IBM WebSphere Application Server Enterprise,
Version 5

Common Object Request Broker Architecture (CORBA)

Note

Before using this information and the product it supports, read the general information under "Notices" on page 173.

Contents

Implementing CORBA applications

The CORBA support provided in IBM® WebSphere® Application Server enables the use of CORBA interfaces between a server object providing a service and a client using the service. In practice, this means that a CORBA client can access a WebSphere CORBA C++ server and a WebSphere EJB server. For a detailed description of the CORBA environment and elements, refer to "CORBA concepts" on page 78.

In addition, IBM WebSphere Application Server provides a basic CORBA environment that can ″bootstrap″ into the J2EE name space and invoke J2EE transactions. However, it does not provide its own naming and transaction services. Therefore, a CORBA C++ client or server relies on the J2EE environment to provide these services.

WebSphere CORBA support can be divided into two categories:

WebSphere to WebSphere CORBA support
: Enables creation of CORBA client and server applications within the IBM WebSphere Application Server environment. You can use the CORBA C++ SDK to build a lightweight WebSphere CORBA server to use with new or existing C and C++ programs. You also can use the SDK to build a WebSphere CORBA C++ client to use with a WebSphere EJB server or WebSphere CORBA C++ server.

WebSphere to other ORB support
: Enables other applications that are based on CORBA Object Request Brokers (ORBs) to interoperate with WebSphere. This enables these applications to leverage WebSphere-supported open technologies, such as Java™ ServerPages, XML, Java Servlets, and enterprise beans.

This documentation focuses on WebSphere to WebSphere, the first category of CORBA support, which encompasses the following scenarios:

WebSphere CORBA C++ client to a WebSphere EJB server
: Enables a CORBA C++ client to access enterprise beans hosted by a WebSphere EJB server. For more information, see "CORBA client to WebSphere EJB server" on page 19.

WebSphere CORBA C++ client to a WebSphere CORBA C++ server
: Enables a WebSphere CORBA C++ client to access a CORBA server implementation object hosted by a C++ CORBA server within the IBM WebSphere Application Server environment. This CORBA support provides the basic CORBA building blocks to create CORBA C++ client and server applications within WebSphere.

In implementing CORBA applications, refer to the following topics:

- "Developing a CORBA C++ client" on page 2
- "Developing a CORBA C++ server" on page 20
- "Managing CORBA applications" on page 37

For comprehensive programming details, refer to the following topics:

- "CORBA programming model" on page 76

- CORBA programming reference (located in the on-line InfoCenter at http://www-3.ibm.com/software/webservers/appserv/infocenter.html)

For more information about CORBA, see CORBA: Resources for learning (located in the InfoCenter at http://www-3.ibm.com/software/webservers/appserv/infocenter.html)

Developing a CORBA C++ client

Use this task to develop a CORBA C++ client. This task generates the client-side usage bindings needed by CORBA C++ client programs to access an object class (enterprise bean or CORBA servant object) hosted by an application server.

Steps for this task

1. Create the interface definition language (IDL) files that specify the public interface to the server implementation object class. If you want the client to access a CORBA server implementation class, create the IDL file as part of the procedure to define the servant implementation as described in Defining the interface for a servant implementation (servant.idl). If you want the client to access an enterprise bean, you can create the IDL file from the bean class, as described in "Creating IDL files for an enterprise bean" on page 3.
2. Use the **idlc** command to emit the client-side usage bindings from the IDL, specifying the `-suc:hh` option.

   If you want the client to access a CORBA server implementation, emit the client-side usage bindings when you compile the servant.idl file, as described in "Compiling a CORBA server implementation class IDL (using idlc)" on page 23. If you want the client to access an enterprise bean, you can use the same procedure with the IDL file created from the bean class.

   For example, to emit the client-side bindings from the Hello.idl file and use a command line to change to the directory that contains the IDL file, type the following command:

   ```
   idlc -suc:hh Hello.idl
   ```

   When the specified idl file is compiled successfully, the **idlc** command creates the binding files and returns a value of zero. For example, using the previous example **idlc** command, the following binding files are created: Hello.hh and Hello_C.cpp
3. Create the main code for the client program, as described in "Creating the CORBA client main code (client.cpp)" on page 3.
4. Compile and link the C++ client main program as described in "Building a CORBA C++ client" on page 9.

What to do next

For additional information related to Developing a CORBA C++ client, see the following topics:

- "Managing the EJB home" on page 10
- "Client bootstrapping operation" on page 11
- "CORBA client exception handling" on page 16
- "Coding tips for CORBA memory management" on page 18
- "CORBA client to WebSphere EJB server" on page 19

Creating IDL files for an enterprise bean

Use this task to generate the interface definition language (IDL) files that specify the interface to an enterprise bean. You then can use the IDL to create client-side usage bindings for CORBA clients to use the EJB's interface. Complete this task only if you are developing a CORBA client that needs to access an enterprise bean.

Steps for this task

1. Develop the enterprise bean.
2. Ensure that the JAR file that contains the EJB class can be accessed by the **rmic** command. The JAR file should be in the system classpath.

   To generate IDL files for the com.ibm.ejb.samples.hello.Hello enterprise bean, you might use the following command:

   ```
   rmic -idl com.ibm.ejb.samples.hello.Hello com.ibm.ejb.samples.hello.HelloHome
   ```
3. Use the Java **rmic -idl** command to generate IDL files from the enterprise bean's remote and home interfaces.

   This step results in the `class.idl` and `classHome.idl` files. For example, the previous **rmic** command for the Hello enterprise bean class created the following idl files: Hello.idl and HelloHome.idl.

What to do next

You can use the IDL file to create the client-side usage bindings needed by a CORBA client, as described in "Developing a CORBA C++ client" on page 2.

Creating the CORBA client main code (client.cpp)

Use this task to create the main code for a CORBA client, to locate a servant object hosted by a CORBA server, and to call methods on the server object. The client's main() function performs the following tasks:

1. Initializes the client environment
2. Gets a pointer to the root naming context
3. Accesses the servant object
4. Calls methods on the servant object
5. Stops the client and releases resources used

Steps for this task

1. Create a source file, *client*.cpp, where *client* is the name of the client program.
2. Edit the client source file, *client*.cpp, to add appropriate code to implement the client. To do this, complete the following steps:
   a. Add the necessary include statements, as described in "Adding include statements" on page 4.
   b. Add the main() function in the following form:

   ```
   main(int argc, char *argv[])
   {
     int rc;
     ::CORBA::Object_ptr objPtr;
     ::CosNaming::NamingContext_var rootNameContext = NULL;
     Servant_var liptr = NULL;

     exit( 0 );
   }
   ```
3. Add code to initialize the client environment as described in "Initializing the client environment" on page 4.

4. Add code to get a pointer to the root naming context as described in “Getting a pointer to the root naming context” on page 5.
5. Add code to access the servant object that has already been created by the server as described in “Accessing the servant object” on page 7.
6. Add code to call methods on the servant object as described in “Invoking methods on the servant object” on page 8.
7. Add code to shutdown the client and release resources used as described in “Shutting down the client and releasing resources used” on page 9.

Adding include statements

Use this task to add the necessary include statements to the source file for a CORBA client main code.

Add the following include statements:

```
#include "servant.hh"
#include <CosNaming.hh>
```

where:

servant.**hh**
: Specifies the name of the client-side usage bindings file for the server implementation class, *servant*. This file is created when the server implementation class IDL is compiled, as described in “Compiling a CORBA server implementation class IDL (using idlc)” on page 23.

CosNaming.hh
: Specifies the header file for the COSNaming functions.

What to do next

You also can add the code for the functions needed in the client main code as described in “Creating the CORBA client main code (client.cpp)” on page 3.

Initializing the client environment

Use this task to add code that initializes the Object Request Broker (ORB) for use by the client program.

To do this, edit the client source file, client.cpp, and add the following code to the main() function:

```
int main ( int argc, char *argv[] )
{
   CORBA::ORB_ptr orbPtr = CORBA::ORB_init ( argc, argv, "DSOM" );
   if ( CORBA::is_nil(orbPtr) )
   {
      cerr << "Error initializing the ORB!" << endl;
      return 1;
   }

   .
   .
   .

}
```

Results

This task adds code to initialize the client environment for a CORBA client.

What to do next

You need to add code to the client source file to enable the client to access naming contexts as described in "Accessing naming contexts" on page 30.

Client environment initialization of the C++ ORB: One of the first tasks for a CORBA C++ application after startup is to initialize the Object Request Broker (ORB) by calling the `CORBA::ORB_init()` method. If necessary, this method creates a new instance of the ORB. For example, the following code fragment initializes the ORB:

```
CORBA::ORB_ptr op;
op = ::CORBA::ORB_init(argc, argv, "DSOM");
```

where argc and argv refer to the properties specified in the command used to start the server.

During the initialization of the ORB, the resolution of initial references can be configured. Refer to the `CORBA::ORB_init` method for more information on how the optional arguments, ORBInitRef and ORBDefaultInitRef are used to configure how the ORB resolves initial references. This affects how the ORB processes the `CORBA::resolve_initial_references` method.

The following is a command line invocation example:

```
myApp -ORBInitRef NameServer=file://c:/temp/namesvr.ref -ORBDefaultInitRef corbaloc::myHost.myOrg

// Where myApp is a C++ program that passes
// the command line args to ORB_init.
//
#include "corba.h"
...
int main(int argc, char *argv[])
{
   //
   // Initialize the ORB and obtain a pointer to it
   //
   CORBA::ORB_ptr p = CORBA::ORB_init(argc, argv, "DSOM");
    ...
   CORBA::Object_ptr obj = p->resolve_initial_references ( "NameService" );
   ...
```

In the previous example, the resolve_initial_references invocation first searches for a Naming Service object reference in the file, `c:/temp/namesvr.ref`. If unsuccessful, the host `myHost.myOrg` is contacted to find the Naming Service.

Getting a pointer to the root naming context

Before a CORBA server can create and make available a servant object, it must have a logical name space for the servant object to exist in. This logical name space is a naming context for servant objects. The server can create a new naming context within any location within the root naming context. For example, a server called *servant*Server might create a new naming context called *servant*Context into which the server binds the servant object. Optionally, this context might be located within a domain context, which in turn is located within the root naming context. (You can create a servant context with only the root naming context as its parent or with one or more intermediary parent contexts.)

