null object reference is returned.

The `Control` object can be passed to the `Resume` method to reestablish this context in the same thread or a different thread.

After the call to `Suspend`, no transaction is associated with the client thread. Any attempt to use `Current`, as if there were a transaction, will raise an exception, such as `CosTransactions.NoTransaction` or `CORBA.TRANSACTION_REQUIRED`, or return a null object reference.

TransactionFactory interface

The TransactionFactory interface defines methods that enable a program to initiate nonVisiBroker for .NET-managed transactions. The TransactionFactory interface gives programs direct control over the propagation of transaction context.

You acquire a TransactionFactory object the way you do any CORBA object; for example, by binding.

TransactionFactory methods

The following sections describe the important TransactionFactory methods. For more details, see the Borland VisiBroker for .NET API documentation.

Create

This method accepts a time-out parameter (`time_out`) and creates a new transaction. It returns a Control object. The Control object can be used to manage or to control participation in the new transaction. The Control object can be used by any thread and passed around explicitly, just like any other CORBA object.

Note

Checked behavior cannot be provided for transactions that use this method.

To establish a new time-out, use the following values of the `time_out` parameter.

Value	Effect
= 0	Sets any subsequent transaction that is begun to the default transaction time-out for the transaction service instance that it uses.
> 0	Sets the new time-out to the specified number of seconds. If the `seconds` parameter exceeds the maximum time-out valid for a transaction service instance being used, then the new time-out is set to that maximum, to bring it in range.

The new time-out applies only to the transaction created on this call. If a transaction does not start transaction completion (begin two-phase or one-phase processing) before the time-out expires, it will be rolled back.

Recreate

This method creates a new Control object using its PropagationContext parameter. The Control object can be used to manage or to control participation in the transaction. Applications will not normally call this method.

To get a transaction's PropagationContext, invoke the `CosTransactions.CoordinatorOperations.GetTxcontext` method on the transaction's Coordinator object.

Control interface

The Control interface enables a program to explicitly manage or propagate a transaction context. A Control object is implicitly associated with one specific transaction.

The Control interface defines two methods: `GetCoordinator` and `GetTerminator`. The `GetCoordinator` method returns a Coordinator object, which supports methods used by participants in the transaction. The `GetTerminator` method returns a Terminator object, which supports methods to complete the transaction. The Terminator and Coordinator objects support methods that are typically performed by different parties. Providing two objects enables each set of methods to be made available only to the parties that require those methods.

You can obtain a Control object by using one of the methods of the TransactionFactory (see "TransactionFactory interface"). You can also obtain a Control object for the current transaction (associated with a thread) through methods of the Current object. See descriptions of the `GetControl` or `Suspend` methods in "Current interface".

Control methods

The following sections describe the important Control methods. For more details, see the Borland VisiBroker for .NET API documentation.

GetCoordinator

This method returns a Coordinator object. The Coordinator provides methods that are called by participants in a transaction. These participants are typically either recoverable objects or agents of recoverable objects.

GetTerminator

This method returns a Terminator object. The Terminator can be used to rollback or commit the transaction associated with the Control. The `CosTransactions.Unavailable` exception is raised if the Control cannot provide the requested object due to the inability of the Terminator object to be transmitted to or be used in other execution environments.

Terminator interface

The Terminator interface supports methods to commit or roll back a transaction. Typically, these methods are used by the transaction originator, but any program that has access to a Terminator object for that transaction can commit or roll back the transaction.

Terminator methods

The following sections describe the important Terminator methods. For more details, see the Borland VisiBroker for .NET API documentation.

Commit

Before committing the transaction, this method performs some checks. If the transaction has not been marked rollback only, and all of the participants in the transaction agree to commit, the transaction is committed and the operation terminates normally. Otherwise, the transaction is rolled back and the `CORBA.TRANSACTION_ROLLEDBACK` standard exception is raised.

If the `report_heuristics` parameter is true, the Transaction Service will report inconsistent or possibly inconsistent outcomes using the `CosTransactions.HeuristicMixed` and `CosTransactions.HeuristicHazard` exceptions when appropriate. Information about the Resources involved in a heuristic outcome will be written to a heuristic log file corresponding to the instance of the Transaction Service.

When a transaction is committed, all changes to recoverable objects made in the scope of this transaction are made permanent and visible to other transactions or clients.

Rollback

This method rolls back the transaction. When a transaction is rolled back, all changes to recoverable objects made in the scope of this transaction are rolled back. All Resources locked by the transaction are made available to other transactions as appropriate to the degree of isolation enforced by the Resources.

This method does not return until the transaction is complete and all related Synchronization objects have been notified.

Coordinator interface

The Coordinator interface provides methods that are used by participants in a transaction. These participants are typically either recoverable objects or agents of recoverable objects. Each Coordinator is implicitly associated with a single transaction.

Several of the Coordinator methods are equivalent, that is, they return the same result.

– `GetStatus`

– `GetTopLevelStatus`

– `GetParentStatus`

Similarly, certain methods return `TRUE` only when the target object and the parameter refer to the same Coordinator object. Therefore, the following methods are also equivalent:

– `IsSameTransaction`

– `IsRelatedTransaction`

– `IsAncestorTransaction`

– `IsDescendantTransaction`

And, the following methods are equivalent:

– `HashTransaction`

– `HashTopLevelTran`

Coordinator methods

The following sections describe the important Coordinator methods. For more details, see the Borland VisiBroker for .NET API documentation.

GetStatus

This method returns the status of the transaction associated with the target object, as an enumerated value (`enum Status`). If there is no transaction associated with the target object, then the method returns the value `StatusNoTransaction`.

Because VisiBroker for .NET does not support nested transactions, the `GetStatus`, `GetTopLevelStatus` and `GetParentStatus` methods return the same result.

The following are the possible return values, as defined in CosTransactions.idl:

StatusActive	StatusUnknown
StatusMarkedRollback	StatusNoTransaction
StatusPrepared	StatusPreparing
StatusCommitted	StatusCommitting
StatusRolledBack	StatusRollingBack

For information about each Status value, see "GetStatus".

GetTransactionName

This method returns a printable string that is a descriptive name for the transaction. This method is intended to assist with diagnostics and debugging. If there is no transaction associated with the client thread, an empty string is returned.

GetTxcontext

The `GetTxcontext` method returns a PropagationContext, which can be used by one Transaction Service domain to export a transaction to a new Transaction Service domain.

HashTransaction

This method returns a hash code for the transaction associated with the target object. Each transaction has a single hash code. The hash code can be used to efficiently compare Coordinators for inequality against the hash codes of other transactions. If the hash codes of two Coordinators are not equal, then they represent different transactions. If two hash codes are equal, then `IsSameTransaction` must be used to guarantee equality or inequality, because two Coordinators might have the same hash code but, in fact, represent two different transactions.

IsSameTransaction

This method returns true if, and only if, the target object and the parameter object both refer to the same transaction.

RegisterResource

This method registers the specified Resource as a participant in the transaction associated with the target object. When the transaction is terminated, the Resource will receive requests to prepare, commit, or rollback the updates performed as part of the transaction. For information on Resource methods, see "Resource interface".

This method returns a RecoveryCoordinator that can be used by this Resource during recovery.

RegisterSynchronization

This method registers the specified Synchronization object so that it will be notified to perform the necessary processing before and after completion of the transaction. Such methods are described in the description of the Synchronization interface; see "Synchronization interface".

RegisterSubtranAware

Because VisiBroker for .NET does not support nested transactions, this method always raises `CosTransactions.SubtransactionsUnavailable`.

RollbackOnly

This method modifies the transaction associated with the Coordinator so that rollback is the only possible transaction outcome.

RecoveryCoordinator interface

When a Resource is registered with the Coordinator, a RecoveryCoordinator is returned. The RecoveryCoordinator is implicitly associated with a single Resource registration request and can only be used by that Resource. In case recovery is necessary, the Resource can use the RecoveryCoordinator during the recovery process.

Also, the Resource can use the RecoveryCoordinator if it needs to know the current status of the transaction. For example, the Resource can set its own time-out, and if commit or rollback does not occur within the time-out, the Resource can invoke `ReplayCompletion` to determine the status of the transaction.

RecoveryCoordinator methods

The following section describes the RecoveryCoordinator methods. For more details, see the Borland VisiBroker for .NET API documentation.

ReplayCompletion

This method notifies the Transaction Service that the Resource is available. This method is typically used during recovery, and can be used by the Resource to determine the status of the transaction.

Note

This method does not initiate completion.

Resource interface

VisiBroker for .NET uses a two-phase commit protocol to complete a top-level transaction with each Resource registered with it, that is, with each Resource that might change during the transaction. The Resource interface defines the methods invoked by the Transaction Service on each Resource. Each object supporting the Resource interface is implicitly associated with a single top-level transaction.

VisiBroker for .NET provides the Resource interface in the CosTransactions.idl file, but you must provide the implementation in your Resource. A typical application does not implement a Resource.

Resource methods

The following sections describe the important Resource methods. For more details, see the Borland VisiBroker for .NET API documentation.

Commit

This method attempts to commit all changes associated with the Resource. If a heuristic outcome exception is raised, the Resource must keep the heuristic decision in persistent storage until the `Forget` method is performed so that it can return the same outcome in case `Commit` is invoked again during recovery. Otherwise, the Resource can immediately forget all knowledge of the transaction.

CommitOnePhase

This method requests the Resource to commit all changes made as part of the transaction. This method is an optimization for use when a transaction has only one participating Resource. This method can be called on the Resource, instead of first calling `Prepare` and then `Commit` or `Rollback`.

If a heuristic outcome exception is raised, the Resource must keep the heuristic decision in persistent storage until the `Forget` method is performed. This enables the Resource to return the same outcome in case `CommitOnePhase` is performed again during recovery. Otherwise, the Resource immediately forgets all knowledge of the transaction.

If a failure occurs during `CommitOnePhase`, it is called again when the failure is repaired. Since there is only a single Resource, the `HeuristicHazard` exception is used to report heuristic decisions related to that Resource.

Forget

When VisiBroker for .NET receives a heuristic exception, it records the exception. The Transaction Service will ultimately call `Forget` on the Resource. This means that the Resource can discard all information about the transaction that raised the heuristic exception. This method is called only if a heuristic exception was raised from `Rollback`, `Commit`, or `CommitOnePhase`.

Prepare

This method performs the prepare operation, the first step in the two-phase commit protocol for a Resource object. When finished, the method returns one of these Vote values.

– VoteReadOnly—No persistent data associated with the Resource has been modified by the transaction.

– VoteCommit—The following data has been saved to persistent storage:

  – All data changed as part of the transaction

  – A reference to the RecoveryCoordinator object

  – An indication that the Resource has been prepared

– VoteRollback—Some circumstance has caused the Resource to call for a rollback, such as inability to save the relevant data, inconsistent outcomes, or no knowledge of the transaction (which might happen after a crash).

After returning VoteReadOnly or VoteRollback, the Resource can forget all knowledge of the transaction.

If a heuristic outcome exception is raised, the Resource must save the heuristic decision in persistent storage until the `Forget` method is called so that it can return the same outcome in case `Prepare` is called again.

Rollback

This method rolls back all updates associated with the Resource object.

If a heuristic outcome exception is raised, the Resource must save the heuristic decision in persistent storage until the `Forget` method is invoked. This enables the Resource to return the same outcome in case `Rollback` is called again during recovery. Otherwise, the Resource immediately forgets all knowledge of the transaction.

Synchronization interface

The Synchronization interface defines methods that enable a transactional object to be notified before the start of the two and one-phase commit protocol, and after its completion. In the CosTransactions module, the Synchronization interface provides two methods:

– `BeforeCompletion`—Ensures that `BeforeCompletion` is invoked before starting to commit a transaction.

– `AfterCompletion`—Ensures a transactional object is notified after the transaction has been completed. This applies to all transactions whether they were committed or rolled back.

Here are two limitations you should be aware of:

– If the Transaction Service cannot contact your Synchronization object while trying to call `BeforeCompletion`, then the transaction will be rolled back. If a Synchronization object is unavailable after completion, it will be ignored.

– When the Transaction Service instance recovers from a failure, it does not remember Synchronization objects, and will only replay completion and not Synchronization objects. If a failure occurs, the Synchronization object will not be notified of how the transaction was completed by the VisiTransact Transaction Service.

Note

In certain cases, `AfterCompletion` is called when `BeforeCompletion` was not called. `BeforeCompletion` is called only if a transaction is still continuing towards a commit at the outset of completion. `AfterCompletion` is always called (unless the Transaction Service crashes before the transaction completes).

Synchronization objects are not recoverable. If an instance of a Transaction Service fails, any transactions that are completed will not involve Synchronization objects.

Note

Although the signatures of these methods are fixed by the Synchronization interface, their implementations are user-defined. This enables an application to do custom processing at key points in a transaction, before and after transaction completion.

Synchronization methods

The following sections describe the important Synchronization methods. For more details, see the Borland VisiBroker for .NET API documentation.

AfterCompletion

This is a method that you write that performs customized processing after the completion of the transaction. It is essentially a callback.