Use this task to get a pointer to the root naming context. This task adds a "get name context" method (for example, get_name_context) to the source file for a CORBA client. This method is used to access the naming service and return a pointer to the root naming context. The method performs the following actions after the client environment has been initialized:

1. Receives a pointer to the naming service
2. Receives a pointer to the root naming context

Steps for this task

1. Add the get_name_context() method as shown in the following code extract:

```
// This method accesses the Name Service and then gets
// the root naming context, which it returns;
// the WSLogger context.

::CosNaming::NamingContext_ptr get_naming_context()
{
  ::CosNaming::NamingContext_ptr rootNameContext = NULL;
  ::CORBA::Object_ptr objPtr;

  // Get access to the Naming Service.
  try
  {
    objPtr = op->resolve_initial_references( "NameService" );
  }

  // catch exceptions ...

  if ( objPtr == NULL )
  {
    cerr << "ERROR: resolve_initial_references returned NULL" << endl;
    release_resources( op );
    return( NULL );
  }
  else
    cout << "resolve_initial_references returned = " << objPtr << endl;

  // Get the root naming context.
  rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
  if ( ::CORBA::is_nil( rootNameContext ) )
  {
    cerr << "ERROR: rootNameContext narrowed to nil" << endl;
    release_resources( op );
    return( NULL );
  }
  else
    cout << "rootNameContext = " << rootNameContext << endl;
  // Release the temporary pointer.
  ::CORBA::release(objPtr);

  return( rootNameContext );
}
```

   where

   rootNameContext
   is the pointer to the root naming context.

   This code receives a pointer to the naming service, narrows the pointer object to the appropriate object type, assigns it to the new pointer object called `rootNameContext`, performs some checks, and releases the original pointer object, `objPtr`.

2. Add a statement to the main method to call the new method as shown in bold in the following code extract:

```
...
  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    exit( rc );

  // Get the root naming context.
  rootNameContext = get_naming_context();
  if ( ::CORBA::is_nil( rootNameContext ) )
    exit( -1 );
```

Results

This step returns a pointer object, `rootNameContext`, to the root naming context.

What to do next

Add code to the client source file to access the servant object created by the server as described in "Accessing the servant object".

Accessing the servant object

Use this task to add code to the source file for a CORBA client and get access to the servant object that is created by the CORBA server and bound into the name space. The client code gets access to the servant object by creating a ::CosNaming::Name that specifies the full name of the object from the root naming context.

For the example code in this task, the CORBA server created the servant object called *servant*Object1 in a new context, *servant*Context. *servant*Context is bound to the domain naming context, which is bound to the root naming context. Therefore, the full name for the servant object, from the root naming context, is domain.*servant*Context.*servant*Object1.

Steps for this task

1. Edit the client source file *client*.cpp
2. In this client source file, add code to the main() function to get a new ::CosNaming::Name for the servant object and to look up the servant object with that name in the name space. For an example, see the following code:

```
// Get the root naming context.
  rootNameContext = get_naming_context();
  if ( ::CORBA::is_nil( rootNameContext ) )
    exit( -1 );

  // Find the servant_Impl created by the server. Look up the
  // object using the complex name of domain.servantContext.servantObject1,
  // which is its full name from the root naming context, as created
  // by the server.
  //Note: The actual complex name may vary.
  //Run the command WAS_HOME/bin/dumpNameSpace to identify
  //the exact name of your servant object.
  try
  {
    // Create a new ::CosNaming::Name to pass to resolve().
    // Construct it as the full three-part complex name.
    ::CosNaming::Name servantName;
    servantName.length( 3 );
    servantName[0].id = ::CORBA::string_dup( "domain" );
    servantName[0].kind = ::CORBA::string_dup("");
    servantName[1].id = ::CORBA::string_dup( "servantContext" );
    servantName[1].kind = ::CORBA::string_dup( "" );
    servantName[2].id = ::CORBA::string_dup( "servantObject1" );
    servantName[2].kind = ::CORBA::string_dup( "" );
    ::CORBA::Object_ptr objPtr = rootNameContext->resolve(servantName);
  }

  // catch exceptions ...
```

Results

This task adds code that enables a CORBA client to find the specified servant object (created by a CORBA server) in the system name space.

What to do next

Add code to the client source file to enable the client to call methods on the servant object as described in "Invoking methods on the servant object". For related information on accessing the servant object, see "Servant object access".

Servant object access: To be able to locate a servant object somewhere in a CORBA environment, a client must know the object reference that uniquely identifies the target object.

When an object is created, it is assigned an object reference, which can be bound with a name in the naming service. Any client (or any other object) with access to the naming service can use the associated name to retrieve the object reference.

Object references are bound into the naming service relative to the root naming context. After a client has located the root naming context, it can use the standard CosNaming interface to navigate the name space and retrieve the object reference associated with any name as shown in the following example:

```
// Create a new ::CosNaming::Name to pass to resolve().
// Construct it as the full three-part complex name.
::CosNaming::Name loggerName;
loggerName.length( 3 );
loggerName[0].id = ::CORBA::string_dup( "persistent" );
loggerName[0].kind = ::CORBA::string_dup("");
loggerName[1].id = ::CORBA::string_dup( "WSLoggerContext" );
loggerName[1].kind = ::CORBA::string_dup( "" );
loggerName[2].id = ::CORBA::string_dup( "WSLoggerObject1" );
loggerName[2].kind = ::CORBA::string_dup( "" );
::CORBA::Object_ptr objPtr = rootNameContext->resolve( loggerName );
liptr = WSLogger::_narrow( objPtr);
```

If the client bootstrapping operation does not establish contact with a remote naming service, you can use the alternative strategies to retrieve the IOR of a remote object, as outlined in the topic, "Strategies for retrieving the IOR of a remote object" on page 15.

Invoking methods on the servant object

After a CORBA client has got access to a servant object, the client can call methods on the servant object. The methods depend entirely on the business functionality of the client, but have the following general syntax:

```
servant_pointer->method_name( arguments );
```

For example, the following code calls the getFileName() method on the object identified by the liptr object reference and gets the value of the FileName attribute:

```
...
liptr = servant::_narrow( objPtr);
...
cout << "Logging to file " << liptr->getFileName() << endl;
liptr->setFileName( argv[1] );
cout << "Now logging to file " << liptr->getFileName() << endl;
```

What to do next

This code forms the main business functionality of the client. When you have added the method calls needed to your client code, the next stage is to add code to shut down the client and release the resources that it uses as described in "Shutting down the client and releasing resources used" on page 9.

Shutting down the client and releasing resources used

Use this task to create code for a CORBA client, shut down the client, and release the resources that it used.

To shut down a CORBA client, add the following code to the main() function in the *client*.cpp client source file:

```
int main ( int argc, char *argv[] )
{
   CORBA::ORB_ptr orbPtr = CORBA::ORB_init ( argc, argv, "DSOM" );

   .
   .
   .

   // Release the ORB object.
   CORBA::release ( orbPtr );

   return 0;
}
```

Results

This task adds code that shuts down a CORBA client and releases the resources that it used.

Building a CORBA C++ client

This topic provides an overview of how to build the code for a CORBA C++ client. The actual steps that you need to complete depend on your development environment.

For example, if you are using the Microsoft® Visual C++ 6.0 compiler on Microsoft Windows NT® to build a CORBA C++ client, you can use the following commands:

Steps for this task

1. At a command line, type the following command:

   ```
   cl /c /GX /DEXCL_IRTC /DSOMCBNOLOCALINCLUDES -IWAS_HOME\include client.cpp
   ```

   where:

   WAS_HOME
   : is the directory into which IBM WebSphere Application Server is installed.

   client
   : is the name of the C++ client main code file.

2. At a command line, type the following command:

   ```
   cl /c /GX /DEXCL_IRTC /DSOMCBNOLOCALINCLUDES -IWAS_HOME\include servant_C.cpp
   ```

   where

   servant
   : is the name of the server implementation object (servant) that the client accesses.

3. At a command line, type the following command:

   ```
   link client.obj servant_C.obj /OUT:client.exe wasororm.lib wasosa1m.lib
   ```

What to do next

For more examples of building CORBA C++ client code (on several platforms) for IBM WebSphere Application Server, see the Samples Gallery, which is installed with IBM WebSphere Application Server.

Managing the EJB home

To access an enterprise bean, a CORBA C++ client needs to locate the EJB home. The client obtains the root naming context from the naming service and uses it to locate the EJB home. Obtaining the naming context can be done explicitly by the client or implicitly by the Object Request Broker's (ORB's) string_to_object method.

Steps for this task

1. Identify the EJB home JNDI name for the enterprise bean's full path.

   In WebSphere, the JNDI name for a bean is mapped to the home class for that bean. The JNDI name is specified in the ibm-ejb-jar-bnd.xmi file generated for the deployed bean's JAR file. If you run the command WAS_HOME/bin/dumpNameSpace, you can see the mapping of the JNDI name to the corresponding Java class. For an enterprise bean, the JNDI name (top)/ejbhome is mapped to the home class. The enterprise bean is located in (top), which is the IBM WebSphere Application Server equivalent to the server's root.

   If the EJB home is located on a known server, such as server1, the three components of (top) are "cell", "server", and "server1". In this case, (top) can be specified as cell/servers/server1.

   For example, if the EJB home for the valuetype sample class com.ibm.websphere.vtlib.sample.Person is on server1, the full EJB home JNDI name is cell/servers/server1/com/ibm/websphere/vtlib/sample/Person.

   What happens when you perform this step.

2. Map the JNDI name to an object using one of two methods

   Map the EJB home JNDI name using a URL and string_to_object

   For the URL method of mapping the name, form a URL representing the EJB home JNDI name and pass it on a call to the ORB's string_to_object method. The URL should be a corbaname that contains the string-modified name of the EJB home. For the valuetype sample, the person bean might be located as follows when the person home is deployed on server1:

   ```
   char * home_url = "corbaname::localhost/NameService#cell/servers
   /server1/com/ibm/websphere/vtlib/sample/Person";
   ::CORBA::Object_var homeObj = NULL;
   homeObj = orb->string_to_object( home_url );
   if (CORBA::is_nil(homeObj))
        { // handle error }
   ```

   Note: The first line of the previous example wrapped onto a seond line due to the width of the page.