Note

The `AfterCompletion` method is always invoked during normal processing.

IDL for the Synchronization interface inherits from the TransactionalObject interface. As a programmer, you are responsible for writing the implementation of an `AfterCompletion` method that conforms to the IDL.

If `AfterCompletion` is to be called in processing a particular transaction, the following actions must be taken:

1 A Synchronization object must be created by the transaction originator or some other transaction participant.

2 The Synchronization object must be registered by getting the transaction's Coordinator, and calling the `RegisterSynchronization` method in Coordinator and Current. See the description for the `RegisterSynchronization` method in "Coordinator interface". Registration must be done after the transaction is created and before the start of the two-phase commit.

Multiple Synchronization objects can be created and registered for a single transaction.

The Transaction Service calls this method after the two-phase commit protocol completes. As an example of its use, `AfterCompletion` can be used by a transactional object to discover the outcome of the transaction. This is particularly useful for transactional objects that are not also recoverable objects, and so are not automatically notified of the outcome.

You can call `GetStatus` to see whether or not the transaction has been marked for rollback.

Notice that because Synchronization inherits from TransactionalObject, the transaction context will be available through the Current object.

All exceptions will be ignored.

BeforeCompletion

This is a method that you write to perform customized processing at the onset of the completion of a transaction. It is called only if the transaction is still continuing towards successful completion. It is essentially a callback.

Note:

The `BeforeCompletion` method is invoked after the application invokes commit, but before the Transaction Service begins transaction completion. The `BeforeCompletion` method is not invoked for a rollback request.

The IDL for the Synchronization interface inherits from the TransactionalObject interface. As a programmer, you are responsible for writing the implementation of a `BeforeCompletion` method that conforms to the IDL.

If `BeforeCompletion` is to be called when processing a particular transaction, the Synchronization object must be registered using the `RegisterSynchronization` method in the Coordinator interface. Register the Synchronization object from your transactional object or recoverable server. See the description for the

`RegisterSynchronization` method in "Coordinator interface". Registration must be done after the transaction is created and before the start of the two-phase commit.

Multiple Synchronization objects can be created and registered for a single transaction.

The Transaction Service calls this method after the transaction work has been done but before the two-phase commit protocol starts; that is, before `Prepare` is called on the participating Resource. The transaction service calls `BeforeCompletion` only if a transaction is still continuing towards a commit at the outset of completion. This means that `Terminator.Commit` was called and the transaction has not been marked for rollback. If `Terminator.Rollback` was called, or the first of several Synchronization objects marked the transaction for rollback, or the transaction was already marked for rollback, `BeforeCompletion` calls will not be called again for this transaction.

Within this method, you can ensure the transaction will be rolled back by calling the `RollbackOnly` method. You can also call `GetStatus` to see whether or not the transaction has been marked for rollback. At the time the method is called, however, you cannot rely upon the status to indicate whether or not the transaction will actually be committed.

Notice that because the Synchronization interface inherits from TransactionalObject, the transaction context will be available through the Current object. This means that `BeforeCompletion` can use all objects on the Current object, such as `GetStatus` and `GetControl`.

All CORBA exceptions raised by your Synchronization objects will result in the transaction being rolled back.

TransactionalObject interface

The TransactionalObject interface provides for the automatic propagation of transaction context on method calls of transactional objects. The TransactionalObject interface defines no methods.

Methods that work on transactions must have access to the transaction context. The transaction context can be made available to such methods in two ways:

– Explicit propagation—A method receives and passes the transaction context as a Terminator, Control, Coordinator, or PropagationContext structure.

– Implicit propagation—The transaction context is passed automatically (and implicitly) on method calls.

Implicit propagation is the typical, and easiest, way. This is the capability that the TransactionalObject interface provides to your transactional objects. An instance of TransactionalObject can participate in implicit propagation. Implicit propagation is where the transaction context associated with the client thread is automatically propagated to TransactionalObject instances through method calls.

To use VisiBroker for .NET-managed transactions, all of your transactional objects must inherit from TransactionalObject. By using VisiBroker for .NET-managed transactions, you benefit from checked behavior.

The transaction context is always passed implicitly to an object that inherits from `CosTransactions.TransactionalObject`. In addition, a program may be passed a transaction context explicitly, as a parameter.

12

Using the Security service

As more businesses deploy distributed applications and conduct operations over the Internet, the need for high quality application security has grown.

Sensitive information routinely passes over Internet connections between web browsers and commercial web servers; credit card numbers and bank balances are two examples. For example, users engaging in commerce with a bank over the Internet must be confident that:

- They are in fact communicating with their bank's server, not an impostor that mimics the bank for illegal purposes.

- The data exchanged with the bank will be unintelligible to network eavesdroppers.

- The data exchanged with the bank software will arrive unaltered. An instruction to pay $500 on a bill must not accidentally or maliciously become $5000.

VisiBroker for .NET Security lets the client authenticate the bank's server. The bank's server can also take advantage of the secure connection to authenticate the client. In a traditional application, once a secure connection has been established, the client sends the user's name and password to authenticate. This technique can be used once a VisiBroker for .NET secure connection has been established, with the benefit that the user name and password exchanges will be encrypted.

VisiBroker for .NET Security overview

VisiBroker for .NET Security lets you establish secure connections between clients and servers, and it provides a framework for secure communication. VisiBroker for .NET Security uses the Microsoft Windows Secure Channel (Schannel) library for SSL and TLS (Transport Layer Security) communications and the Microsoft CryptoAPI for cryptographic operations.

VisiBroker for .NET Security features include

- J2EE server and CORBA server interoperability: VisiBroker for .NET Security seamlessly interoperates with EJB security via the underlying CORBA Common Secure Interoperability specification (CSIv2).

- Microsoft Windows Certificate Store integration: VisiBroker for .NET Security uses the Microsoft Windows Certificate Store for public and private key management.

- ASP.NET integration: VisiBroker for .NET Security propagates the security identities authenticated by ASP.NET applications to the J2EE server or CORBA server.

– Secure Transport Layer: VisiBroker for .NET Security utilizes SSL and TLS protocols as a secure transport layer. Both protocols provide message confidentiality, message integrity, and certificate-based authentication support through a trust model.

– Borland GateKeeper integration: VisiBroker for .NET Security supports a secure connection through GateKeeper. For details see "Using VisiBroker for .NET with Borland GateKeeper," and the *VisiBroker GateKeeper Guide*.

Enabling VisiBroker for .NET Security

By default VisiBroker for .NET Security is disabled. To enable the VisiBroker for .NET Security include the <security> section in the configuration file as shown below:

```
<visinet>
   <security enabled="true">
   </security>
</visinet>
```

Alternatively, you can enable security by setting the `janeva.security` property to `true`. See (properties chapter) for instructions.

Interoperating with J2EE servers and CORBA servers

VisiBroker for .NET Security supports two kinds of user authentication:

– User name and password authentication

– Certificate-based authentication

VisiBroker for .NET Security supports the .NET Remoting API, a CORBA-based API, and a configuration file method of setting up the security identity. These methods are described in each of the following sections.

User name and password authentication

If the J2EE server or CORBA server requires user authentication, VisiBroker for .NET Security provides multiple ways to set up the user credentials and pass them to the server side. User name and password authentication lets a VisiBroker for .NET client authenticate users by passing a user name and password to the server. You can implement user name and password authentication in one of the following ways: the .NET Remoting API, the CORBA-based API, or the application configuration file.

Using the .NET Remoting API for user name and password authentication

The following example shows you how to use the .NET Remoting API to do user name and password authentication.

The first step is to resolve the Remoting proxy reference:

```
// creating the CartHomeRemotingProxy configured as
// a well-known remoting object in the config file
CartHome home = new CartHomeRemotingProxy();
```

The next step is to resolve the Sink Properties of this Remoting proxy object:

```
// setup security credentials
IDictionary props =
   System.Runtime.Remoting.Channels.ChannelServices.GetChannelSinkProperties(
      home);
```

Next, the application sets the user name and password properties:

```
props["username"] = "joeshopper";
props["password"] = "joepass";
```

Optionally, you can also set the realm.

```
props["realm"] = "myuprealm";
```

In the absence of the realm property, the realm defaults to `default`.

Note

Different application servers might have different names for the default realm. You can set the default realm name in the configuration file. When you need to override the default realm set in the configuration file you can set the property in the command line or programmatically as shown above. See "Configuring properties" for more information.

Once the properties are set you can invoke methods on the Remoting proxy. The user name, password, and realm are passed transparently to the server side as a part of the invocation context.

```
// creating a new instance of Cart session
Cart cart = home.Create(...);
```

Note that it is important to set the credentials before the first invocation on the Remoting proxy, otherwise the credentials will not be passed to the server.

Keep in mind that every object on the same server shares the same secure connection. Once the first invocation is completed, any subsequent invocation on the same or other objects located on the same server shares the credentials established with the first invocation. To change the credentials resolve the Sink Properties and set the `username` and `password` properties again.

In the example below, you do not need to set up the credentials again for the cart object. The cart uses the same credentials established by the secure connection to the home object.

```
// adding a new book into the cart
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

More example code is located in the <janeva_install_dir>\examples\Advanced\ Security\RemotingUsernameClient directory.

Using the CORBA-based API for user name and password authentication

The following example shows how to use the CORBA-based API to establish the user credentials.

The first step is to resolve the VisiBroker for .NET security context on the orb instance. `Janeva.Security.Context` is the object which exposes the API with which you manipulate the user's identity.

```
// initialize the ORB
CORBA.ORB orb = CORBA.ORB.Init(args);

// resolve the Securuty Context
Janeva.Security.Context context =
    (Janeva.Security.Context) orb.ResolveInitialReferences("SecurityContext");
```

Next, to set the user name, password and realm use the `Janeva.Security.IdentityWallet` class as follows:

```
// create a wallet with the credentials
Janeva.Security.IdentityWallet wallet = new Janeva.Security.IdentityWallet(
    "joeshopper", "joepass".ToCharArray(), "myuprealm");
```

In the absence of the realm property, the realm defaults to `default`.

Note

Different application servers might have different names for the default realm. You can set the default realm name in the configuration file. When you need to override the default realm set in the configuration file you can set the property in command line

or programmatically as shown above. See "Configuring properties" for more information.

The last step is to call a `Login` method on the security context with the wallet:

```
// login in to the security conext with the wallet
context.Login(wallet);
```

The `Janeva.Security.Context` object provides different login methods. Please see the VisiBroker for .NET API reference for details.

Keep in mind that every object on the same server shares the same secure connection. Once the first invocation is completed, any subsequent invocation on the same or other objects located on the same server shares the credentials established with the first invocation. To change the credentials call `Logout` and then call `Login` again.

Once you set the credentials with the `Login` method you can invoke methods on the sever:

```
// creating a new instance of Cart session
Cart cart = home.Create(...);
```

Once you login to the `Janeva.Security.Context` the same credentials are used with any subsequent remote invocation:

```
// adding a new book into the cart
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

Using a configuration file for user name and password authentication

The following is an example of how to set security credentials using a configuration file.

```
<configuration>
   <visinet>
      <security enabled="true">
         <identity>
            <username>joeshopper</username>
            <password>joepass</password>
            <realm>myuprealm</realm>
         </identity>
      </security>
   </visinet>
</configuration>
```

Setting the identity in the configuration file has a global effect on the application. The same identity is used for each remote invocation.

Certificate-based authentication

VisiBroker for .NET certificate support is based on the Microsoft Windows Certificate Store. Before the certificate can be used with a VisiBroker for .NET application it needs to be imported into the Certificate Store. Refer to the Microsoft documentation for more information on how to issue and manage certificates.

One of the optional certificate attributes is a *friendly name*. VisiBroker for .NET uses the certificate's friendly name as the identifier to address a particular certificate. If a certificate does not have a friendly name you can set it in the Microsoft Windows Internet Options control panel.

Note:

When the certificate is used to authenticate the client, it is important that the certificate have both public and private keys in it. This is the requirement of the SSL/TLS protocol.

Using the .NET Remoting API for certificate-based authentication

The following example shows you how to use the .NET Remoting API to achieve certificate-based authentication.

The first step is to resolve the Remoting proxy reference:

```
// creating the CartHomeRemotingProxy configured as
// a well-known remoting object in the config file
CartHome home = new CartHomeRemotingProxy();
```

The next step is to resolve the SinkProperties of this Remoting proxy object:

```
// setup security credentials
IDictionary props =
    System.Runtime.Remoting.Channels.ChannelServices.GetChannelSinkProperties(
        home);
```

Next, set the certificate's friendly name:

```
props["certificate"] = "joeshopper";
```

Note:

Instead of using the friendly name, you can also specify an asterisk (*) to let VisiBroker for .NET decide which certificate to use.

Once the property is set you can invoke methods on the Remoting proxy. The certificate is passed transparently to the server side as a part of the invocation context.

```
// creating a new instance of Cart session
Cart cart = home.Create(...);
```

Note that it is important to set the credentials before the first invocation on the Remoting proxy, otherwise the credentials will not be passed to the server.

Keep in mind that every object on the same server shares the same secure connection. Once the first invocation is completed, any subsequent invocation on the same or other objects located on the same server shares the credentials established with the first invocation. To change the credentials resolve the Sink Properties and set the `username` and password `properties` again.

In the example code below, you do not need to set up the credentials again for the cart object. The cart uses the same credentials established by the secure connection to the home object.

```
// adding a new book into the cart
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

More example code is located in the <janeva_install_dir>\examples\Advanced\Security\RemotingCertificateClient directory.

Using the CORBA-based API for certificate-based authentication

The following example shows how to use the CORBA-based API to achieve certificate-based authentication.

The first step is to resolve the VisiBroker for .NET security context on the orb instance. The `Janeva.Security.Context` is the object which exposes the API with which you manipulate the user's identity.

```
// initialize the ORB
CORBA.ORB orb = CORBA.ORB.Init(args);

// resolve the Securuty Context
Janeva.Security.Context context =
    (Janeva.Security.Context) orb.ResolveInitialReferences("SecurityContext");
```

Next, to set the certificate friendly name you need to use the `Janeva.Security.CertificateWallet` class as follows:

```
// create a wallet with the credentials
Janeva.Security.CertificateWallet wallet = new
    Janeva.Security.CertificateWallet("joeshopper",
        CertificateWallet.CLIENT_AUTHENTICATION);
```

The second parameter defines the certificate usage. This parameter is set differently when used for secure server (see "Enabling security for .NET servers"). For details on the other values for this parameter see the VisiBroker for .NET API reference.

Note:

Instead of using the friendly name, you can also specify an asterisk (*) to let VisiBroker for .NET decide which certificate to use.

Last step is to call a `Login` method on the security context with the wallet:

```
// login in to the security conext with the wallet
context.Login(wallet);
```

The `Janeva.Security.Context` object provides different login methods. See the VisiBroker for .NET API reference for details.

Keep in mind that every object on the same server shares the same secure connection. Once the first invocation is completed, any subsequent invocation on the same or other objects located on the same server shares the credentials established with the first invocation. To change the credentials call `Logout` and then call `Login` again.

Once you set the credentials with the `Login` method you can invoke methods on the object:

```
// creating a new instance of Cart session
Cart cart = home.Create(...);
```

Once you login to the `Janeva.Security.Context` the same credentials are used with any subsequent remote invocation.

```
// adding a new book into the cart
Item book = new Book();
book.Title = "War and Peace";
book.Price = 20.99f;
cart.AddItem(book);
```

Using a configuration file for certificate-based authentication

The following is an example of how to specify the certificate in a configuration file.

```
<configuration>
    <visinet>
        <security enabled="true">
            <identity>
                <certificate>joeshopper</certificate>
            </identity>
        </security>
    </visinet>
</configuration>
```

Setting the certificate in the configuration file has a global effect on the application. The identity presented by the certificate is used for each remote invocation.

ASP.NET integration

The VisiBroker for .NET Security integration with ASP.NET is based on the concept of identity assertion. Whenever the VisiBroker for .NET runtime on ASP.NET makes an outgoing call, it will propagate two identities to the called server. It will identify itself to the called server and assert the caller's identity.

The caller's identity is the identity that the browser or other clients use to communicate with the ASP.NET tier. When VisiBroker for .NET asserts this identity as the caller

identity, it is, in fact, asserting to the called server that this tier trusts that this caller is authenticated and it is performing this request on behalf of this caller.

It is up to the called server to decide whether it will accept this assertion from this tier. Since the ASP.NET tier identifies itself, it allows the called server to authenticate and decide whether this tier has the privileges to make this assertion.

Note

Some servers may need explicit configuration that defines which peer identities (in this case, the identity of the ASP.NET tier) it will accept assertions from. Refer to the server's documentation for more details.

ASP.NET configuration

Within an ASP.NET environment, VisiBroker for .NET implicitly detects whether a user is authenticated and passes the user identity as the caller identity to the server side. The peer identity is established explicitly using the VisiBroker for .NET Security API as shown in the examples in "Interoperating with J2EE servers and CORBA servers".

The following is an example of how a configuration file can establish the peer identity for an ASP.NET application. Note that this example is similar to the example in "Using a configuration file for certificate-based authentication".

```
<configuration>
   <visinet>
      <security enabled="true">
         <identity>
            <username>peer</username>
            <password>pwd</password>
       ...
</configuration>
```

Note

You can also explicitly set the caller identity with the `Janeva.Security.Context.ImportIdentity()` API. This allows you to use the trust model outside of the ASP.NET environment. See the VisiBroker for .NET API reference for details about `Janeva.Security.Context.ImportIdentity`.

More example code is located in the <janeva_install_dir>\examples\Advanced\ Security\AspNetClient directory.

Enabling security for .NET servers

For secure .NET server applications VisiBroker for .NET Security can be enabled on the server side by setting the `janeva.security.server` property to `true`. The following is an example of how to set the property in the application configuration file.

```
<configuration>
   <visinet>
      <security>
         <server enabled="true" defaultPort="15000">
            <certificate>cert_friendly_name</certificate>
         </server>
       ...
```

You can set the port which the VisiBroker for .NET server will use for SSL/TSL communication on the server side by setting the `janeva.security.server.defaultPort` property. See the previous example for how this is done in a configuration file.

The server side must be identified with a certificate per SSL/TSL protocol requirement. You can do this using the CORBA-based API or a configuration file property.

The following is an example of how to set the certificate identity in the configuration file:

```
<configuration>
```

```
<visinet>
  <security>
    <server enabled="true" defaultPort="15000">
      <certificate>cert_friendly_name</certificate>
    </server>
  ...
```

The following is an example of how to set the certificate identity using the CORBA-based API.

The first step is to resolve the VisiBroker for .NET security context on the orb instance. The `Janeva.Security.Context` is the object which exposes the API with which you manipulate the identity.

```
Janeva.Security.Context context = (Janeva.Security.Context)
    CORBA.ORB.Init().ResolveInitialReferences("SecurityContext");
```

Next, to set the certificate you need to use the `Janeva.Security.CertificateWallet` class as follows:

```
Janeva.Security.CertificateWallet wallet = new
Janeva.Security.CertificateWallet(
    "joeshopper", Janeva.Security.CertificateWallet.SERVER_AUTHENTICATION);
```

Note that the second parameter defines the certificate usage for the server. For details on the other values for this parameter see the VisiBroker for .NET API reference.

The last step is to call a `Login` method on the security context with the wallet:

```
context.Login(wallet);
```

The `Janeva.Security.Context` object provides different login methods. See the VisiBroker for .NET API reference for details.

Note

When using the CORBA-based API configuration method, the certificate needs to be set up before the server starts listening for incoming requests, that is, before calling the `CORBA.ORB.Run()` method in your code.

More example code is located in the <janeva_install_dir>\examples\Advanced\ Security\SslServer directory.

13

Using VisiBroker for .NET with Partially Trusted Applications

Code access security is a very powerful feature that allows systems to be configured to execute partially trusted code without prompting the user.This is, in fact, the default setting. Meanwhile, partially trusted code is only allowed to do things appropriate to its level of trust.

The level of trust applied to a body of code depends on various pieces of evidence that are provided to the security policy engine at runtime. Evidence is provided at the granularity of an assembly. There are many kinds of evidence. Some evidence is provided by the hosting CLR environment such as the source of the assembly, the 'Zone' to which that source belongs (much like Internet Explorer), and some evidence is provided by the assembly itself, such as its Public Key Token. Based on the evidence associated with an assembly, the assemblies are assigned to code groups by the policy engine. Each code group can have a membership condition (such as 'Assembly must be from the Intranet zone') and an associated set of permissions.

Read the following documents to get a basic familiarity with Partially Trusted Applications.

- .NET documentation on the MSDN web site for a detailed introduction to Code Access Security (http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcodeaccesssecurity.asp).

- An article that describes Partially Trusted Code and Partially Trusted Environments located at http://msdn.microsoft.com/msdnmag/issues/02/06/Rich/default.aspx.

Using VisiBroker for .NET in Partially Trusted Environments

To use VisiBroker for .NET in partially trusted environments, VisiBroker for .NET should be installed locally. Alternatively, you may also configure your security policies such that VisiBroker for .NET DLLs are given full trust. This is due to the fact that VisiBroker for .NET uses other assemblies (such as Visual J#) that are not usable directly by partially trusted assemblies (PTAs)

Note

To be used by partially trusted code, an assembly must have the AllowPartiallyTrustedCallersAttribute (APTCA) applied to it.

VisiBroker for .NET's public assemblies are marked with the APTCA to allow partially trusted callers to load and call into it. However, VisiBroker for .NET itself requires full trust to perform its functions.

Once VisiBroker for .NET is locally installed, partially trusted applications can load VisiBroker for .NET and call methods on it. However, to communicate with remote servers using VisiBroker for .NET, changes to the local security policy are required to give appropriate permissions to the partially trusted assemblies.

For example, partially trusted assemblies do not have access to network. Specifically, they do not have the ability to open sockets to arbitrary servers. The only exception is for assemblies that are loaded from the 'LocalIntranet' zone. These are allowed to connect back to the site they came from using the protocol they were downloaded with (or a protocol that is more secure). This typically means that an assembly that is downloaded using http (which is the most common scenario) is allowed to have http (or https) access back to the host it came from.

When using VisiBroker for .NET, however, the assembly will communicate using IIOP (and TCP/UDP sockets) and the default security policy does not give the PTA, the permission to use sockets. VisiBroker for .NET will NOT assert that permission as that will constitute a security hole. If VisiBroker for .NET asserts that permission, then any partially trusted application can access the remote host it came from, without restrictions, and this is not recommended from a security perspective.

To enable VisiBroker for .NET clients to communicate remotely, make sure that such applications *do* have the socket permission. The criteria for determining which particular application (or PTA) has socket permissions to connect to a given host or not is dependent on user requirements. Users must configure the Code Access Security policies of the machine that is executing the application such that the appropriate permissions are given to the application. There are a few choices provided by the Microsoft security framework. In addition, you may use custom code groups to fine tune the security policy.

Permissions Required by VisiBroker for .NET

VisiBroker for .NET requires partially trusted application code the following permissions to execute correctly. Note that the application may require other permissions depending on what it actually does. For example, it may need UI permissions if it is launching windows.

`SecurityPermission.Execute`: This permission is not required by VisiBroker for .NET per-se but is required by any assembly that is partially trusted to load and execute

`DnsPermission.Unrestricted`: The ability to resolve DNS names. VisiBroker for .NET will not assert this permission. Otherwise, the code could use VisiBroker for .NET to probe for valid hosts. The PTA must have the ability to do DNS resolves.

`SocketPermission`: Code that uses VisiBroker for .NET is invariably calling into remote servers. So, the PTA requires permissions to open sockets to (or allow connections from, in the case of callbacks) the appropriate server host and port. VisiBroker for .NET will not assert permissions even back to the same host. This is because doing so, will allow arbitrary untrusted code to connect back to its host. Note here that socket permissions given to the application must allow the client to connect to the remote server that VisiBroker for .NET code is accessing. This may not be the same as the host that is serving the application assemblies.

See the documentation for Security Policy configuration and the caspol tool for more details on how one can configure security policy.

Usage in No Touch Deployment environments

It is expected that client machines are configured with appropriate security policies that give partially trusted code (based on appropriate evidence, such as Strong Name, Site, or URL) the appropriate permissions in order to use VisiBroker for .NET. Once VisiBroker for .NET is installed, and these security policies are in place, the application code that uses VisiBroker for .NET can be deployed using No Touch Deployment techniques. If you choose not to install VisiBroker for .NET, it is required that you give VisiBroker for .NET DLLs Full Trust for VisiBroker for .NET to function properly. It is

recommended that you use StrongName membership condition to give VisiBroker for
.NET DLLs full trust unless other membership conditions are deemed more appropriate
for your environment.

14

Using VisiBroker for .NET with COM

This chapter presents a set of development techniques for using VisiBroker for .NET to enable COM-based client applications to access server-side components developed with RMI, EJB or CORBA.

Although in theory any object developed for the Common Language Runtime (CLR) can be exposed to COM-based clients, in practice certain development and deployment techniques will make such access simple, flexible, and trouble-free. We discuss these various technique in this chapter, and show you how to use VisiBroker for .NET to make COM access work.

The first problem encountered when exposing managed objects to COM clients is determining which objects to expose. In general, it is desirable to expose the types which provide access to business logic, while hiding the types which simply provide the "middleware infrastructure." Following this guideline, your component interfaces should be visible to COM, while your marshaling stubs should be invisible.

When run with the –COM flag, the VisiBroker for .NET compiler adds the `ComVisible` attribute to all public type declarations:

    [System.Runtime.InteropServices.ComVisible(value)]

where `value` is `true` for types that are exposed to COM clients, and `false` for all other types.

The following generated types are visible to COM clients:

– All remote interfaces:

  – Interfaces defined in IDL

  – Interfaces defined in Java using RMI

  – Interfaces defined in Java using EJB

– Certain data types defined in IDL:

  – structs

  – unions

  – enums

  – valuetypes

- – Certain data types defined in Java:
  - – Public classes (excluding direct or indirect extensions of `java.lang.Throwable`)
  - – Public interfaces

The following generated types are invisible to COM clients:

- – Certain data types defined in IDL:
  - – exceptions
  - – constants
  - – valueboxes
- – Certain data types defined in Java:
  - – Classes which extend (directly or indirectly) `java.lang.Throwable`

All other generated classes and interfaces:

- – The `Helper` class
- – The `ValueFactory` and `ValueData` class
- – The `Operations` interface
- – The `MarshalingStub` and `LocalStub` classes
- – The `RemotingProxy` class
- – The `POA` and `POATie` classes

Overriding COM Visibility

Although the default COM visibility provided by the compiler is adequate for most applications, there are cases where it may be desirable to fine-tune to COM visibility of certain types. All ComVisible declarations include the fully-scoped type name immediately prior to the visibility value of `true` or `false`. It should be straightforward to write regular expressions to modify the visibility of individual types. VisiBroker for .NET compilers do not generate COM visibility attribute for type data members and you may need to fine tune this to control visibility of class data members. Also note that static and const members of a type are not COM visible.

ClassInterface attributes

By default, the following types will be annotated as requiring `ClassInterfaceType.AutoDual` interface classes:

- – Certain data types defined in IDL:
  - – structs
  - – unions
  - – valuetypes
- – Certain data types defined in Java:
  - – Public classes (excluding direct or indirect extensions of `java.lang.Throwable`)

In COM, an AutoDual interface class provides early binding COM clients with type access to all the public methods and properties provided by a given class. Unfortunately, AutoDual interface classes must be used with caution, as they can lead to fragile COM clients. As such, if a class that is AutoDual is modified, all the early binding COM clients that use that class must be recompiled (or redefined, in the case of using interpreted languages like Visual Basic). As such, it is strongly recommended that the AutoDual class interface only be used in situations where the underlying type is immutable. That is, newer versions of the types will not break existing COM clients.

However, the immutability requirement for using AutoDual in COM is analogous to the immutability requirement for the above listed types defined in IDL or Java. That is, if the user makes changes to the IDL definition of a struct, union or valuetype (or if a user makes changes to the serializable data fields of a class defined Java) it will lead to IIOP marshaling errors.

In short, IDL and Java types must be immutable with respect to field layout (which determines their marshaled format) in the same way that AutoDual types must be immutable with respect to their method and field layout. As such, the default behavior of the VisiBroker for .NET compilers is to annotate the above-mentioned types as AutoDual.

As with COM visibility, the AutoDual annotation of a given type (or all types) can be fine-tuned if required.

Note

This behavior may change in the future. VisiBroker for .NET compilers may choose to add `ClassInterface.None` and `ComInterfaceType.InterfaceIsDual` for generated interfaces.

Defining custom interfaces

Microsoft recommends using user-defined interfaces as a more robust alternative to using `ClassInterfaceType.AutoDual` on implementation classes, where the implementation class is likely to change over time. It is also suggested to mark the implementation classes with `ClassInterfaceType.None` ClassInterface attribute to avoid generation of the `_<impl class>` interface (which becomes the [default] interface otherwise.) The user-defined interfaces can be inspected and appropriately marked with `ComInterfaceType.InterfaceIsDual` InterfaceType attribute if necessary to generate dual interface COM servers.

At this time VisiBroker for .NET compilers do not explicitly mark classes with `ClassInterfaceType.None`. Neither are interfaces marked with `InterfaceIsDual`. VisiBroker for .NET compilers generate AutoDual flags for public implementation classes that are generated or Java based.

There are a number of different techniques that can be used, which we will illustrate using the following Java class definition:

```
public class Quote implements java.io.Serializable {
  private String symbol;
  private float price;
  public Quote(String symbol, float price) {
    this.symbol = symbol;
    this.price = price;
  }
  public String getSymbol() {
    return symbol;
  }
  public float getPrice() {
    return price;
  }
}
```

This Java class represents a stock quote, and contains data fields corresponding to the stock symbol and price. Note that this class allows users to access the Quote's symbol or price, but not to modify these values. Ideally, we would like our C# type to be likewise read-only with respect to the symbol and price fields.

By default, if we run the VisiBroker for .NET `java2cs` compiler over this class (with the –COM flag enabled), we will produce the following C# class:

```
[Serializable]
[ComVisible(true)]
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Quote  {
```

```
public Quote() {
}
public Quote(float Price, string Symbol) {
  this._Price = Price;
  this._Symbol = Symbol;
}
private float _Price;
public virtual float Price {
  get { return this._Price; }
  set { this._Price = value; }
}
private string _Symbol;
public virtual string Symbol {
  get { return this._Symbol; }
  set { this._Symbol = value; }
}
}
```

This class has the following drawbacks:

1 It uses the AutoDual mode, which makes it immutable, in the sense that most early bound clients need to recompile.

2 The `Symbol` and `Price` properties have public getters and setters. This is at odds with our design guidelines, which indicate that `Symbol` and `Price` should be read-only.

So, instead of using this generated class, we would like to introduce a user-defined interface. Here is the user-defined C# interface representing a `Quote`:

```
[System.Runtime.InteropServices.ComVisible(true)]
public interface Quote {
  string GetSymbol();
  float GetPrice();
}
```

This class is COM visible, and has getter methods for `Symbol` and `Price`. Mark this interface with the appropriate ComInterfaceType attribute to specify dual or dispatch interface or IUnknown as required.

The next step is to tell the VisiBroker for .NET compiler not to generate the `Quote` interface, since we are providing our own implementation. This is done by introducing a hint file, which contains the following hint:

```
<?xml version="1.0"?>
<hints>
  <hint>
    <java-class>Quote</java-class>
    <cs-sig-type>Quote</cs-sig-type>
    <cs-impl-type>QuoteImpl</cs-impl-type>
    <mode>automatic</mode>
  </hint>
</hints>
```

This hint indicates that the Java type `Quote` maps to a pair of C# types: a signature type `Quote`, and an implementation type `QuoteImpl`. We also specify that we will be using the `automatic` code-generation mode. (In fact, the
element can be omitted, as `automatic` is the default code-generation mode.)

The XML element `<cs-sig-type/>` indicates the type name that will be used when clients interact with a `Quote`. The XML element `<cs-impl-type/>` indicates the type that will be used to implement the `Quote` (for example, `QuoteImpl`).

The user must then provide implementations of both the public `Quote` type and the internal `QuoteImpl` type. The `Quote` interface was listed above. Below is the `QuoteImpl`:

```
internal class QuoteImpl : Quote {
  internal string Symbol;
```

```
    internal float Price;
    public string GetSymbol() {
      return Symbol;
    }
    public float GetPrice() {
      return Price;
    }
  }
```

A few notes about this implementation class:

– Since this class is marked `internal`, we do not have to indicate its COM visibility: only `public` types can be COM visible. This implementation class is invisible to COM clients (which was our intent).

– The `automatic` code generation mode indicated in the hint file requires that this class have fields corresponding to the serializable fields in the Java class. The Java class has two serializable fields (`symbol` and `price`) and thus our C# implementation class also has two such fields (`Symbol` and `Price`). Obviously, we could have implemented these fields as properties instead, if desired.

– The two serializable fields in `QuoteImpl` (`Symbol` and `Price`) must be marked as `internal` (or `public`), since these fields are read/written by the generated class `QuoteValueFactory` when marshaling a `QuoteImpl`. These fields cannot be `private` or `protected`.

An alternate technique is available for implementing the `QuoteImpl` class, if it is desirable to not have to "repeat" the serializable fields. In such cases, it is possible to implement the `QuoteImpl` by extending the generated class `QuoteValueData`:

```
  internal class QuoteImpl : QuoteValueData, Quote {
    public string GetSymbol() {
      return Symbol;
    }
    public float GetPrice() {
      return Price;
    }
  }
```

Note that this class does not declare the fields `Symbol` and `Price`, as these fields are "inherited" from the base class `QuoteValueData`.

Support for array-valued parameters and return values

There are known issues with respect to invoking methods from COM clients on types implemented in managed code, where one or more of the parameters or the return value of the method is an array type.

To address these issues, when the –COM flag is specified, the VisiBroker for .NET compilers generate an "overloaded" method for every such problematic method.

Let's consider, as an example, the following method:

```
  int[] GetLengths(string[] strings);
```

This method takes an array of `strings` as a parameter, and returns an array of `integers`, where each element in the result indicates the length of the corresponding input string. So, if this method is called as follows:

```
  string[] strings = { "VisiBroker", "Rocks" };
  int[] lengths = o.GetLengths(strings);
```

The result would be an array containing the elements 10 and 5.

Unfortunately, if we export this C# signature to COM, some COM clients will not be able to invoke the method `GetLengths`. For example, if we run the following Visual Basic code within an Excel spreadsheet:

```
Dim strings(1) As String
strings(0) = "VisiBroker"
strings(1) = "Rocks"
lengths = o.GetLengths(strings)
```

We will receive the following error:

```
Compile error: Function or interface marked as restricted, or the function uses
an Automation type not supported by Visual Basic
```

To handle this problem, the VisiBroker for .NET compiler will output an "overloaded" method with the following signature:

```
object GetLengthsForCom(object strings);
```

This method signature substitutes the type `object` for all array-valued parameters and/ or return values. (Note that this method is technically not overloaded with respect to the original method `GetLengths`, since we append the suffix `ForCom` to the original method name. We cannot use true overloading because C# does not permit method signatures that are overloaded based on return type.)

We can now use this generated method in our Visual Basic client:

```
Dim strings(1) As String
strings(0) = "VisiBroker"
strings(1) = "Rocks"
lengths = o.GetLengthsForCom(strings)
```

We will obtain a `lengths` value which is an array of 32-bit integers, where the array elements contain the values 10 and 5, as expected.

Avoiding ProgId collisions

Microsoft's COM interoperation documentation indicates that problems may occur when trying to export types with very long type names to COM clients. In particular, if the C# type name exceeds 39 characters, COM client may not be able to access the type unambiguously. Microsoft recommends adding a `ProgId` annotation to long type names that would otherwise be ambiguous. A simple workaround is to use regular expression-based tools to modify the code generated by the VisiBroker for .NET compiler.

15

Using VisiBroker for .NET with Borland GateKeeper

This chapter explains how to configure properties to use the VisiBroker GateKeeper service with VisiBroker for .NET applications. Refer to the *VisiBroker GateKeeper Guide* for information on using GateKeeper.

What is GateKeeper?

The Borland VisiBroker GateKeeper is a CORBA General Inter-ORB Protocol (GIOP) compliant GIOP Proxy Server that enables CORBA clients and servers to communicate across networks while conforming to security restrictions imposed by Internet browsers, firewalls, and Java sandbox security. In effect, GateKeeper serves as a gateway or proxy for clients and servers when security restrictions prevent clients from communicating with the servers directly.

GateKeeper is often used when you do not want to expose the server directly to clients or when a client's access to the server is restricted. In the latter case, either the client is an unsigned applet or there is an intervening firewall.

Enabling the VisiBroker for .NET Firewall feature

VisiBroker for .NET supports a firewall that is compliant with CORBA 3.0. By default, the firewall feature is turned off in VisiBroker for .NET. If you are developing a VisiBroker for .NET application to work with the VisiBroker GateKeeper service, you will need to turn the firewall feature on explicitly with the `janeva.firewall` property.

VisiBroker for .NET server-side configuration

In order to enable the client to communicate with the server through the GateKeeper, the server has to export the firewall path to the client by setting certain properties. The following table describes the properties specific to server side configuration.

Table 15.1   Server-side GateKeeper properties

Property	Valid values	Description
vbroker.orb.exportFirewallPath	true false (default)	When this property is set to true the firewall path is embedded in the server's IOR profile component. vbroker.orb.exportFirewallPath=true
vbroker.se.iiop_tp.firewallPaths	<empty> (default) <paths>	Use this property to declare all firewall paths. <paths> is a set of user defined names for the communication paths from the clients to the servers, separated with commas. vbroker.se.iiop_tp.firewallPaths=x,y
vbroker.firewall-path.<pathname>	<empty> (default) <components>	Specifies the list of components in the firewall path <pathname>. vbroker.firewall-path.x=a,b vbroker.firewall-path.y=c
vbroker.firewall.<component>.type	<empty> PROXY TCP	Specifies the type of the components. vbroker.firewall.a.type = PROXY vbroker.firewall.b.type = TCP
vbroker.firewall.<component>.ior	<empty> <ior_filename> <ior_URL> IOR:<stringified_ior>	Specifies the IOR of the component. This is specified together with vbroker.firewall.<component.type>=PROXY. ■ file:C:/GateKeeper/GateKeeper.ior ■ http://www.inprise.com/GK GateKeeper.ior ■ IOR:2398402841729073423497234234234
vbroker.firewall.<component>.host	<empty> <fake host name>	Specifies a fake host name for the component server. This is specified together with vbroker.firewall.<component>.type=TCP and the component is a TCP Firewall with NAT.
vbroker.firewall.<component>.iiop_port	<empty> <fake IIOP Port>	Specifies a fake IIOP port for the component server. This is specified together with vbroker.firewall.<component>.type=TCP and the component is a TCP Firewall with NAT.
vbroker.firewall.<component>.ssl_port	<empty> (default) <fake SSL Port>	Specifies a fake SSL port for the component server. This is specified together with vbroker.firewall.<component>.type=TCP and the component is a TCP Firewall with NAT.

Table 15.1   Server-side GateKeeper properties (continued)

Property	Valid values	Description
`vbroker.firewall.<component>.hiop_port`	`<empty>` (default) `<fake HIOP Port>`	Specifies a fake HIOP port for the component server. This is specified together with `vbroker.firewall.<component>.type=TCP` and the component is a TCP Firewall with NAT.
`vbroker.orb.enableBiDir`	client server both none (default)	If the client defines `vbroker.orb.enableBiDir=client`, and the server defines `vbroker.orb.enableBiDir=server`, the value of `vbroker.orb.enableBiDir` in GateKeeper determines the state of the connection. If you set the `vbroker.se.exterior.scm.ex--iiop.manager.importBiDir` property to true, GateKeeper will accept bidirectional connections from the client. Setting the `vbroker.se.exterior.scm.ex--iiop.manager.exportBiDir` property to true causes GateKeeper to request bidirectional connections with the server.

VisiBroker for .NET client-side configuration

The following table describes the properties specific to client side configuration.

Property	Valid values	Description
`vbroker.orb.gatekeeper.ior`	`<empty>` (default) `<ior_filename>`	Specifies the URL of the GateKeeper IOR file.
`vbroker.orb.alwaysProxy`	false (default) true	Specifies whether the client must always connect to the server via GateKeeper.
`vbroker.locator.ior.ior`	`<empty>` (default) `<ior_filename>`	Specifies URL of the GateKeeper Locator IOR file. GateKeeper provides limited location services. It cannot forward location requests to another GateKeeper.
`vbroker.orb.alwaysTunnel`	false (default) true	Specifies whether the client must always make HTTP tunnel (IIOP wrapper) connections to the server.
`vbroker.orb.enableBiDir`	client server both none (default)	You can selectively make bidirectional connections. If the client defines `vbroker.orb.enableBiDir=client`, and the server defines `vbroker.orb.enableBiDir=server`, the value of `vbroker.orb.enableBiDir` in GateKeeper determines the state of the connection. For example, if you set the `vbroker.se.exterior.scm.ex--iiop.manager.importBiDir` property to true, GateKeeper will accept bidirectional connections from the client. Setting the `vbroker.se.exterior.scm.ex--iiop.manager.exportBiDir` property to true causes GateKeeper to request bidirectional connections with the server.

Callbacks with GateKeeper's bidirectional support

With bidirectional IIOP, servers use the client-initiated connections to transmit asynchronous information back to the clients. Servers need not initiate any connections to the client.

Figure 15.1   Callback with GateKeeper's bidirectional support
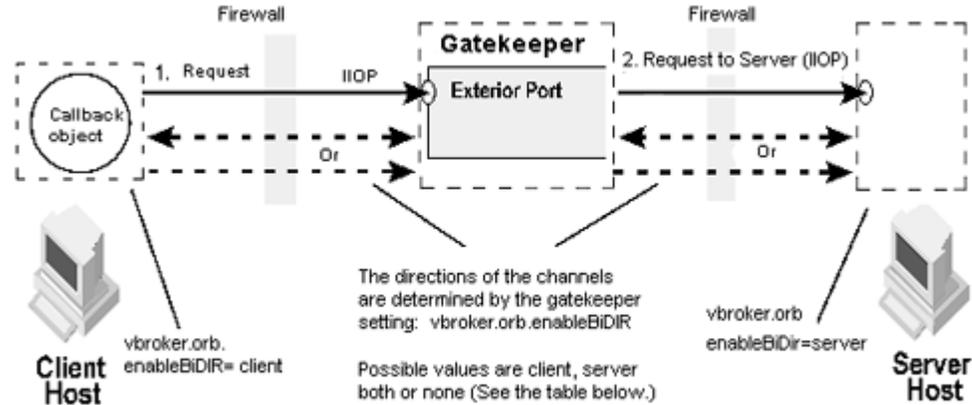
In the figure above, GateKeeper sits between the client and server and therefore it acts as a server for the client and as a client for the server. The Client/GateKeeper and the GateKeeper/Server communication channels can be set to unidirectional or bidirectional connections.

You can also selectively set the channels to unidirectional or bidirectional. If the client defines `vbroker.orb.enableBiDir=client` and the server defines `vbroker.orb.enableBiDir=server`, the following table describes the type of channels for the different values of vbroker.orb.enableBiDir for GateKeeper.

vbroker.orb.enableBiDir=	Client GateKeeper	GateKeeper Server
client	unidirectional	bidirectional
server	bidirectional	unidirectional
both	bidirectional	bidirectional
none	unidirectional	unidirectional

Security considerations

Use of bidirectional IIOP may raise significant security issues. In the absence of other security mechanisms, a malicious client may claim that its connection is bidirectional for use with any host and port it chooses. In particular, a client may specify the host and port of security-sensitive objects not even resident on its host. In the absence of other security mechanisms, a server that has accepted an incoming connection has no way to discover the identity or verify the integrity of the client that initiated the connection. Further, the server might gain access to other objects accessible through the bidirectional connection. If there are any doubts as to the integrity of the client, it is recommended that bidirectional IIOP not be used. For security reasons, a server running VisiBroker for .NET will not use bidirectional IIOP unless explicitly configured to do so.

Examples

The following example shows a client side configuration. The client always communicates with the server via GateKeeper as a proxy.

```
<configuration>
  <configSections>
    <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
  </configSections>
```

```
    <visinet>
      ...
      <firewall enabled="true"/>
      <vbroker vbroker.orb.alwaysProxy="true"/>
    </visinet>
</configuration>
```

The following example shows a server side configuration. It defines a firewall path called "internet" with one node named "proxy". This node is of the PROXY type.

```
<configuration>
    <configSections>
      <section name="visinet" type="Janeva.Settings, Borland.Janeva.Runtime"/>
    </configSections>
    <visinet>
      ...
      <firewall enabled="true"/>
      <server defaultPort="10000">
         <remoting enabled="false"/>
      </server>
      <vbroker vbroker.orb.exportFirewallPath="true"
            vbroker.se.iiop_tp.firewallPaths="internet"
            vbroker.firewall-path.internet="proxy"
            vbroker.firewall.proxy.type="PROXY"
            vbroker.firewall.proxy.ior="http://localhost:9091/gatekeeper.ior"/>
    </visinet>
</configuration>
```

Compiler options

This chapter describes the options you can use with the VisiBroker for .NET compilers.

Options are processed in order from left to right, giving the last value precedence. All the options in the list are enabled by default and they are preceded by a hyphen (-). For some of the options you can use the inverse of the default value by either using -[no_] or removing the hyphen. For example, to display a "warning" if a `#pragma` is not recognized, the default value is:

```
warn_unrecognized_pragmas
```

To turn off the default, type the following command:

```
-no_warn_unrecognized_pragmas
```

idl2cs[j]

The `idl2cs` tool compiles an IDL source file and creates a directory structure containing the C# mappings for the IDL declarations. The `idl2cs` and `idl2csj` tools are identical except that `idl2csj` will run in a Java-only environment (allowing the compiler to be run on platforms without .NET, including Unix and older Windows machines), while `idl2cs` will run in a C#-only (.NET Framework) environment.

One IDL file maps to one C# file. The default output filename for Foo.idl is Foo.cs. The output file can be specified using the `-o` option. Typically IDL file names must end with the `.idl` extension.

Syntax

```
idl2cs [options] {source_file}
```

Example

```
idl2cs -no_Object_method Example.idl
```

Option	Description
-D, -define foo[=bar]	Defines a preprocessor macro `foo`, optionally with a value `bar`. You can use this option more than once.
-I, -include <dir>	Specifies the full or relative path to the directory for #include files. Used in searching for include files. You can use this option more than once.

Option	Description
-P, -no_line_directives	Suppresses the generation of line number information in the generated code. The default is off.
-H, -list_includes	Prints the full paths of included files on the standard error output. The default is off.
-C, -retain_comments	Retains comments from IDL file in the preprocessor output. The default is off.
-U, -undefine foo	Undefines a preprocessor macro *foo*.
-[no_]idl_strict	Specifies strict adherence to OMG standard interpretation of idl source. The default is off.
-[no_]builtin (TypeCode|Principal)	Creates builtin type ::TypeCode or ::Principal. The default is on.
-[no_]warn_unrecognized_pragmas	Displays a warning that appears if a #pragma is not recognized. The default is on.
-[no_]back_compat_mapping	Use mapping that is compatible with VisiBroker 3.x. The default is off.
-[no_]preprocess	Preprocesses the input file before parsing. The default is on.
-[no_]preprocess_only	Stops parsing the input file after preprocessing. The default is off.
-[no_]warn_all	Turns all warnings on or off simultaneously. The default is off.
-[no_]case_sensitive	Treat identifiers in a case-sensitive manner. The default is on.
-[no_]comments	Suppresses the generation of comments in the code. The default is on.
-gen_included_files	Generates code for #included files. The default is off.
-list_files	Lists files written during code generation. The default is off.
-root_dir <path>	Specifies the directory in which the generated files reside.
-[no_]servant	Generates servant (server-side) code. The default is on.
-[no_]tie	Generates Tie classes. The default is on.
-[no_]warn_missing_define	Warns if any forward declared interfaces were not defined. The default is on.
-[no_]strict_reverse_mapping	Use strict Java reverse mapping. The default is off.
-o <file>	Specifies the name of the output file, or "-" for stdout.
-[no_]bind	Generates the bind() code. The default is off.
-idl2namespace <IDL name> <ns>	Overrides default namespace for a given IDL container type.
-[no_]Object_method	Generates all methods on Object. The default is on.
-namespace <ns>	Specifies the root namespace for generated code.
-map_keyword <kwd> <replacement>	Specifies the keyword to avoid and designates its replacement.
-[no_]mixed_caps	Converts methods to MixedCaps and members to mixedCaps. The default is on.
-[no_]examples	Generates sample implementations. The default is off.
-hint_file <file_uri>	Specifies a hint file URI for custom type mappings. Only available to idl2csj.
-[no_]remoting_proxy	Generates a Proxy class for use with .NET Remoting. The default is on.
-h, -help, -usage, -?	Prints option help information.

Option	Description
-version	Displays the VisiBroker for .NET software version number.
file1 [file2] ...	Designates one or more files to process, or "-" for stdin.

java2cs

This command generates C# code from a Java class. java2cs translates a remote interface defined in Java RMI into corresponding C#. It will translate remote interfaces, EJB interfaces, and value classes into C#. Note that java2cs will also translate types referred to directly or indirectly by the input types.

You can use more than one Java class name (in Java byte code) as input. If you enter more than one class name, make sure you include spaces in between the class names. Use fully scoped class names. You can also provide an EJB JAR or EAR or any library JAR as input.

Note

The java2cs compiler does not support overloaded methods on CORBA interfaces.

Note

To use this compiler, you must have a Java Virtual Machine supporting JDK 1.4 or later.

If you use a class that extends org.omg.CORBA.IDLEntity in some Java remote interface definition, it must have the following:

– an IDL file that contains the IDL definition for that type because the org.omg.CORBA.IDLEntity interface is a signature interface that marks all IDL data types mapped to Java.

– all related (supporting) classes according to the CORBA 2.4 IDL2Java Specification from the Object Management Group (OMG).

If you use a class that extends org.omg.CORBA.IDLEntity in some Java remote interface definition, use the -import <IDL files> directive in the java2cs tool's command line.

For more information, refer to the CORBA 2.4 IDL2Java Specification located at http://www.omg.org.

Syntax