   Map the EJB home JNDI name by invoking resolve on the naming context

   For the resolve method of mapping the name, create a CosNaming::Name object and initialize it to identify the person home. Obtain the naming context and call its resolve method. The following code creates a CosNaming Name for the bean's full path:

   ```
   // Obtain the naming context.  (where op is a valid ORB pointer)
   //
   CORBA::Object * objPtr = op->resolve_initial_references( "NameService" );
   rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
   if (CORBA::is_nil( rootNameContext ))
     { // handle error }

   // Create a CosNaming Name for the bean's full path
   //
   ```

```
::CosNaming::Name *ejbName = new ::CosNaming::Name;
ejbName->length( 9 );
(*ejbName)[0].id = ::CORBA::string_dup( "cell" );
(*ejbName)[0].kind = ::CORBA::string_dup( "" );
(*ejbName)[1].id = ::CORBA::string_dup( "servers" );
(*ejbName)[1].kind = ::CORBA::string_dup( "" );
(*ejbName)[2].id = ::CORBA::string_dup( "server1" );
(*ejbName)[2].kind = ::CORBA::string_dup( "" );
(*ejbName)[3].id = ::CORBA::string_dup( "com" );
(*ejbName)[3].kind = ::CORBA::string_dup( "" );
(*ejbName)[4].id = ::CORBA::string_dup( "ibm" );
(*ejbName)[4].kind = ::CORBA::string_dup( "" );
(*ejbName)[5].id = ::CORBA::string_dup( "websphere" );
(*ejbName)[5].kind = ::CORBA::string_dup( "" );
(*ejbName)[6].id = ::CORBA::string_dup( "vtlib" );
(*ejbName)[6].kind = ::CORBA::string_dup( "" );
(*ejbName)[7].id = ::CORBA::string_dup( "sample" );
(*ejbName)[7].kind = ::CORBA::string_dup( "" );
(*ejbName)[8].id = ::CORBA::string_dup( "Person" );
(*ejbName)[8].kind = ::CORBA::string_dup( "" );

// Invoke resolve
//
::CORBA::Object_var homeObj = NULL;
homeObj = rootNameContext->resolve( ejbName );
if (CORBA::is_nil( homeObj ))
  { // handle error }
```

3. Narrow to a specific EJB Home

   Finally, narrow the object returned from either resolve or string_to_object. Use the PersonHome's _narrow method:

   ```
   ::com::ibm::websphere::vtlib::sample::PersonHome_ptr personHome = NULL;
   personHome = ::com::ibm::websphere::vtlib::sample::PersonHome::_narrow(homeObj);
   if (CORBA::is_nil(personHome))
      { // handle error }
   ```

Usage scenario

For an example of how to locate an EJB home, see the samples article ″Tutorial: Creating a user-defined C++ client that uses an EJB″ in the Samples Gallery, which is installed with IBM WebSphere Application Server.

What to do next

The object pointer to the EJB home can be used to create an enterprise bean object. For example:

```
ejbPtr = ejbHomePtr->create();
```

When the EJB object is created successfully, any of its methods can be called. For example:

```
msg = ejbPtr->message();
```

Client bootstrapping operation

The naming service can be used to manage a directory of objects and to map the name of each object to its associated object reference. To locate a server object somewhere in a CORBA environment, a client can locate the naming service and then use a name to retrieve an associated object reference from the naming service.

The location of the naming server that provides the naming service and the number of the port that it uses to communicate with clients and servers are specified by the WebSphere run-time properties. The values that you specify for the run-time properties must match the equivalent settings used to configure the IBM WebSphere Application Server.

Object references are bound into the naming service relative to the root naming context.

When a client is started, it uses a ″bootstrapping″ operation to locate the naming service and obtain the root naming context, as follows:

1. The client calls the `CORBA::ORB::resolve_initial_references(″NameService″)` method, which returns a CORBA::Object.
2. Before the client can use the returned reference as a naming context, the client must narrow the object to the desired class.

The following is an example of this ″bootstrapping″ procedure:

```
objPtr = op->resolve_initial_references( "NameService" );
rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
```

Bootstrapping of the Naming Server is one example of using ″Initial References″.

If the client bootstrapping operation does not establish contact with a remote naming service, you can use alternative strategies to retrieve the Interoperable Object Reference (IOR) of the naming service, as outlined in the topic “Initial references”.

Initial references

An initial reference is a well-known object reference associated with an identifier. CORBA provides mechanisms to configure, register, list, and get (or resolve) initial references. Obtaining an initial reference of the Naming Service also is called ″bootstrapping″.

A reference can refer to a local object or to a remote object.

Initial references can be obtained from the Object Request Broker (ORB) by calling ORB::resolve_initial_references(). Initial object references returned by this call are of the CORBA::Object class and generally must be narrowed to an appropriate interface to be useful.

For example:

```
CORBA::Object * objPtr;
objPtr = op->resolve_initial_references( "NameService" );
CosNaming::NamingContext * rootNameContext;
rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
```

A list of the initial references that are available from the ORB can be obtained by calling ORB::list_initial_references().

There are several ways an identifier and an initial reference are known to the ORB:

- CORBA specifies some well-known initial references that represent CORBA services, such as the Naming Service and the Interface Repository. Their identifiers are ″NameService″ and ″InterfaceRepository″. These are defined in the ORB, by default.
- An application can register the reference of a service with the ORB by calling ORB::register_initial_reference().
- An application can define objects that inherit from the current interface. However, doing so causes a local-only singleton object to be registered with the ORB.

Resolution of initial references occurs when ORB::resolve_initial_references() is invoked. The ORB follows an ordered search for the reference:

- The ORB checks if a reference for the identifier is defined using the ORBInitRef argument of CORBA::ORB_init().

- If the ORBDefaultInitRef argument was specified when invoking CORBA::ORB_init(), the ORB checks if the reference can be resolved by appending the identifier name to the ORBDefaultInitRef argument.

  Note: This argument might be a corbaloc URL string that specifies multiple host port pairs. Resolution is attempted with each host port pair in order of their specification. This operation follows CORBA's Interoperable Naming Specification (INS).
- The ORB checks if a reference for the identifier is registerd by the application using the CORBA::register_initial_reference() method.
- The ORB checks if the identifier is one of the predefined or current references.

The C++ ORB supports CORBA's Interoperable Naming Specification (INS) and a different method that is an IBM extension. For backward compatibility, the ORB tries the latter if the INS mode does not succeed.

If an initial reference cannot be obtained using the framework described previously, there are alternatives for obtaining a reference. For more information, see the topic, "Strategies for retrieving the IOR of a remote object" on page 15.

Object URLs

Object URLs are intended to provide a human-readable string form of an object reference. With a syntax similar to internet URLs, they offer an easier way to specify an object than the string-modified interoperable object reference (IOR). Because URLs are "universal", an IOR sometimes might be considered an object URL. However, for the purposes of this discussion, an IOR is not an object URL. IORs are not easy for humans to read.

Object URLs are defined primarily in the context of enabling bootstrapping of the Naming Service, but can be used to reference any initial service. They also can be used to obtain an object by invoking the CORBA::string_to_object method.

The syntax of an object URL supports three basic formats: corbaloc, corbaname, and file. These formats are called URL schemes. The corbaloc URL scheme uses either IIOP or resolve_initial_references style addressing. The corbaname URL scheme extends the corbaloc format to specify an entry within a Naming Service. The file URL scheme identifies a file containing another object reference. For more detail on each scheme, see the following topics:

- corbaloc
- corbaname
- file

Unlike IOR object references, converting an object URL usually requires the client ORB to contact the server where the implementation (servant) object resides. The ORB must retrieve additional information from the server because object URLs are much simpler than IORs and contain insufficient information to fully describe the object.

Corbaloc URL scheme: The corbaloc URL scheme, one of the three CORBA object URL schemes, uses either IIOP or resolve_initial_references (RIR) style addressing.

The IIOP style allows specification of host and port information, which identifies a server or agent that can be resolved to provide the object. Alternatively, corbaloc URLs can indicate that an object can be contacted by CORBA::resolve_initial_references.

Examples of corbaloc URL scheme strings:

```
corbaloc:iiop:1.2@xyz_host.net/NameService
corbaloc:rir:/NameService
corbaloc::ABC_host.com/ProductZ/TradingService
corbaloc::primary.com:110,:1.2@backup.com:120/Dev/NameService
```

A corbaloc URL scheme begins with the characters ″corbaloc:″. This is followed by ″rir:″, which indicates the resolve_initial_references form or one or more IIOP addresses. Both RIR and IIOP forms are followed by a forward slash delimiter and the object key.

An IIOP address begins with either ″iiop:″ or just ″:″. Optionally, the GIOP version can be specified and delimited by a trailing ″@″. Next, a host identifier is specified, either as a DNS-style host name or as a dotted decimal address. Optionally, the host identifier is followed by a delimiting ″:″ and a port number. If more than one IIOP address is specified, a ″,″ delimiter separates each one.

After the last iiop: or rir: address, a forward slash, ″/″, delimits the object key, which is the final part of the URL. The object key can contain embedded slashes.

If the GIOP version is not specified, it defaults to 1.0. This determines the GIOP level of messages that are used when contacting the host and port. If the port number is not specified, it defaults to 2809.

In the corbaloc URL scheme example, `corbaloc::primary.com:110,:1.2@backup.com:120/Dev/NameService`, two IIOP addresses are specified. An attempt to convert this to an object causes the Object Request Broker (ORB) to contact port 110 at host ″primary.com″ and request an object identified by the object key, ″Dev/NameService″. If this fails, the ORB contacts port 120 at the host ″backup.com″, using GIOP Version 1.2 and the object key ″Dev/NameService″.

When a resolve_initial_reference form of a corbaloc URL scheme is converted to an object, the object key portion of the URL is passed to CORBA::resolve_initial_references and the result is returned.

Only the IIOP form of a corbaloc URL scheme can be used as a parameter with the CORBA::ORB_init method to configure Initial References.

Corbaname URL scheme: The corbaname URL scheme, one of three CORBA object URL schemes, extends the corbaloc form to specify an entry within a Naming Service.

The corbaname URL scheme is similiar to the corbaloc scheme, but adds the ability to specify an entry within the Naming Service. Converting a corbaname URL scheme to an object is a two step process. The first step obtains a reference to the naming service. The second step requests an entry within that naming service.

The additional specification of the entry within the naming service is called the key string. Syntactically, the key string follows the object key and is preceeded by a ″#″ delimiter. After a naming service object is obtained, the key string is passed to the naming service using the CosNaming::resolve_str method.

If the object key is not specified in a corbaname URL scheme, it defaults to ″NameService″.

Note: The portion of corbaname URL scheme, excluding the key string, must identify an object that supports CosNaming.