```
java2cs [options] {input_class_name}
```

Example

```
java2cs -no_tie Account Client Server
```

Use java2cs if you have existing Java byte code that you wish to adapt to use distributed objects or if you do not want to write IDL. By using java2cs, you can generate the necessary container classes, client stubs, and server skeletons from Java byte code.

Option	Description
-D, define foo[=bar]	Defines a preprocessor macro foo, optionally with a value bar.
-I, -include <dir>	Specifies the full or relative path to the directory for #include files. Used in searching for include files.
-P, -no_line_directives	Suppresses the generation of line number information in the generated code. The default is off.
-H, -list_includes	Prints the full paths of included files on the standard error output. The default is off.

Option	Description
-C, -retain_comments	Retains comments from Java file when the C# code is generated. Otherwise, the comments will not appear in the C# code. The default is off.
-U, -undefine foo	Undefines a preprocessor macro foo.
-[no_]idl_strict	Specifies strict adherence to OMG standard interpretation of IDL source. The default is off.
-[no_]builtin (Typecode|Principal)	Creates builtin type ::TypeCode or ::Principal. The default is on.
-[no_]warn_unrecognized_pragmas	Displays a warning that appears if a #pragma is not recognized. The default is on.
-[no_]back_compat_mapping	Uses mapping that is compatible with VisiBroker 3.x. The default is off.
-[no_]preprocess	Preprocesses the input file before parsing. The default is on.
-[no_]preprocess_only	Stops parsing the input file after preprocessing. The default is off.
-[no_]warn_all	Turns all warnings on or off simultaneously. The default is off.
-[no_]idlentity_array_mapping	Maps array of IDLEntity to boxedIDL in boxedRMI. The default is off.
-exported <pkg>	Specifies an exported package.
-[no_]export_all	Exports all packages. The default is off.
-import <IDL file name>	Loads extra IDL definitions.
-imported <pkg> <IDL file name>	Specifies the name of an imported package.
-gen_hints <file-name>	Produces a template hint file. The default is off.
-show_ignored	Prints out all warnings about unloadable classes. The default is off.
-list_classes	Prints out classes which are compiled. The default is off.
-[no_]ignore <class>|<package>	Ignores a class (or all classes in a package) and all classes that depend on it.
-[no_]case_sensitive	Treats identifiers in a case-sensitive manner. The default is on.
-[no_]comments	Places comments in generated code. The default is on.
-gen_included_files	Generates code for #included files. The default is off.
-list_files	Lists files written during code generation. The default is off.
-root_dir <path>	Specifies the directory in which the generated files reside.
-[no_]servant	Generates servant (server-side) code. The default is on.
-[no_]tie	Generates tie classes. The default is on.
-[no_]warn_missing_define	Warns if any forward declared names were never defined. The default is on.
-[no_]strict_reverse_mapping	Uses strict Java reverse mapping. The default is off.
-o <file>	Specifies the name of the output file, or "-" for stdout.
-[no_]bind	Generates the bind() code. The default is off.
-idl2namespace <IDL name> <ns>	Overrides default namespace for a given IDL container type.
-[no_]Object_method	Generates all methods defined in java.lang.Object methods, such as string and equals. The default is on.

Option	Description
-namespace <ns>	Specifies the root namespace for generated code.
-map_keyword <kwd> <replacement>	Specifies the keyword to avoid and designates its replacement.
-[no_]mixed_caps	Converts methods to MixedCaps and members to mixedCaps. The default is on.
-[no_]examples	Generates sample implementations. The default is off.
-hint_file <file-uri>	Specifies a hint file URI for custom type mappings.
-[no_]remoting_proxy	Generates a Proxy class for use with .NET Remoting. The default is on.
-h, -help, -usage, -?	Prints option help information.
-version	Displays the VisiBroker for .NET software version number.
[file.jar] [file.ear] ...	Optional list of J2EE archives (JAR or EAR) to process.
[class1] [class2] ...	Optional list of Java classes to process. Note that is no target Java classes are specified, they will be determined automatically from the specified J2EE archive.

IDL to C# mapping

This chapter describes the VisiBroker for .NET IDL-to-C# language mapping, as generated by the `idl2cs` code generation tool.

Names

By default, IDL names and identifiers are mapped to C# names and identifiers using mixed case. This is an optional mapping, controlled by the compiler directive — `[no_]mixed_caps` with mixed caps being the default.

By default, methods, attributes, and factory methods will be named so that the name begins with an initial capital letter and each logical "word" in the name also has an initial capital letter. In this context, an identifier part is considered a logical word if it is separated by an underscore (_) on either side. For example an IDL method name, `foo_bar`, would be mapped to `FooBar` in the generated C# code.

Enums and member fields of structs, exceptions, and valuetypes are mapped to names beginning with a lower case letter, but each logical "word" that follows will have an initial capital letter. For example, `foo_bar` would become `fooBar`.

There is an exception where names in all capital letters would not be translated to mixed case names in the generated C# code, nor would any intermediate underscores be collapsed. For example `FOO_BAR` would remain as is.

Leading and trailing underscores are preserved in all translated names. For example, `_foo_bar_` becomes `_FooBar_`.

If a name collision with a C# keyword is generated in the mapped C# code, the name collision is resolved by prepending a commercial at (@) symbol to the mapped name. The @ prefix is a C# convention. For example, the C# keyword `string` would be mapped to `@string`, but the symbol's real name is still `string` when interpreted in .NET. In other .NET languages where `string` is not a keyword (for example, J#) the symbol is recognized as `string`.

In addition, because of the nature of the C# language, a single IDL construct may be mapped to several (differently named) C# constructs. The additional names are constructed by appending a descriptive suffix. For example, the IDL interface `AccountManager` is mapped to the C# interface `AccountManager` and additional C# classes `AccountManagerOperations` and `AccountManagerHelper`.

In the exceptional cases where the additional names may conflict with other mapped IDL names, the resolution rule described above is applied to the other mapped IDL names. In other words, the naming and use of required "additional" names takes precedence.

For example, an IDL interface whose name is `fooHelper` is mapped to C# class `_fooHelper`, regardless of whether an interface named `foo` exists. The helper class for C# class `_fooHelper` is named `_fooHelperHelper`.

IDL names that would normally be mapped unchanged to C# identifiers that conflict with C# reserved words will have the collision rule applied.

Reserved generated suffixes

The mapping reserves the use of several names for use as class suffixes. The use of any of these names for a user-defined IDL type or interface (assuming it is also a legal IDL name) will result in the mapped name having an underscore (_) prepended. The reserved generated suffix names are as follows:

– Helper—The C# class `<type>Helper`, where `<type>` is the name of an IDL user-defined type

– NS—The nested scope C# namespace name `<interface>NS`, where `<interface>` is the name of an IDL interface.

– Operations

– POATie

– POA

– RemotingProxy

– ValueFactory

– ValueData—The C# classes `<valuetype>ValueData` and `<valuetype>ValueFactory` where <valuetype> is the name of an IDL valuetype type.

Reserved words

The mapping reserves the use of several words for its own purposes. The use of any of these words for a user-defined IDL type or interface (assuming it is also a legal IDL name) will result in the mapped words having a commercial at (@) symbol prepended. The reserved keywords in the C# language are as follows:

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
dontinue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	float	fixed	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort
using	virtual	void	volatile
while			

Basic types

The following table shows how the defined IDL types map to basic C# types.

IDL type	C# type
boolean	bool
char	char
wchar	char
octet	byte
string	string
wstring	string
short	short
unsigned short	short
long	int
unsigned long	int
longlong	long
unsigned longlong	long
float	float
double	double

When there is a potential mismatch between an IDL type and its mapped C# type, a standard CORBA exception can be raised. For the most part, exceptions are in two categories,

– Range of the C# type is larger than the IDL type. For example, C# chars are a superset of IDL chars.

– Because there is no uniform support in .NET for unsigned types, unsigned IDL types are mapped to their signed equivalents in C#. The developer is responsible for ensuring that large unsigned IDL type values are handled correctly as negative integers in .NET.

Additional details are described in the following sections.

C# null

The C# null may only be used to represent null CORBA object references and valuetypes (including recursive valuetypes). For example, a zero length string, rather than null must be used to represent the empty string. This is also true for arrays and any constructed type, except for valuetypes. If you attempt to pass a null for a structure, it will raise a system NullReferenceException.

Boolean

The IDL type boolean is mapped to the C# type bool. The IDL constants TRUE and FALSE are mapped to the C# constants true and false.

Char

IDL characters are 8-bit quantities representing elements of a character set while C# characters are 16-bit unsigned quantities representing Unicode characters. To enforce type-safety, the CORBA runtime asserts range validity of all C# chars mapped from IDL chars when parameters are marshaled during method invocation. If the char falls outside the range defined by the character set, a CORBA::DATA_CONVERSION exception is thrown.

The IDL wchar maps to the C# char type.

String and WString

The IDL type `string`, both bounded and unbounded variants, is mapped to the C# type `string`. Range checking for characters in the string as well as bounds checking of the string are done at marshal time.

The IDL type `wstring`, used to represent Unicode strings, is mapped to the C# type `string`. Bounds checking of the string is done at marshal time.

Integer types

IDL `short` and `unsigned short` map to C# type `short`. IDL `long` and `unsigned long` map to C# type `int`.

Note

Because there is no uniform support in .NET for unsigned types, unsigned IDL types are mapped to their signed equivalents in C#. The developer is responsible for ensuring that negative integers in .NET are handled correctly as large unsigned values.

IDL type extensions

This section summarizes VisiBroker for .NET support for IDL type extensions. The first table provides a summary for quick look-ups. This is followed by a table summarizing support for new types.

Type	Supported in VisiBroker for .NET
`longlong`	yes
`unsigned longlong`	yes
`long double`	no[1]
`wchar`	yes[2]
`wstring`	yes[2]
`fixed`	no[1]

[1] VisiBroker for .NET might support it in a future release of the OMG standard implementation.
[2] UNICODE is used "on the wire."

New types	Description
`longlong`	64-bit signed 2's complements integers
`unsigned longlong`	64-bit unsigned 2's complements integers
`long double`	IEEE Standard 754-1985 double extended floating point
`wchar`	Wide characters
`wstring`	Wide strings
`fixed`	Fixed-point decimal arithmetic (31 significant digits)

Constants

Constants are mapped to a public abstract class with the same name as the constant and containing a `public const int` field named `Value`. This field holds the constant's value.

This code sample shows the mapping of an IDL constant within a module to a C# class.

```
/* From Example.idl: */
module Example {
const long aLongerOne = -123;
};
```

```
// Example.cs
namespace Example {
public abstract class ALongerOne {
   public const int Value = (int) -123;
   }
}
```

Note:

Constants within an interface or valuetype are put into a namespace with the NS suffix appended to the name of the interface or valuetype.

Constructed types

IDL constructed types include `enum`, `struct`, `union`, `sequence`, and `array`. The types `sequence` and `array` are both mapped to the C# `array` type. The IDL constructed types `enum`, `struct`, and `union` are mapped to a C# class that implements the semantics of the IDL type. The C# class generated will have the same name as the original IDL type.

Enumerations

An IDL `enum` is mapped to a C# enum with the same name as the enum type which declares the enum values. The code sample below is an example of an IDL enum mapped to a C# enum.

```
// Example.idl
module Example {
   enum EnumType (first, second, third};
};

// Example.cs
public enum EnumType {
   first
   second
   third
}
```

Structs

An IDL `struct` is mapped to a C# class with the same name that provides instance variables for the fields in IDL member ordering and a constructor for all values. This code sample shows the mapping of an IDL struct to C#.

```
// Example.idl
module Example {
   struct StructType {
      long field1;
      string field2;
   };

};

// Example.cs
public sealed class StructType
   public int field1;
   public string field2;
   public StructType() {
      field2 = "";
   }
   public StructType (int field1, string field2) {
      this.field1 = field1;
      this.field2 = field2;
```

```
      }
      override public string ToString() {
         System.Text.StringBuilder _ret =
            new System.Text.StringBuilder("struct Example.StructType {");
         _ret.Append("\n");
         _ret.Append("int field1=");
         _ret.Append(field1);
         _ret.Append(",\n");
         _ret.Append("string field2=");
         _ret.Append("field2 != null?'\"' + field2 + '\"':null);
         _ret.Append("\n");
         _ret.Append("}");
         return _ret.ToString();
      }
      override public int GetHashCode() {
         returns base.GetHashCode();
      }
      override public bool Equals(object o) {
         if(this == o) return true;
         if(o == null) return false;
         if(o is Example.StructType) {
            Example.StructType obj = (Example.StructType) o;
            bool res = true;
            do {
               res = this.field1 == obj.field1;
               if(!res) break;
               res = this.field2 == obj.field2 ||
                   (this.field2 != null && obj.field2 != null &&
                      this.field2Equals(obj.field2));
            } while(false);
            return res;
         }
         else {
            return false;
         }
      }
   }
```

Unions

An IDL `union` is mapped to a sealed C# class of the same name. It provides the
following:

– Default constructor

– Accessor method for the union's discriminator, named `discriminator()`

– Accessor method for each branch

– Modifier method for each branch

– Modifier method for each branch having more than one case label

– Default modifier method, if needed

If there is a name clash with the mapped union type name or any of the field names,
the normal name conflict resolution rule is used: prepend an underscore (_) for the
discriminator.

The branch accessor and modifier methods are overloaded and named after the
branch. Accessor methods will raise the `CORBA::BAD_OPERATION` system exception if the
expected branch has not been set.

If there is more than one case label corresponding to a branch, the simple modifier
method for that branch sets the discriminant to the value of the first case label. In

addition, an extra modifier method which takes an explicit discriminator parameter is generated.

If the branch corresponds to the `default` case label, then the modifier method sets the discriminant to a value that does not match any other case labels.

It is illegal to specify a union with a default case label if the set of case labels completely covers the possible values for the discriminant. It is the responsibility of the C# code generator (for example, the IDL compiler, or other tool) to detect this situation and refuse to generate illegal code.

A default method `_default()` is created if there is no explicit default case label, and the set of case labels does not completely cover the possible values of the discriminant. It will set the value of the union to be an out-of-range value.

This code sample shows the mapping of an IDL union to C#.