Example of corbaname URL strings:

```
corbaname::primary_name.com:402/myNamingService#productX/subcategoryY/specificZ
corbaname:iiop:Linux7.com:114#productX/subcategoryY/widgetW
corbaname:rir:#productX/subcategoryY/specificZ
```

Using CORBA::string_to_object to convert the first example URL causes the Object Request Broker (ORB) to contact the primary_name.com host on port 402 and obtain an object associated with the object key ″myNameService″. The ORB sends a CosNaming::resolve_str method to this object with the argument ″productX/subcategoryY/specificZ″. The result of the CosNaming::resolve_str method is returned from string_to_object().

In the second example, the object key is not specified. When converting this, the ORB contacts the Linux7.com host on port 114 and asks for an object associated with the default key, ″NameService″. The key string, ″productX/subcategoryY/widgetW″, is passed using a CosNaming::resolve_str() method to this naming context object.

The third example shows the use of the resolve_initial_references form of addressing. When converting this, the ORB performs the equivalent of calling CORBA::resolve_initial_references(″productX/subcategoryY/specificZ″) and returns the result.

File URL Scheme: The file URL scheme, one of three CORBA object URL schemes, identifies a file containing another object reference.

A file URL scheme begins with the characters, ″file://″. The rest of the URL is a path name that identifies a file. When converting this, the Object Request Broker opens the file and reads from it another object reference, which can be either an object URL or an Interoperable Object Reference (IOR).

Example of corbaname URL scheme strings:

```
file://c:\temp\a_url.txt
```

Converting the previous file URL scheme to an object causes the ORB to read the object reference (URL or IOR) from the file and, in turn, convert it to an object.

Strategies for retrieving the IOR of a remote object

If the client ″bootstrapping″ operation does not establish contact with a remote naming service, you can use the following alternative strategies to retrieve the Interoperable Object Reference (IOR) of a remote object:

- Have the Object Request Broker (ORB) use a name service URL for the name service initial reference.

  You can obtain the remote ORB's root name context and store it as a string-modified IOR in a file. During ORB initialization, CORBA C++ clients and servers can set the `com.ibm.CORBA.localObjrefFile` ORB property to the path name of the file that contains the string-modified IOR of a root naming context. The root naming context is then returned by calling ORB::resolve_initial_references(″NameService″).
- Pass the naming service object reference directly to a client.

  You can write an application to store the string-modified IOR of a remote ORB's root naming context into a file. You then can make the file available (for

example, by copying) to the client environment. The client then can read the string-modified IOR from the supplied file and use the ORB::string_to_object interface to resolve the root naming context. Use this approach only once during initialization, even if the client is to access many different server objects registered with the same naming service. In addition, the IOR for the name server is typically fairly static, so it is relatively simple to manage in a distributed environment.

- Use string_to_object and a file reference.

  Similar to the preceding option, put a string-modified IOR for a remote object into a file, have the client read that IOR, and use the ORB::string_to_object interface to resolve the object reference. However, it does not use a Name Service at all.

- Name space federation.

  The client can look up an entry in the name server of one ORB and then rebind the reference in the name server of a different ORB. For example, you can write a utility to look-up an EJB's home in the WebSphere name service, string-modify the object reference and write it to a file. You then can use another utility to read this file, remodify the object reference, and bind it into another ORB's naming service.

- Use a co-existent naming service.

  A client can make use of another co-existent ORB that supports ″bootstrapping″ with other ORBs from the same vendor.

dumpior command: The dumpior command is a utility which can be used to display a stringified Interoperable Object Reference (IOR) in a human-readable fashion.

Syntax

`dumpior` *`filename`*

where *filename* is the name of the file which contains the stringified IOR.

Examples

The following example demonstrates the correct syntax:

```
dumpior file1.ref
```

CORBA client exception handling

The preferred coding practice for handling errors in C++ and Java is by using exceptions, which is supported by using the standard try, catch, and throw logic of exception handling. Handling exceptions are a critical part of the client programming model. The exceptions that are thrown must be understood and handled appropriately by application developers.

In some cases, a server implementation object can encounter an error for which it might need to throw an exception to the client to give the client the opportunity to recover from the error.

This topic provides the following information about handling exceptions:

- CORBA support, using appropriate exceptions
- CORBA support, catching exceptions

CORBA support, using appropriate exceptions

CORBA exceptions are used to communicate between server implementation objects and client applications. You must follow specific rules regarding which CORBA exceptions to use. The following abstract CORBA exception classes are defined:

CORBA::Exception
This is the abstract class that is the base of all of the CORBA exceptions. Because this class is an abstract, it is never thrown. However, it can be used in catch blocks to process all of the CORBA exceptions in one block.

CORBA::UserException
This is the abstract class for all of the CORBA user exceptions and is a subclass of CORBA::Exception. This class must be used as the base class of all user-defined exception classes. The contents of these classes have no special format. Methods that throw these classes must declare their usage in IDL using the `raises` clause.

CORBA::SystemException
This is the abstract class for all of the CORBA standard exceptions and is a subclass of CORBA::Exception. These exceptions can be thrown by any method regardless of the interface specification. Standard exceptions cannot be listed in raises expressions, therefore whether an interface throws a system exception is unknown. This means, be prepared to handle standard exceptions on all of the method calls. Each standard exception includes a minor code to provide more detailed information.

Any method can throw a standard exception, even if there are no exceptions declared in the `raises` clause of that method. Thus a method can throw an exception at any time.

Note: CORBA standard exceptions are a predefined list of exceptions that can be thrown from any method. CORBA has defined the class that provides this support as CORBA::System Exception. For more information about CORBA exceptions, see *The Common Object Request Broker: Architecture and Specification*.

CORBA support, catching exceptions

Client programs are required to handle exceptions because the default behavior for uncaught exceptions is to end the process. (If the client process ends unexpectedly, suspect an uncaught exception.)

A client program can handle exceptions within the catch clause of a try or catch block that encompasses remote method invocations or calls to ORB services. Typically, exception instances are actually instances of either the SystemException or UserException classes.

When deciding how or what exceptions to catch in a client application, consider the following general rules for exception handling:

- Perform specific error recovery as necessary. You can perform specific error recovery by proper structuring of `catch` clauses.
- Check for the most specific exceptions first and most general exceptions last.
- Make use of the information that is available in the exception. All CORBA exceptions support the id() method that returns the exception identifier. System exceptions also provide minor() and completed() methods that return the minor code and completion status respectively.

Specific standard exceptions cannot be caught individually. If you need to handle individual standard exceptions, you can do so within a CORBA::SystemException catch block and use the id() method.

Consider the following simple client example:

```
try
{
   // Some real code goes here
   foo.boo();
}
// Catch and process specific User exceptions
...
// Catch any other User exceptions defined for the method in the
// `raises' clause
catch (CORBA::UserException &ue)
{
   // Process any other User exceptions. Use the id() method to
   // record or display useful information
   cout << "Caught a User Exception: " << ue.id() << endl;
}
// Catch any System exceptions
catch (CORBA::SystemException &se)
{
   // Process any System exceptions. Use the id() and minor()
   // methods to record or display useful information
   cout << "Caught a System Exception: " << ue.id() << ": " <<
        ue.minor() << endl;
}
catch (...)
{
   // Process any other exceptions. This would catch any other C++
   // exceptions and should probably never occur
   cout << "Caught an unknown Exception" << endl;
}
```

Coding tips for CORBA memory management

The rule for proper CORBA memory management is that the caller owns all of the storage.

Memory management in Java is somewhat automatic.

The general model for CORBA C++ memory management on the client is to use _var objects. This means that when an _ptr is returned, it must be placed into an _var by the client. The _var assumes responsibility for the storage pointed to by the _ptr that is placed into the _var. The _var is a class and its destructor runs when the _var goes out of scope.

The other option for CORBA C++ clients is to use the duplicate() and release() methods. The duplicate() method is available for making a copy of a client stub object, while the release() method is used to free the local memory used by a pointer.

For more reference information about CORBA C++ memory management, see the topic, “Storage management and _var types” on page 129.

Managing the storage of object references

Managing the storage of object references is one of the areas where proper memory management is required. You must use use _var variables or the duplicate() and release() methods as stated previously.

There are also special considerations when passing object references as parameters. The caller is always responsible for allocating storage for object references. The caller also is responsible for releasing of all inout and returned object parameters.

For inout parameters, the caller provides an initial value. If the receiver wants to reassign the inout parameter, it must call the release() operation on the initial input value. To continue to use an object reference passed as an inout, the caller first must duplicate the reference.

CORBA client to WebSphere EJB server

CORBA clients can use the CORBA client programming model to access enterprise beans hosted by a WebSphere EJB server, as shown in the following figure:

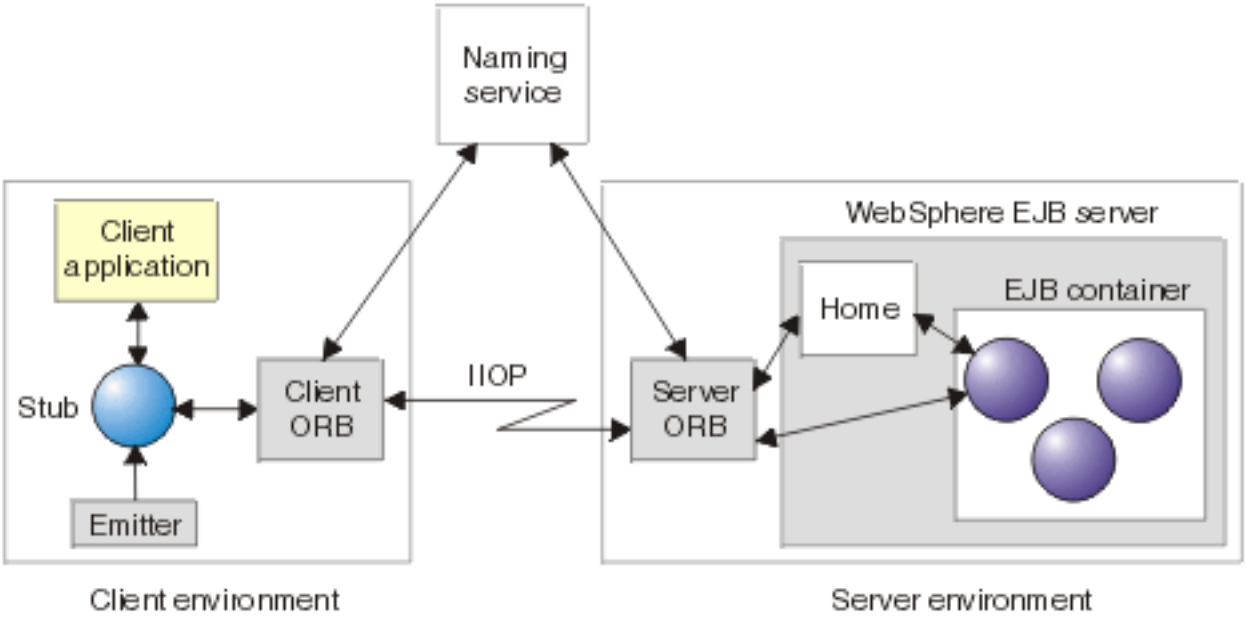


Figure 1. Scenario: CORBA client to WebSphere EJB server

The EJB server provides the server implementation objects (enterprise beans) that client applications need to access and implement the services that support those objects. The EJB class file is used to generate the Interface Definition Language (IDL) for the class and its home. Serializable objects used in the EJB interface are expressed in the IDL as CORBA valuetypes. Implementations for CORBA valuetypes must be provided on the client, so it is important to keep this simple.