```
// Example.idl
module Example {
    enum EnumType { first, second, third, fourth, fifth, sixth };
    union UnionType switch (EnumType) {
       case first: long win;
       case second: short place;
       case third:
       case fourth: octet show;
       default: boolean other;
       };

};

// Example.cs
public sealed class UnionType {
   private object _object;
   private Example.EnumType _disc = Example.EnumType.fifth;
   internal bool _defaultState = false;

   // constructor
   public UnionType() {
   }
// discriminator accessor
   public Example.EnumType discriminator() {
      return _disc;
   }

   // win
   public int Win() { ... }
   public void Win(int _vis_value) { ... }

   // place
   public short Place() { ... }
   public void Place(short _vis_value) { ... }

   // show
   public byte Show() { ... }
   public void Show(byte _vis_value) { ... }
   public void Show(Example.EnumType disc, byte _vis_value) { ... }

   // other
   public bool Other() {...}
   public void Other(bool _vis_value) { ... }
   public void Other(Example.EnumType disc, bool _vis_value) { ... )
   override public string ToString () { . . .}
   overrinde public int GetHashCode() { ... }
   public bool Equals(object o) { . . .}
}
```

Sequences and Arrays

An IDL `sequence` is mapped to a C# array. In the mapping, anywhere the sequence type is needed, an array of the mapped type of the sequence element is used.

An IDL `array` is mapped in the same way as an IDL bounded sequence. In the mapping, anywhere the array type is needed, an array of the mapped type of array element is used. In C#, the natural C# subscripting operator is applied to the mapped array. The length of the array can be made available in C#, by bounding the array with an IDL constant, which will be mapped as per the rules for constants.

The following code sample shows the mapping for an array.

```
// Example.idl
const long ArrayBound = 42;
typedef long larray[ArrayBound];

// Example.cs
public abstract class ArrayBound {
   public const int Value = (int) 42;
}
```

Modules

An IDL module is mapped to a C# namespace with the same name. All IDL type declarations within the module are mapped to corresponding C# class or interface declarations within the generated namespace.

IDL declarations not enclosed in any modules are mapped into the (unnamed) C# global scope.

The code sample below shows the C# code generated for an IDL module.

```
// Example.idl
module Example {
    ...
};

// Example.cs
namespace Example {
    ...
}
```

Interfaces

Given a user-defined type named `Foo`, the `idl2cs` compiler generates the following:

– `public sealed class FooHelper`

– `public interface Foo : CORBA.Object, Example.FooOperations`

– `public class FooOperations`

– `public class _FooStub`

There are no special "nil" object references. C# `null` can be passed freely wherever an object reference is expected.

Attributes are mapped to a pair of C# accessor and modifier methods. These methods have the same name as the IDL attribute and are overloaded. There is no modifier method for IDL "readonly" attributes.

This code sample shows the mapping of an IDL interface to C#.

```
// Example.idl
module Example {
    interface Foo {
        long method(in long arg) raises(AnException);
```

```
        attribute long assignable;
        readonly attribute long nonassignable;
    };
};

// Example.cs
namespace Example {
    public sealed class FooHelper { ... }
    public interface Foo : CORBA.Object, Example.FooOperations {
    }
    public interface FooOperations {
        int Method(in long arg) throws Example.AnException;
        int Assignable();
        void Assignable(int assignable);
        int Nonassignable ();
    }
    public class _FooStub : CORBA.ObjectIml, Example.Foo { ... }
}
```

Signature and Operations interfaces

In the example above, the two interfaces, `Foo` and `FooOperations`, provide the complete signature of your IDL interface when mapped to C#. **The signature** interface defines the signature for each interface you declare in your IDL file, while the Operations interface provides the implementation details.

The Operations interface contains only the operations and attributes declared in the IDL interfaces. The C# `Operations` interface contains the mapped operation signatures. Methods can be invoked on an object reference to this interface.

Helper classes

A `Helper` class is provided for most of the classes in the CORBA namespace. They are also generated by the `idl2cs` compiler for user-defined types and are given the name of the class that is generated for the type with an additional `Helper` suffix. The reason for using the `Helper` class is to avoid loading the methods that the class offers if they are not needed. Several static methods needed to manipulate the type are supplied.

– `Any` insert and extract operations for the type

– Getting the repository id

– Getting the typecode

– Reading and writing the type from and to a stream

The `Helper` class declares a static narrow method that allows an instance of `CORBA.Object` to be narrowed to the object reference of a more specific type. The IDL exception `CORBA::BAD_PARAM` is thrown if the narrow fails because the object reference doesn't support the request type. A different system exception is raised to indicate other kinds of errors. Trying to narrow a null will always succeed with a return value of null.

For objects like mapped structures, enumerations, unions, exceptions, valuetypes, and valueboxes, the `Helper` class provides methods for reading and writing the object to a stream and returning the object's repository identifier. The `Helper` classes generated for interfaces contain additional methods, like `bind` and `narrow`.

Methods for all Helper classes

The following methods appear in all generated Helper classes.

```
public static <interface_name> Extract(CORBA.Any any)
```

This method extracts the type from the specified `Any` object.

Parameter	Description
any	The Any object to contain the object.

public static void **Insert**(CORBA.Any **any**, <type_name> **_vis_value**)

This method insert a type into the specified `Any` object.

Parameter	Description
any	The Any object to contain the type.
_vis_value	The type to insert.

public static <type_name> **Read**(CORBA.InputStream **_input**)

This method reads a type from the specified input stream.

Parameter	Description
input	The input stream from which the object is read.

public static CORBA.TypeCode **GetTypeCode**()

This method returns the `TypeCode` associated with this object.

public static void **Write**(CORBA.OutputStream **_output**, <type_name> _vis_value)

This method writes a type to the specified output stream.

Parameter	Description
output	The output stream to which the object is written.
value	The type to be written to the output stream.

Methods generated for interfaces

public static <*interface_name*> **Bind**()

This method attempts to bind to any instance of an object of type <*interface_name*>.

public static <*interface_name*> **Bind**(string **name**)

This method attempts to bind to an object of type <*interface_name*> that has the specified instance name.

Parameter	Description
name	The instance name of the desired object.

public static <*interface_name*> **Bind**(string **name**, string **host**,
  CORBA.Visi.BindOptions **options**)

This method attempts to bind to an object of type <*interface_name*> that has the specified instance name and which is located on the specified host, using the specified `BindOptions`.

Parameter	Description
name	The instance name of the desired object.
host	The optional host name where the desired object is located.
options	The bind options for this object.

public static <*interface_name*> **Narrow**(CORBA.Object **obj**)

This method attempts to narrow a `CORBA.Object` reference to an object of type `<interface_name>`. If the object reference cannot be narrowed, a `null` value is returned.

Parameter	Description
obj	The object to be narrowed to the type `<interface_name>`.

Generated stub classes

A stub class is generated by the `idl2cs` compiler to provide a stub implementation for `<interface_name>` which the client calls. This class provides the implementation for transparently acting on an object implementation.

Abstract interfaces

An IDL abstract interface is mapped into a single public C# interface with the same name as the IDL interface. The mapping rules are similar to the rules for generating the C# operations interface for a nonabstract IDL interface. However, this interface also serves as the signature interface.

The mapped C# interface has the same name as the IDL interface and is also used as the signature type in method declarations when interfaces of the specified types are used in other interfaces. It contains the methods which are the mapped operations signatures.

Passing parameters

IDL parameters are mapped to normal C# actual parameters. The results of IDL operations are returned as the result of the corresponding C# method.

This code sample shows the IDL parameter mapping to C#.

```
// Example.idl
module Example {
    interface Modes {
        long operation(in long inArg, out long outArg, inout long inoutArg);
    };
};

// Example.cs
namespace Example;
public interface Modes : CORBA.Object, Example.ModesOperations {
}
public interface ModesOperations {
    int Operation(int inArg, out int outArg, ref int inoutArg);
}
```

Interface scope

OMG IDL to C# mapping specification does not allow declarations to be nested within an interface scope, nor does it allow namespaces and interfaces to have the same name. Accordingly, interface scope is mapped to a package with the same name with an NS suffix.

Mapping for exceptions

IDL exceptions are mapped very similarly to structs. They are mapped to a C# class that provides instance variables for the fields of the exception and constructors.

CORBA system exceptions are unchecked exceptions. They inherit (indirectly) from `RuntimeException`.

User defined exceptions are checked exceptions. They inherit (indirectly) from `Exception`.

User-defined exceptions

User-defined exceptions are mapped to C# classes that extend `CORBA.UserException` and are otherwise mapped just like the IDL `struct` type, including the generation of `Helper` classes.

If the exception is defined within a nested IDL scope (essentially within an interface or valuetype) then its C# class name is defined within a special namespace whose name is the name of the containing interface or valuetype with an NS suffix appended. Otherwise its C# class name is defined within the scope of the C# namespace that corresponds to the exception's enclosing IDL module.

This code sample shows the mapping of user-defined exceptions.

```
// Example.idl
module Example {
    exception AnException {
        string reason;
    };
};
// Example.cs
namespace Example {
   public sealed class AnExceptionHelper : CORBA.Streamable { ... }

   public sealed class AnException : CORBA.UserException {
    public string reason;
    public AnException() : base(Example.AnExceptionHelper.GetRepID()) {
      }
    public AnException(string reason) : this() {
        this.reason = reason;
      }
    public AnException (string _reason, string reason) :
        base(Example.AnExceptionHelper.GetRepId() + ' ' + _reason) {
        this.reason = reason;
      }
     override public string ToString() { . . . }
     override public GetHashCode() { ... }
     override public bool Equals(object o) { ... }
   }
}
```

System exceptions

The standard IDL system exceptions are mapped to final C# classes that extend `CORBA.SystemException` and provide access to the IDL major and minor exception code, as well as a string describing the reason for the exception. There are no public constructors for `CORBA.SystemException`; only classes that extend it can be instantiated.

The C# class name for each standard IDL exception is the same as its IDL name and is declared to be in the `CORBA` namespace. The default constructor supplies 0 for the minor code, `COMPLETED_NO` for the completion code, and the empty string ("") for the reason string. There is also a constructor which takes the reason and uses defaults for the other fields, as well as one which requires all three parameters to be specified.

Mapping for the Any type

The IDL type `Any` maps to the C# class `CORBA.Any`. This class has all the necessary methods to insert and extract instances of predefined types. If the extraction operations have a mismatched type, the `CORBA::BAD_OPERATION` exception is thrown.

In addition, insert and extract methods are defined to provide a high speed interface for use by portable stubs and skeletons. There is an insert and extract method defined for each primitive IDL type as well as a pair for a generic streamable to handle the case of nonprimitive IDL types.

The insert operations set the specified value and reset the `Any`'s type if necessary.

Setting the typecode via the `type()` accessor wipes out the value. An attempt to extract before the value is set will result in a `CORBA::BAD_OPERATION` exception being raised. This operation is provided primarily so that the type may be set properly for IDL `out` parameters.

Mapping for certain nested types

IDL allows type declarations nested within interfaces. C# does not allow classes to be nested within interfaces. Hence those IDL types that map to C# classes and that are declared within the scope of an interface must appear in a special "scope" namespace when mapped to C#.

IDL interfaces that contain these type declarations generate a scope namespace to contain the mapped C# class declarations. The scope namespace name is constructed by appending `NS` to the IDL type name.

This code sample shows the mapping for certain nested types.

```
// Example.idl
module Example {
    interface Foo {
        exception e1 {};
    };
};

// Example.cs
namespace Example
    public sealed class FooHelper { ... }
    public interface Foo : CORBA.Object, Example.FooOperations {
    }
    public interface FooOperations {
    }
    namespace FooNS {
        public sealed class e1Helper : CORBA.Streamable { ... }
        public sealed class e1 : CORBA.UserException { ... }
    }
    public class _FooStub : CORBAObjectIpml, Example.Foo { ... }
}
```

Mapping for TypeDef

IDL types that are mapped to simple C# types may not be subclassed in C#. Therefore, any typedefs that are type declarations for simple types are mapped to the original (mapped type) any where the typedef type appears. For simple types, `Helper` classes are generated for all typedefs.

Typedefs for non arrays and sequences are "unwound" to their original type until a simple IDL type or user-defined IDL type (of the non typedef variety) is encountered.

This code sample shows the mapping of a complex IDL typedef.