In this scenario, when the client wants to call a method on a server object (an enterprise bean), the following sequence of events occur:

1. When the client environment is started, the client ORB is initialized and the ORB "bootstrap" process accesses the naming service (with CORBA CosNaming bindings).
2. When a client application needs to access an enterprise bean, the client environment uses the naming service to find a location for the bean.
3. The home locates or creates the enterprise bean and passes the interoperable object reference (IOR) of the bean back to the client.
4. The client's ORB creates a proxy (stub) object for the bean and returns it to the client application.

5. The client invokes methods on the proxy object to communicate with the remote bean as though it is in the local process.

Developing a CORBA C++ server

Use this task to develop a CORBA server to service requests for business functions used in the implementation of client objects. The instructions and code extracts provided in this task are based on the development of the WSLoggerServer sample and are accessible from the Samples Gallery. The Samples Gallery is installed with IBM WebSphere Application Server.

Steps for this task

1. Create and edit an IDL file, *servant*.idl, to specify the public interface to the *servant* object class; where *servant* is the name of the server implementation class.

   For more information about creating and editing an IDL file for the *servant* object class, see Defining the interface for a CORBA servant class.

   This step results in a fully specified *servant*.idl file.
2. Compile the *servant* IDL file, *servant*.idl, to produce the usage binding files needed to implement and use the *servant* object within a particular programming language.

   For more information about compiling an IDL file, see "Compiling a CORBA server implementation class IDL (using idlc)" on page 23.

   This step results in the set of usage binding files required for the *servant*.idl file.
3. Add declarations for class variables, constructors, and destructors to the *servant* implementation header (*servant*.ih).

   For more information about adding declarations to an implementation header, see "Adding declarations to a CORBA servant class definition (*servant*.ih)" on page 23.

   This step results in the *servant* implementation header file, *servant*.idl, that contains all the declarations for class variables, constructors, and destructors needed by the implementation.
4. Complete the *servant* implementation *servant*_I.cpp, to add the code that implements the *servant* business logic.

   For more information about completing the *servant* implementation, see "Completing the CORBA servant implementation (*servant*_I.cpp)" on page 26.

   This step results in the server implementation file, *servant*_I.cpp, that contains the code needed by the implementation to support the business logic.
5. Create the server main source file, *server*.cpp, to write the code for the methods that the server implements. (For example, to perform initialization tasks and create *servant* objects).

   For more information about completing the *servant* implementation, see "Creating the CORBA server main code (server.cpp)" on page 27.

   This step results in the server main source file, *servant*.cpp, that contains the main() function and associated code needed to implement the server.
6. Build the server code as described in Building a CORBA C++ server.
7. Store a logical definition for the server in the system implementation repository (using regimpl).

   For more information about storing a logical definition in the implementation repository, see "Storing a logical definition for a CORBA server in the system implementation repository" on page 37.

This step results in the implementation repository containing an entry for the server and the server object implementation class that it supports.

Defining the interface for a CORBA servant class

Use this task to define the public interface of a CORBA servant class. It provides the business logic to be used by clients. This defines the information that a client must know to call and use servant objects of that class and forms one stage of the tasks to develop a CORBA server or client.

Steps for this task

1. Create an IDL file, *servant*.idl, where *servant* is the name of the server implementation class.

   This step results in a fully specified *servant*.idl file.

2. Edit the *servant*.idl file to add an interface definition.

   The interface definition declares the interface name (and optionally its base interface names) and the methods (operations), and any constants, type definitions, and exception structures that the interface exports.

   The following information is an overview of the format of an interface declaration. It provides links to the reference topics that describe parts of the IDL declaration and IDL syntax. For reference information about IDL interface declarations and the component declarations that they can contain, see "IDL interface declarations" on page 151.

   An interface declaration has the following syntax:

```
interface  interface-name
 [: base-interface1, base-interface2, ...]
  {
   [constant declarations]
   [type declarations]
   [exception declarations]
   [attribute declarations]
   [operation declarations]
 };
```

   interface-name
   : The name of the public interface for the servant object. This should match the servant class name.

   [: *base-interface1*, *base-interface2*, **...]**
   : The base interface names for one or more parent interfaces from which this interface, *interface-name*, is derived.

     Specify base interface names only if this interface is derived from one or more parent interfaces. Each base interface is specified in the form, interface_name, and can be named only once in the interface statement header. If you specify a base interface name, you also must add an include statement for the base interface IDL file to the top of the *servant*.idl file.

   [constant declarations] and [type declarations]
   : An interface declaration can include constant declarations and type declarations, as in C and C++, with some restrictions and extensions. For more information about these declaration types, see "IDL type declarations" on page 152.

   [exception declarations]
   : An interface declaration can include exception declarations that define data structures to be returned when an exception occurs during the execution of an operation. Each exception declaration specifies a name

and, optionally, a struct-like data structure for holding error information. For more information about these declaration types, see "IDL exception declarations" on page 155.

[attribute declarations]
Declaring an attribute as part of an interface is equivalent to declaring one or two accessor operations: one to retrieve the value of the attribute (a get or read operation) and, unless the attribute specifies read only, one to set the value of the attribute (a set or write operation). For more information about these declaration types, see "IDL attribute declarations" on page 156.

[operation declarations]
Operation declarations define the interface of each operation introduced by the interface. An IDL operation typically is implemented by a method in the implementation programming language. Hence, the terms operation and method often are used interchangeably. For more information about these declaration types, see "IDL operation declarations" on page 157.

The order in which these declarations are specified is usually optional and declarations of different kinds can be intermixed. Although all of the declarations listed previously are optional, in some cases using one declaration can mandate another. For example, if an operation raises an exception, the exception structure must be defined beforehand. In general, types, constants, and exceptions, as well as interface declarations, must be defined before they are referenced, as in C or C++.

Results

This task results in a fully specified IDL file, *servant*.idl, that contains a declaration for the public interface to a servant class, *servant*.

For example, for a servant class called WSLogger, the IDL file, WSLogger.idl, was created and edited to add the following interface definition:

```
interface WSLogger
{
  void setFileName(in string newFileName);
  string getFileName();
  void setMethodName( in string newMethodName );
  string getMethodName();
  short openLogFile();
  short closeLogFile();
  short writeLogMessage(in string newMessage, in short newSeverity);
  enum mdyFormat { DMY_DATE_FORMAT,
                   MDY_DATE_FORMAT };
  void setDateFormat(in unsigned short newDateFormat);
  unsigned short getDateFormat();
};
```

What to do next

Compile the *servant*.idl to create the usage bindings and other files needed to complete the implementation as described in "Compiling a CORBA server implementation class IDL (using idlc)" on page 23.

Compiling a CORBA server implementation class IDL (using idlc)

Use this task to compile the IDL file, *servant*.idl, that defines the public interface for a CORBA server implementation class. You also can use this task to compile the IDL file (also referred to in this task as *servant*.idl) for an enterprise bean.

Note: If your servant.idl file references other IDL files, ensure that all those other IDL files can be accessed by the idlc program.

Steps for this task

1. At a command line, change to the directory that contains the IDL file, *servant*.idl, where *servant* is the name of the server implementation class.
2. Type the following command:

   ```
   idlc -ehh:ih:ic:uc:sc servant.idl
   ```

   The names of the generated output files are derived from the file name of the specified IDL file. For example, for the IDL file, *servant*.idl, the `-ehh` emitter option outputs the file *servant*.hh.

   This produces the files *servant*.hh, *servant*.ih, *servant*_I.cpp, *servant*_C.cpp, and *servant*_S.cpp.

Results

This task creates the usage binding files needed to implement and use the servant object within a particular programming language. For example, for a server object implemetation class called WSLogger, the IDL file, WSLogger.idl, is created and edited to add its interface definition. To compile the IDL file, the following command is used:

```
idlc -ehh:ih:ic:uc:sc WSLogger.idl
```

This creates the following files: WSLogger.hh, WSLogger.ih, WSLogger_I.cpp, WSLogger_C.cpp, and WSLogger_S.cpp.

What to do next

Add declarations for class variables, constructors, and destructors to the servant class definition, *servant*.ih, as described in "Adding declarations to a CORBA servant class definition (*servant*.ih)".

This task also can be used to create the client-side bindings files needed to develop a CORBA client to access an enterprise bean as described in "Developing a CORBA C++ client" on page 2.

Adding declarations to a CORBA servant class definition (*servant*.ih)

Use this task to add declarations for class variables, constructors, and destructors for a CORBA servant class to its skeleton implementation header file, *servant*.ih. This defines any private data members for the implementation code in the associated *servant*_I.cpp file.

This task follows the task to compile the *servant*.idl file, which defines the public interface for the server implementation class. For more information about

compiling the IDL file, which creates the *servant*.ih file, see "Compiling a CORBA server implementation class IDL (using idlc)" on page 23.

Steps for this task

1. At a command line change to the directory that contains the *servant*.ih file, where *servant* is the name of the servant class.
2. Edit the implementation header file, *servant*.ih, to add appropriate declarations for class variables, constructors, and destructors. For more information about the types of declarations that you can add to an implementation header file, see "IDL type declarations" on page 152.

   For example, the **idlc** command, `idlc -ehh:ih:ic:uc:sc -mdllname=WSLogger WSLogger.idl`, converts the following interface declaration to the class declaration in the WSLogger.ih file. The WSLogger.ih file was edited to add the extra declarations that are shown in bold in the following example:

   Example: WSLogger interface and declarations added to the skeleton implementation header

Interface declaration in WSLogger.idl	Implementation header in WSLogger.ih

interface WSLogger {
  void setFileName(in string newFileName);
  string getFileName();
  void setMethodName(in string newMethodName);
  string getMethodName();
  short openLogFile();
  short closeLogFile();
  short writeLogMessage(in string newMessage, in short newSeverity);
  const short DMY_DATE_FORMAT = 1;
  const short MDY_DATE_FORMAT = 2;
  void setDateFormat(in unsigned short newDateFormat);
  unsigned short getDateFormat();
};
	class WSLogger_Impl : public virtual ::WSLogger_Skeleton {
	  public:
	  ::CORBA::Void setFileName (const char* newFileName);
	  char* getFileName ();
	  ::CORBA::Void setMethodName (const char* newMethodName);
	  char* getMethodName ();
	  ::CORBA::Short openLogFile ();
	  ::CORBA::Short closeLogFile ();
	  ::CORBA::Short writeLogMessage (const char* newMessage, ::CORBA::Short newSeverity)
	  ::CORBA::Void setDateFormat (::CORBA::UShort newDateFormat);
	  ::CORBA::UShort getDateFormat ();
	private:
	char * fileName;
	char * methodName;
	::CORBA::UShort dateFormat;
	ofstream logFile;
	::CORBA::UShort logFileOpen;
	public:
	WSLogger_Impl(char * newFileName);
	virtual ~WSLogger_Impl();
	};

What to do next

Add code to the skeleton implementation definition, *servant*_I.cpp, to implement the business logic as described in "Completing the CORBA servant implementation (*servant*_I.cpp)".

Completing the CORBA servant implementation (*servant*_I.cpp)

Use this task to add code for a CORBA server implementation class to its skeleton implementation file, *servant*_I.cpp. The code defines the methods that implement the business logic for the server implementation class, *servant*.

This task follows the task to add declarations for class variables, constructors, and destructors to the servant implementation header file, *servant*.ih. For more information about adding declarations to an implementation header, see "Adding declarations to a CORBA servant class definition (*servant*.ih)" on page 23.

Steps for this task

1. At a command line, change to the directory that contains the *servant*_I.cpp file, where *servant* is the name of the server implementation class.
2. Edit the implementation file, *servant*_I.cpp, to add appropriate code to implement the business logic methods.

   For example, the **idlc** command, `idlc -ehh:ih:ic:uc:sc -mdllname=WSLogger WSLogger.idl`, converts the following interface declaration to the skeleton methods in the implementation file, WSLogger_I.cpp. The WSLogger_I.cpp file was edited to add the code to implement the methods. The code added for the `WSLogger_Impl::writeLogMessage` method is shown in bold in the following example:

```
::CORBA::Void WSLogger_Impl::setFileName (const char* newFileName)
{
}

char*  WSLogger_Impl::getFileName ()
{
}

::CORBA::Void WSLogger_Impl::setMethodName (const char* newMethodName)
{
}

char*  WSLogger_Impl::getMethodName ()
{
}

::CORBA::Short WSLogger_Impl::openLogFile ()
{
}

::CORBA::Short WSLogger_Impl::closeLogFile ()
{
}

// This method writes one line of message text to the log file. The line
// prefaced with the current date and time in the currently specified
// format, the current method name (if any), the severity level, and
// the message text.

::CORBA::Short WSLogger_Impl::writeLogMessage (const char* newMessage, ::CORBA::Short newSeverity)
{
  ::CORBA::String_var timeString;

  if ( logFileOpen == FALSE )
    return( -1 );
  // Get the date and time string.
  time_t tp;
  time_t tp2;
  if ( ( tp = time(&tp2) ) != -1 )
  {
    struct tm *x = gmtime( &tp2 );
    timeString = ::CORBA::string_dup( ctime( &tp2 ) );
  }
```

```
  // Determine the day and month.
  ::CORBA::String_var day = ::CORBA::string_alloc( 3 );
  ::CORBA::String_var month = ::CORBA::string_alloc( 4 );
  day[0] = timeString[8];
  day[1] = timeString[9];
  day[2] = 0;
  month[0] = timeString[4];
  month[1] = timeString[5];
  month[2] = timeString[6];
  month[3] = 0;

  // Copy the time and year.
  ::CORBA::String_var time = ::CORBA::string_alloc( 14 );
  strncpy( time, (const char *) &timeString[11], 13 );
  time[13] = 0;

  // Output the time of the log message.
  if ( dateFormat == DMY_DATE_FORMAT )
    logFile << day << " " << month;
  else if ( dateFormat == MDY_DATE_FORMAT )
    logFile << month << " " << day;
  logFile << " " << time << ", ";

  if ( getMethodName() != NULL )
    logFile << getMethodName() << ", ";

  logFile << "severity " << newSeverity << ": ";

  // Output the log message.
  logFile << newMessage << endl;

  return 0;
}

::CORBA::Void WSLogger_Impl::setDateFormat (::CORBA::UShort newDateFormat)
{
}

::CORBA::UShort WSLogger_Impl::getDateFormat ()
{
}
```

What to do next

Create the server main code (server.cpp), to implement the server as described in "Creating the CORBA server main code (server.cpp)".

Creating the CORBA server main code (server.cpp)

Use this task to create a CORBA server that hosts a servant object. The server performs the following tasks:

1. Validates user input
2. Initializes the server environment
3. Accesses naming contexts
4. Names, creates, and binds a servant object
5. Creates a server shutdown object
6. Goes into a wait loop
7. Services requests

This task is the next step after adding code for the business logic methods in the servant implementation file, *servant*_I.cpp. For more information about adding code to a servant implementation file, see "Completing the CORBA servant implementation (*servant*_I.cpp)" on page 26.

Steps for this task

1. Create a source file, *servant*Server.cpp, where *servant* is the name of the implementation class for which the server is to service requests.

2. Edit the server source file, *servant*Server.cpp, to add appropriate code to implement the server methods. To do this, complete the following steps:
   a. Add the necessary include statements, as described in "Adding include statements".
   b. Add the main() function, in the form:

```
int main( int argc, char *argv[] )
 {
   int rc = 0;
 }
```

3. Edit the server source file, *servant*Server.cpp, to add appropriate code to initialize the server environment as described in "Initializing the server environment" on page 29.
4. Edit the server source file, *servant*Server.cpp, to add appropriate code to access naming contexts as described in "Accessing naming contexts" on page 30.

   At this point, initialization has been accomplished and a naming context has been created for servant objects.
5. Edit the server source file, *servant*Server.cpp, to add appropriate code to create and bind servant objects as described in "Creating and binding servant objects" on page 33.
6. Edit the server source file, *servant*Server.cpp, to add code to create a server shutdown object as described in "Creating a server shutdown object" on page 34.
7. Edit the server source file, *servant*Server.cpp, to add code to put the server into an infinite loop (to service any ORB requests received) as described in "Putting the server into a loop to service requests" on page 35.
8. Edit the server source file, *servant*Server.cpp, to add code to shutdown the server and release resources used as described in "Shutting down the server and releasing resources used" on page 35.

Adding include statements

Use this task to add the necessary include statements to the source file for a CORBA server main code.

For example:

```
#include "servant.ih"
#include "servershutdown.h"

#include <CosNaming.hh>
```

where:

*servant***.ih**
: Specifies the name of the implementation header file for the servant class, *servant*, to be hosted by the server.

*servershutdown***.h**
: Specifies the name of the header file for the class used to shut down the CORBA server.

CosNaming.hh
: Specifies the header file for the COSNaming functions.

What to do next

You can add the code for the functions needed in the server main code as described in Creating a CORBA server main code (server.cpp).

Initializing the server environment

One of the first tasks for a CORBA server application after startup is to initialize the server environment to perform the following actions:

1. Get a pointer to the implementation repository.

   The implementation repository is a persistent data store of ImplementationDef objects, each representing a logical CORBA server that has been registered in the repository. A server application typically receives a pointer to the implementation repository by using the CORBA::ImplRepository method. For example:

   ```
   ::CORBA::ImplRepository_ptr implrep = new ::CORBA::ImplRepository();
   ```

2. Get a pointer to the ImplementationDef associated with the server alias.

   The ImplementationDef, which is obtained from the Implementation Repository, describes the server. For example, it specifies a UUID that uniquely identifies the server throughout a network. Each server must retrieve its own ImplementationDef object from the Implementation Repository (using the ImplRepository class) because the ImplementationDef is a parameter required by the BOA::impl_is_ready method. A server application typically receives a pointer to its ImplementationDef by using the CORBA::ImplRepository find_impldef or find_impldef_by_alias method. For example:

   ```
   imp = implrep->find_impldef_by_alias(argv[1] );
   ```

   where argv[1] is the server alias specified as a string in the command used to start the server.

3. Initialize the Object Request Broker (ORB) and BOA.

   This action is used to initialize the ORB and BOA and to return a pointer to each.

   A server application initializes the ORB by calling the CORBA::ORB_init() method, which also returns a pointer to the ORB. (If necessary, this method creates a new instance of the ORB.) For example, the following code extract initializes the ORB and returns a pointer to it:

   ```
   op = ::CORBA::ORB_init(argc, argv, "DSOM");
   ```

   where argc and argv refer to the properties specified in the command used to start the server.

   A server application initializes the BOA by calling the CORBA::BOA_init() method on the ORB. For example, the following code extract initializes the BOA and returns a pointer to it:

   ```
   bp = op->BOA_init(argc, argv, "DSOM_BOA");
   ```

   where argc and argv refer to the properties specified in the command used to start the server. Using the BOA_init() method, you must specify `DSOM_BOA` after the parameter *argv*.

4. Register the server application as a CORBA server.

   This action calls the CORBA::BOA::impl_is_ready method to initialize the server application as a CORBA server. This method initializes the server's communications resources so that it can accept incoming request messages and export objects. For example, the following code extract registers the server (with the alias specified on the command used to start the server):

   ```
   bp->impl_is_ready(imp, 0 );
   ```

   Note: The zero (0) value indicates that the server must not register itself with the location service daemon because CORBA servers within WebSphere support

transient objects only. This parameter is an IBM extension to the CORBA specification and must be specified only for lightweight servers of transient objects.