```
// Example.idl
module Example {
    struct EmployeeName {
        string firstName;
        string lastName;
    };
    typedef EmployeeName EmployeeRecord;
};

// Example.cs
namespace Example {
    public sealed class EmployeeNameHelper : CORBA.Streamable { ... }
    public sealed class EmployeeName { ... }
    public sealed class EmployeeRecordHelper { ... }
}
```

Java built-in type support

This chapter describes the Java to .NET mapping for built-in types supported in VisiBroker for .NET.

The following table describes the default Java package to .NET namespace mapping in VisiBroker for .NET.

Java package	.NET namespace	Refer to section
java	J2EE	
java.lang	J2EE.Lang	"java.lang"
java.io	J2EE.Io	"java.io"
java.math	J2EE.Math	"java.math"
java.net	J2EE.Net	"java.net"
java.rmi	J2EE.Rmi	"java.rmi"
java.sql	J2EE.Sql	"java.sql"
javax	J2EE	
javax.ejb	J2EE.Ejb	"javax.ejb"
javax.naming	J2EE.Naming	"javax.naming"
javax.rmi	J2EE.Rmi	"javax.rmi"
javax.transaction	J2EE.Transaction	"javax.transaction"
java.util	J2EE.Util	"java.util"

java.lang

Java type	.NET type
java.lang.Error	J2EE.Lang.Error
java.lang.Exception	J2EE.Lang.Exception
java.lang.Object	System.Object
java.lang.RuntimeException	J2EE.Lang.RuntimeException
java.lang.StringBuffer	System.Text.StringBuilder
java.lang.Throwable	J2EE.Lang.Throwable

Java type	.NET type
java.lang.Boolean	J2EE.Lang.Boolean
java.lang.Byte	J2EE.Lang.Byte
java.lang.Character	J2EE.Lang.Character
java.lang.Double	J2EE.Lang.Double
java.lang.Float	J2EE.Lang.Float
java.lang.Integer	J2EE.Lang.Integer
java.lang.Long	J2EE.Lang.Long
java.lang.Number	J2EE.Lang.Number
java.lang.Short	J2EE.Lang.Short

The following java.lang error and exception types are mapped to the J2EE.Lang namespace using the same type names.

AbstractMethodError	AssertionError
ClassCircularityError	ClassFormatError
Error	ExceptionInInitializerError
IllegalAccessError	IncompatibleClassChangeError
InstantiationError	InternalError
LinkageError	NoClassDefFoundError
NoSuchFieldError	NoSuchMethodError
OutOfMemoryError	StackOverflowError
ThreadDeath	UnknownError
UnsatisfiedLinkError	UnsupportedClassVersionError
VerifyError	VirtualMachineError

ArithmeticException	ArrayIndexOutOfBoundsException
ArrayStoreException	ClassCastException
IllegalArgumentException	IllegalMonitorStateException
IllegalStateException	IllegalThreadStateException
IndexOutOfBoundsException	NegativeArraySizeException
NullPointerException	NumberFormatException
RuntimeException	SecurityException
StringIndexOutOfBoundsException	UnsupportedOperationException

ClassNotFoundException	CloneNotSupportedException
IllegalAccessException	InstantiationException

InterruptedException	NoSuchFieldException
NoSuchMethodException	

java.io

The following java.io exception types are mapped to the J2EE.Io namespace using the same type names.

CharConversionException	EOFException
FileNotFoundException	IOException
InterruptedIOException	InvalidClassException
InvalidObjectException	NotActiveException
NotSerializableException	ObjectStreamException
OptionalDataException	StreamCorruptedException
SyncFailedException	UTFDataFormatException
UnsupportedEncodingException	WriteAbortedException

java.math

Java type	.NET type
java.math.BigDecimal	J2EE.Math.BigDecimal
java.math.BigInteger	J2EE.Math.BigInteger

java.net

Java type	.NET type
java.net.URL	System.Uri
java.net.URI	System.Uri
java.net.InetAddress	System.Net.IPHostEntry
java.net.Inet4Address	System.Net.IPHostEntry
java.net.SocketAddress	System.Net.EndPoint
java.net.InetSocketAddress	System.Net.IPEndPoint

The following java.net exception types are mapped to the J2EE.Net namespace using the same type names.

BindException	ConnectException
MalformedURLException	NoRouteToHostException
PortUnreachableException	ProtocolException
SocketException	SocketTimeoutException
URISyntaxException	UnknownHostException
UnknownServiceException	

java.rmi

Java type	.NET type
java.rmi.Remote	CORBA.Object

The following java.rmi exception types are mapped to the J2EE.Rmi namespace using the same type names.

RMISecurityException	

AccessException	AlreadyBoundException
ConnectException	ConnectIOException
MarshalException	NoSuchObjectException
NotBoundException	RemoteException
ServerError	ServerException
ServerRuntimeException	StubNotFoundException
UnexpectedException	UnknownHostException
UnmarshalException	

java.sql

Java type	.NET type
java.sql.Date	System.DateTime
java.sql.Time	System.DateTime
java.sql.Timestamp	System.DateTime

The following java.sql exception types are mapped to the J2EE.Sql namespace using the same type names.

BatchUpdateException	DataTruncation
SQLException	SQLWarning

javax.ejb

Java type	.NET type
javax.ejb.EJBHome	J2EE.Ejb.EJBHome
javax.ejb.EJBMetaData	J2EE.Ejb.EJBMetaData
javax.ejb.EJBObject	J2EE.Ejb.EJBObject
javax.ejb.Handle	J2EE.Ejb.Handle
javax.ejb.HomeHandle	J2EE.Ejb.HomeHandle

The following javax.ejb exception types are mapped to the J2EE.Ejb namespace using the same type names.

AccessLocalException	CreateException
DuplicateKeyException	EJBException
FinderException	NoSuchEntityException
ObjectNotFoundException	RemoveException
TransactionRequiredLocalException	TransactionRolledbackLocalException

javax.naming

Java type	.NET type
javax.naming.Binding	J2EE.Naming.Binding
javax.naming.Context	J2EE.Naming.Context
javax.naming.InitialContext	J2EE.Naming.InitialContext
javax.naming.NameClassPair	J2EE.Naming.NameClassPair

The following javax.naming exception types are mapped to the J2EE.Naming namespace using the same type names.

AuthenticationException	AuthenticationNotSupportedException
CannotProceedException	CommunicationException
ConfigurationException	ContextNotEmptyException
InsufficientResourcesException	InterruptedNamingException
InvalidNameException	LimitExceededException
LinkException	LinkLoopException
MalformedLinkException	NameAlreadyBoundException
NameNotFoundException	NamingException
NamingSecurityException	NoInitialContextException
NoPermissionException	NotContextException
OperationNotSupportedException	PartialResultException
ReferralException	ServiceUnavailableException
SizeLimitExceededException	TimeLimitExceededException

javax.rmi

Java type	.NET type
javax.rmi.PortableRemoteObject	J2EE.Rmi.PortableRemoteObject

javax.transaction

The following javax.transaction exception types are mapped to the J2EE.Transaction namespace using the same type names.

HeuristicCommitException	HeuristicMixedException
HeuristicRollbackException	InvalidTransactionException
NotSupportedException	RollbackException
SystemException	TransactionRequiredException
TransactionRolledbackException	

java.util

Java type	.NET type
java.util.Calendar	J2EE.Util.Calendar
java.util.Date	System.DateTime
java.util.GregorianCalendar	J2EE.Util.GregorianCalendar
java.util.Locale	System.Globalization.CultureInfo
java.util.Random[1]	System.Random
java.util.TimeZone	System.TimeZone
java.util.SimpleTimeZone	System.TimeZone
sun.util.calendar.ZoneInfo[2]	System.TimeZone

1. See "Interoperability property" for information on using this type.
2. `sun.util.calendar.ZoneInfo` is intended to be an internal Sun implementation class, but it shows up in remote procedure calls in certain situations.

Java type	.NET type
java.util.Comparator	System.Collections.IComparer
java.util.Iterator	System.Collections.IEnumerator
java.util.ListIterator	System.Collections.IEnumerator

Java type	.NET type
java.util.Collection	System.Collections.ICollection
java.util.List	System.Collections.IList
java.util.Map	System.Collections.IDictionary
java.util.Set	System.Collections.ICollection
java.util.SortedMap	System.Collections.IDictionary
java.util.SortedSet	System.Collections.ICollection

Java type	.NET type
java.util.AbstractCollection	System.Collections.ICollection
java.util.AbstractList	System.Collections.IList
java.util.AbstractMap	System.Collections.IDictionary
java.util.AbstractSequentialList	System.Collections.IList
java.util.AbstractSet	System.Collections.ICollection
java.util.Dictionary	System.Collections.IDictionary

Java type	.NET type
java.util.ArrayList	System.Collections.ArrayList
java.util.BitSet	System.Collections.BitArray
java.util.HashMap	System.Collections.Hashtable
java.util.HashSet	System.Collections.ArrayList
java.util.Hashtable	System.Collections.Hashtable
java.util.IdentityHashMap	System.Collections.Hashtable
java.util.LinkedHashMap	System.Collections.Hashtable
java.util.LinkedHashSet	System.Collections.ArrayList
java.util.LinkedList	System.Collections.ArrayList
java.util.Properties	System.Collections.Specialized.StringDictionary
java.util.Stack	System.Collections.Stack
java.util.TreeMap	System.Collections.Hashtable
java.util.TreeSet	System.Collections.ArrayList
java.util.Vector	System.Collections.ArrayList

Java type	.NET type
java.util.Arrays$ArrayList	System.Collections.ArrayList
java.util.Collections$CopiesList	System.Collections.ArrayList
java.util.Collections$SingletonList	System.Collections.ArrayList
java.util.Collections$SingletonMap	System.Collections.Hashtable
java.util.Collections$SingletonSet	System.Collections.ArrayList
java.util.Collections$SynchronizedCollection	System.Collections.ICollection
java.util.Collections$SynchronizedList	System.Collections.IList
java.util.Collections$SynchronizedMap	System.Collections.IDictionary
java.util.Collections$SynchronizedRandomAccessList	System.Collections.IList
java.util.Collections$SynchronizedSet	System.Collections.ICollection
java.util.Collections$SynchronizedSortedMap	System.Collections.IDictionary
java.util.Collections$SynchronizedSortedSet	System.Collections.ICollection

Java type	.NET type
java.util.Collections$UnmodifiableCollection	System.Collections.ICollection
java.util.Collections$UnmodifiableList	System.Collections.IList
java.util.Collections$UnmodifiableMap	System.Collections.IDictionary
java.util.Collections$UnmodifiableSet	System.Collections.ICollection
java.util.Collections$UnmodifiableSortedMap	System.Collections.IDictionary
java.util.Collections$UnmodifiableSortedSet	System.Collections.ICollection
java.util.TreeMap$SubMap	System.Collections.Hashtable

The following java.util exception types are mapped to the J2EE.Util namespace using the same type names.

ConcurrentModificationException	EmptyStackException
MissingResourceException	NoSuchElementException

TooManyListenersException	

Application server support

The following table describes application server-specific type mappings that are included in VisiBroker for .NET.

Application server	Java type	.NET type
Borland AppServer	com.inprise.ejb.iterator. CustomVector	System.Collections.ArrayList

Index

Symbols

.NET Framework class library 4
.NET Remoting 4, 6, 9, 12
    example 9
    extension 12

A

Abstract interfaces 165
activation, client 14
activation, server 14
Any type mapping 167
application server support 178
arrays 162
    mapping 159
ASP.NET 1, 4, 34

B

basic IDL types 157
boolean type mapping 157
bootstrapping 13
Borland AppServer 178
Borland.Janeva.Private 34
Borland.Janeva.Runtime 33, 34
Borland.Janeva.Services 33, 34
borland.slip 35
building VisiBroker for .NET applications 31
built-in types, Java 171

C

C#
    generating code from IDL file 149
    null 157
Callback interface 38
callbacks, adding to clients 41
cast 10
channel, Remoting 13
char type mapping 157
client activation 14
    example 14
client.slip 35
ClientRequestInterceptor 80
Codec 82
CodecFactory 82
collision rule 156
command line 32, 33
commands
    idl2cs 149
    idl2csj 149
common intermediate language 2, 4
common language runtime 3
common language system 2
common type system 2
compiler options 149
compiler overview 2
complex data types 3
configuration file 10, 12
    licensing 35
configuring properties 17
    command-line 17

configuration file 19
    programmatically 18
    property descriptions 19
conflict resolution 156
constants mapping 158
constructed data types 76
constructed types mapping 159
contexts 3
CORBA
    example 11
    naming service 11
    overview 5
corbaloc URL scheme 13
corbaname URL scheme 13
CosTransactions 108
Current
    interface 82
Current object reference 107
custom marshaling 45, 54

D

data types 3
    constructed 76
    traversing the components 76
declarative activation 12
deploying VisiBroker for .NET applications 31, 33
deployment license 34
developer tools overview 2
developing Remoting server 37
development process 9
dynamically managed types 75
DynAny
    access and initializing 76
    constructed data types 76
    creating 76
    CurrentComponent method 76
    interface 75
    Next method 76
    Rewind method 76
    Seek method 76
    types 75
    usage restrictions 75
DynArray data type 77
DynEnum data type 76
DynSequence data type 77
DynStruct data type 77
DynUnion data type 77

E

EAR 31, 32, 151
effective policies 61
EJB interfaces 151
EJBHome object 10
embedded resource licensing 34
Enterprise JavaBeans overview 4
enums mapping 159
exceptions
    mapping 165
    system 166
    user-defined 166
extract method, Helper classes 163

Index

W

words, reserved 156
wstring mapping 158

X

XML 6
   configuration file 10, 12
   license configuration 35