The following example illustrates the initialization code that must be added to the server's main() function:

```
int main( int argc, char *argv[] )
{
   // Replace "servantServer" with the appropriate server alias.
   const char *serverAlias = "servantServer";

   // Create the implementation repository.
   ::CORBA::ImplRepository *implRep    = new ::CORBA::ImplRepository();

   // Find this server's implementation definition.
   ::CORBA::ImplementationDef *implDef = implRep->find_impldef_by_alias ( serverAlias );

   if ( implDef == NULL )
   {
      cerr << "Error: could not find the implementation definition: "
           << serverAlias << endl;
      return 1;
   }

   ::CORBA::ORB_ptr orbPtr = ::CORBA::ORB_init ( argc, argv, "DSOM" );
   if ( ::CORBA::is_nil(orbPtr) )
   {
      cerr << "Error: could not initialize the ORB." << endl;
      return 2;
   }

   ::CORBA::BOA_ptr boaPtr = orbPtr->BOA_init ( argc, argv, "DSOM_BOA" );
   if ( ::CORBA::is_nil(boaPtr) )
   {
      cerr << "Error: could not initialize the BOA." << endl;
      return 3;
   }

   try
   {
      bp->impl_is_ready ( implDef, 0 );
   }
   catch ( ::CORBA::SystemException &e )
   {
      cerr << "Error: received system exception: " << e.id()
           << ", minor code " << (void *) e.minor() << endl;
   }
   catch ( ... )
   {
      cerr << "Error: received unknown exception." << endl;
   }

   .
   .
   .

   return 0;
}
```

What to do next

Add code to the server source file to enable the server to access naming contexts as described in "Accessing naming contexts".

Accessing naming contexts

Use this task to add code to the source file for a CORBA server to access naming contexts. This code is used to create a new naming context within which the CORBA server can bind servant objects. The code performs the following actions after the server environment has been initialized:

1. Gets a pointer to the root naming context

2. Creates a ::CosNaming::Name for the domain and getting a pointer to the domain naming context
3. Gets a new servant naming context for servant objects

To add the get_naming_context() function to the source file for a CORBA server, edit the server source file, servantServer.cpp, and add the following code:

```
//
// This function accesses the Name Service and then gets or creates
// the desired naming contexts. It returns the naming context for
// the servant context.
//
::CosNaming::NamingContext_ptr get_naming_context ( ::CORBA::ORB_ptr orbPtr )
{
   ::CosNaming::NamingContext_var rootNameContext = NULL;
   ::CosNaming::NamingContext_var domainNameContext = NULL;
   ::CosNaming::NamingContext_ptr servantNameContext = NULL;
   ::CORBA::Object_var objPtr;

   // Get access to the Naming Service.
   try
   {
     objPtr = orbPtr->resolve_initial_references ( "NameService" );
   }
   // catch exceptions

   // Narrow the returned object to a NamingContext object.
   rootNameContext = ::CosNaming::NamingContext::_narrow ( objPtr );
   if ( ::CORBA::is_nil(rootNameContext) )
   {
      cerr << "Error: could not narrow the root naming context." << endl;
      return NULL;
   }

   // Create the "domain" Name.
   ::CosNaming::Name name1;
   name1.length(1);
   name1[0].kind = ::CORBA::string_dup("");
   name1[0].id   = ::CORBA::string_dup("domain");

   // Find the "domain" naming context.
   try
   {
      objPtr = rootNameContext->resolve ( name1 );
   }
   // catch exceptions

   // Next, narrow the domain naming context object.
   domainNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
   if ( ::CORBA::is_nil( domainNameContext ) )
   {
      cerr << "Error: could not narrow the domain naming context." << endl;
      return NULL;
   }

   // Create the "servantContext" Name.
   ::CosNaming::Name name2;
   name2.length(1);
   name2[0].kind = ::CORBA::string_dup("");
   name2[0].id   = ::CORBA::string_dup("servantContext");

   // Create the "servantContext" naming context.
   try
   {
      objPtr = domainNameContext->bind_new_context ( name2 );
   }
```

```
   // If the servant naming context already exists,
   // then just resolve it...
   catch ( ::CosNaming::NamingContext::AlreadyBound &e )
   {
      cout << "Warning: servant's naming context already exists." << endl;
      cout << "Trying to resolve the context." << endl;
      try
      {
    objPtr = domainNameContext->resolve( name2 );
      }
      // catch exceptions ...
   }

   // Next, narrow the new naming context object.
   servantNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
   if ( ::CORBA::is_nil( servantNameContext ) )
   {
      cerr << "Error: could not narrow the servant's naming context." << endl;
      return NULL;
   }

   return servantNameContext;
}
```

Next, modify the server's main() function so that it calls the get_naming_context() function:

```
int main( int argc, char *argv[] )
{
   .
   .  initialization code
   .

   // Get the naming contexts to which the servant object will be bound.
   ::CosNaming::NamingContext_var servantNameContext =
      get_naming_context ( orbPtr );

   if ( servantNameContext == NULL )
   {
      cerr << "Error: failed to obtain the name context." << endl;
      return 4;
   }

   .
   .
   .

}
```

Results

This task creates and returns a pointer object, servantNameContext, to the naming context for servant objects.

What to do next

Add code to the server source file to name, create, and bind servant objects as described in "Creating and binding servant objects" on page 33.

CORBA server naming contexts: Before a CORBA server can create and make available a servant object, it must have a logical name space for the servant object to exist in. This logical name space is a naming context for servant objects. The server can create a new naming context within any location within the root naming

context. For example, a server called *servant*Server might create a new naming context called *servant*Context into which the server binds the servant object. Optionally, this context might be located within a domain context, which in turn is located within the root naming context. (You can create a servant context with only the root naming context as its parent or with one or more intermediary parent contexts.)

Creating and binding servant objects

Use this task to add code to the source file for a CORBA server to create servant objects, and bind them into the appropriate naming context. This makes it possible for clients to find and use servant objects.

To add code to create and bind servant objects, edit the server source file, servantServer.cpp, and add the following code:

```
int bind_object (
   ::CORBA::Object_ptr objPtr,
   ::CosNaming::Name &name,
   ::CosNaming::NamingContext_ptr servantNameContext
   )
{
   try
   {
      servantNameContext->bind ( name, objPtr );

      cout << "The servant object was bound successfully into the name space." << endl;
   }
   // catch exceptions

   return 0;
}
```

Next, modify the main() function so that it creates the servant object and calls the bind_object() function:

```
int main ( int argc, char *argv[] )
{
   .
   . initialization code
   .

   // Get the servant naming context.
   ::CosNaming::NamingContext_var servantNameContext = get_naming_context ( orbPtr );
   if ( servantNameContext == NULL )
   {
      return 4;
   }

   // Create the servant object.
   // This is the object that will be used by the client.
   servantImpl *servantObject = new servantImpl();

   // Create the CosNaming::Name for our object.
   ::CosNaming::Name name;
   name[0].kind = ::CORBA::string_dup("");
   name[0].id   = ::CORBA::string_dup("servantObject1");

   // Next, bind the servant object into the servant naming context.
   rc = bind_object ( servantObject, name, servantNameContext );
   if ( rc != 0 )
   {
      cerr << "Error: could not bind servant object into name space." << endl;
      return 5;
   }

   .
   .
   .
}
```

Results

This task adds code that enables a CORBA server to create and bind servant objects.

What to do next

Add code to the server source file to enable the server to create a server shut down object that can be used to help shut down the server as described in "Creating a server shutdown object".

Creating a server shutdown object

When a CORBA server is started, it initializes itself, calls the execute_request_loop()method, and specifies a blocking mode (::CORBA::BOA::SOMD_WAIT). This puts the server into an infinite wait loop during which the ORB transmits requests to and from the servant object hosted by the server. Because the execute_request_loop() method never returns, the server can never terminate unless it is forced. A server shutdown object makes it possible to terminate the server gracefully. The server creates a server shutdown object and gives it a string that is used to shut down the server.

To stop the server, run the WSStopServer program, which tells the ORB to shut the server down. WSStopServer has the following command syntax:

```
WSStopServer server_alias
```

where *server_alias* is the server alias (defined in the Implementation Respository).

To create a WSServerShutdown object, modify the server's main() function by adding the following code:

```
int main ( int argc, char *argv[] )
{
   .
   . initialization code
   .

   .
   . create and bind the servant object
   .


   // Create a WSServerShutdown object that can break the server out of the
   // execute_request_loop() method when we are ready to terminate
   // the server. The WSStopServer command will cause the subsequent
   // invocation of execute_request_loop() return to the server.
   WSServerShutdown *shutdownObj = new WSServerShutdown ( serverAlias, boaPtr );
   cout << "Created WSServerShutdown object" << endl;

   .
   .
   .
}
```

When created, the WSServerShutdown object is initialized with the server alias and the object adapter pointer.

Results

This task adds code to create a WSServerShutdown object. After the server has initialized itself, it creates the WSServerShutdown object, which waits for a

message informing it that the server is to be shut down. That message can be sent by the StopServer command line program provided with IBM WebSphere Application Server enterprise services.

What to do next

To continue developing the server main code, add code to put the server into an infinite wait loop. During this loop, the Object Request Broker (ORB) can transmit requests to and from the servant object hosted by the server as described in "Putting the server into a loop to service requests".

Putting the server into a loop to service requests

Use this task to add code to the source file for a CORBA server and cause the server to enter its request loop. This allows the server to respond to requests received from clients.

To cause the server to enter its request loop, edit the server source file, *servant*Server.cpp, and add the following code to the main() function:

```
int main( int argc, char *argv[] )
{
   .
   . initialization code
   .

   .
   . create and bind the servant object
   .


   .
   . Create the shutdown object
   .

   // Enter the request loop.
   cout << "Server is ready for e-business..." << endl;
   ::CORBA::Status stat = boaPtr->execute_request_loop ( ::CORBA::BOA::SOMD_WAIT );

   // Terminate the server.
   cout << "Server is now shutting down..." << endl;
   .
   .
   .

   return 0;
}
```

Results

This task adds code that causes the CORBA server to enter its request-processing loop. During this loop, the server can service incoming requests for the servant object or objects that it hosts.

What to do next

Add code to the server source file to enable the server to complete the server shut down when requested as described in "Shutting down the server and releasing resources used".

Shutting down the server and releasing resources used

Use this task to create code for a CORBA server, shut down the server, and release the resources that it used.

To cause the server to shut down, add the following code to the main() function:

```
int main ( int argc, char *argv[] )
{
   .
   .
   .

   // Terminate the server
   cerr << "Server is shutting down." << endl;

   boaPtr->deactivate_impl ( implDef );
   ::CORBA::release ( boaPtr );
   ::CORBA::release ( orbPtr );
   ::CORBA::release ( implDef );
   ::CORBA::release ( implRep );
}
```

Results

This task adds code that shuts down a CORBA server and releases the resources that it used when the server's execute_request_loop() is forced to return. The loop returns when a shut down request is made by a separate server shut down program.

Building a CORBA C++ server

This topic provides an overview of the task to build the code for a C++ CORBA server. The actual steps that you complete depend on the development environment that you use.

For example, if you are using the Microsoft C++ 6.0 compiler on Microsoft Windows NT to build a C++ CORBA server, you can use the following commands:

1. Compile the server.cpp, *servant*_I.cpp, and *servant*_S.cpp files.

   At a command line, run the following command for each file:

   ```
   cl /c /GX /DSOMCBNOLOCALINCLUDES /Iwasee_install\include
   ```

   where:

   wasee_install
   : is the directory into which IBM WebSphere Application Server is installed.

   filename.cpp
   : is the name of the file to be compiled (*server*.cpp, *servant*_I.cpp, or *servant*_S.cpp).

2. At a command line, run the following command for each file:

   ```
   link server.obj servant_I.obj servant_S.obj /OUT:server.exe
        wasororm.lib wasosa1m.lib wassrvsm.lib
   ```

What to do next

For more examples of building CORBA C++ server code (on several platforms) for IBM WebSphere Application Server, see the topic "Tutorial: Creating a user-defined C++ server and client" in the Samples Gallery, which is installed with IBM WebSphere Application Server.

Storing a logical definition for a CORBA server in the system implementation repository

Use this task to register a CORBA server in the implementation repository. To register a CORBA server in the implementation repository, identify the server's alias, the server application program, and the server object implementation class (servant) that the server implements. The information registered is used to activate the server process when the server is started and thereafter to help clients to find the server that supports servants that they want to use.

Steps for this task

1. To register the server, *server_alias*, type the following command:

   ```
   regimpl -A -i server_alias -p server_application
   ```

   where:

   server_alias
   is the server alias by which the server is known

   server_application
   is the name of the application program that implements the server.

   [Microsoft Windows® platform] The program name has the form `program.exe`

   [Unix platform] The program name has the form `program`

Results

This task results in the implementation repository containing an entry for the server and the server object implementation class that it supports.

For example, for a server object implemetation class called WSLogger, supported by the server application WSLoggerServer, you use the following **regimpl** command:

```
regimpl -A -i WSLoggerServer
```

Managing CORBA applications

Steps for this task

1. "Supporting SSL by WebSphere for CORBA C++ clients"
2. "Specifying run-time properties for CORBA C++ clients and servers" on page 52
3. "Resolving CORBA run-time errors" on page 66
4. "Managing the CORBA Interface Repository" on page 72

Supporting SSL by WebSphere for CORBA C++ clients

The following topic describes how to enable SSL to be used between CORBA C++ clients and EJB servers in a WebSphere network:

- "Enabling SSL certificate security between a CORBA C++ client and an EJB server" on page 38

In addition, the following conceptual topics offer information related to the SSL security mechanism:

- "SSL security for CORBA C++ clients" on page 48
- "CORBA C++ client: SSL and certificates" on page 49

- "CORBA C++ client: Structure of a certificate" on page 49
- "CORBA C++ client: Certificate authorities" on page 50
- "CORBA C++ client: Certificate chains" on page 51

Enabling SSL certificate security between a CORBA C++ client and an EJB server

Before you begin

Create a certificate to represent the target EJB server as described in Certificates.

To enable SSL security between a CORBA C++ client and an EJB server, complete the following steps:

Note: Each step is a separate procedure. After you complete each step, return to this overview procedure.

Steps for this task

1. Create a key database file for the client as described in "Creating a key database for a CORBA C++ client". This file is used to hold the client's certificate and the server's public key for use by the client.
2. Create a client certificate to uniquely identify the client as described in "Creating SSL certificates for a CORBA C++ client" on page 39. This also creates the client key database file that is used to hold the server's public key for use by the client.
3. If you have a client certificate from the Certificate Authority (CA), integrate it into your client key database file as described in "Integrating a CA-signed certificate into a CORBA C++ client key database file" on page 45. If you create your own self-signed client certificate, the certificate is created in the specified client key database file.
4. Extract the client certificate (which includes its public key) and add it as a signer certificate into the truststore file for each target server. This is described in the procedures "Extracting a certificate from a CORBA C++ client key database file" on page 46 and "Adding a signer certificate into a CORBA C++ client key database" on page 47.
5. Configure the server to enable SSL security and configure other security properties that you want for the server.
6. Configure the CORBA C++ client to enable security and configure other security properties that you want for the C++ clients as described in "Run-time properties for CORBA clients and servers" on page 54.

Results

When you start a CORBA C++ client application, the application determines its client properties file from the WASPROPS environment variable on the client host. From the file, the application determines the location of the client's key database file. The client then can use the certificates in its key database file to create secure connections with application servers.

Creating a key database for a CORBA C++ client: Use this procedure to create a Certificate Management System (CMS) key database file for a CORBA C++ client. This key database file contains the owner's certificate and its corresponding private key. The owner uses this key database file to identify itself to anyone that wants to

authenticate the owner. This procedure uses the GSKit 5 version of the IBM Key Management tool. To create a CMS key database file for a CORBA C++ client, complete the following steps:

Steps for this task

1. Start the GSKit 5 IBM Key Management tool, as described in "Starting the IBM Key Management tool". Once you have started the IBM Key Management tool, return to this page for the next step in this task.
2. Create a new CMS key database file. To do this, either click **Create a new key database file** on the tool bar or select **Key Database File > New** from the menu bar. You are prompted to enter the file name for the key database.
3. Specify a file name and location for the key database. The file name must be unique for the key database. This is typically in the form name.kdb, where name is the name of the client for which you are creating the key database.
4. Click **OK**. The **Password Prompt** window is displayed. You are prompted to enter a password for the key database.
5. Specify WebAS, or another password, to access the key database. This password is not used to protect the file, therefore, the password itself does not have to be protected. It is only required to release the information stored by the IBM Key Management tool during run time.
6. Click **OK**.

Results

You have successfully created a key database, and the IBM Key Management tool displays all of the default signer certificates. You can add, view, or delete signer certificates from this screen.

What to do next

Continue with the next step in the overview procedure article, "Enabling SSL certificate security between a CORBA C++ client and an EJB server" on page 38.

Starting the IBM Key Management tool: Steps for this task

To create and manage certificates for CORBA C++ clients, use the GSKit 5 IBM Key Management tool provided with the IBM HTTP package. Execute either of the following to start the IBM Key Management tool:

- For Windows platforms, click **Start > Programs > IBM HTTP Server 1.3.26 > Start Key Management Utility**.
- For Unix platforms, start the "ikeyman.sh" script in the IBMHttpServer/bin directory.

Creating SSL certificates for a CORBA C++ client: This procedure creates a unique certificate for a CORBA C++ client that will use SSL security. Use this procedure if the client SSL security is enabled to create secure connections with an EJB server based on SSL that uses client certificates. In this case, WebSphere assumes that you have created and installed a unique certificate for the client (and another for the server).

Note: Depending on how you organize the administration of your certificates (particularly if you involve a commercial certificate authority), the time it takes to create and install a certificate can be significant; perhaps several days. Therefore, plan to complete this procedure a few days before you need to use the certificate.

You can create and install either a test certificate for use during development and testing or a production certificate for use in a production WebSphere network. If you want to create your own self-signed test certificate, complete the following procedure:

- "Creating your own self-signed test certificate on a CORBA C++ client"

If you want to create a production certificate, complete the following procedures:

- "Planning for creating a CA-signed production certificate on a CORBA C++ client" on page 41
- "Creating and sending a certificate signing request on a CORBA C++ client" on page 43

What to do next

Continue with the next step in the overview procedure article, "Enabling SSL certificate security between a CORBA C++ client and an EJB server" on page 38.

Creating your own self-signed test certificate on a CORBA C++ client: Before you begin

If you want to create a self-signed certificate for a key database file, you must have created the key database file. Later, you can extract the certificate and add it to a target server's truststore file. For more information about creating key database files, see "Creating a key database for a CORBA C++ client" on page 38.

When you are developing a production application, you might not want to purchase a true digital certificate until after you are done testing the product. With the IBM Key Management tool, you can create a self-signed digital certificate to use until tesitng is complete. A self-signed digital certificate is a temporary digital certificate yo issue to yourself, with yourself as the CA.

Note: Do not release a production application with a self-signed test certificate; no browser or server will be able to recognize or communicate with your client.

To create a self-signed test certificate in a key database file, follow these steps:

1. Start the IBM Key Management tool as described in "Starting the IBM Key Management tool" on page 39. The **IBM Key Management** window is displayed.
   a. Open the key database file (filename.kdb) for the client for which you want to request a self-signed certificate. To open the key database file, either click **Open a key database file** on the tool bar or select **Key Database File > Open** from the menu bar. Type the name and location of the key database file at the prompt.
   b. Click **OK**. The **Password Prompt** window is displayed.
   c. At the prompt, type the password that you specified when you created the CMS key database file.
   d. Click **OK**. The IBM Key Management tool displays all of the default signer certificates. You can add, view or delete signer certificates from this screen.
2. To continue creating a self-signed test certificate, either click **Create a new self-signed certificate** on the tool bar or select **Create > New Self-Signed Certificate** from the menu bar. The **Create New Self-Signed Certificate** window is displayed.

3. Fill in the following certificate attributes, including the name of your client as the distinguished name. You can leave other attributes with their default values.

   Key Label
   The key label is used to uniquely identify the certificate within the key database file. For the CORBA C++ client, there typically is only one certificate in each key database file, so you can assign any label value. However, it is good practice to use a unique label, perhaps related to the client name.

   Version
   The version of the RSA cipher algorithm is used to digitally sign and authenticate certificates. Select the default version X509 V3.

   Key size
   Key size is the size of the key used to digitally sign and authenticate certificates. The default is 1024. For 128-bit cipher algorithms, the value can be either 512 or 1024. For 56-bit cypher algorithms, the value must be 512.

   Common Name
   The common name is the primary, universal identity for the certificate. It must uniquely identify the principal that it represents.

   Organization
   This is the name of your organization.

   Organization Unit
   (Optional) This is the name of your organization unit.

   Locality
   (Optional) This is the name of the location (city).

   State/Province
   (Optional) This is the name of the state/province.

   Zipcode
   (Optional) This is the zip code.

   Country
   This menu is the two-letter identifier of the country in which the server belongs.

   Validity period
   The default validity period of 365 days is typically used. Otherwise, specify the number of days that the certificate is valid.

4. Click **OK**. The **IBM Key Management** window is displayed. The **Personal Certificates** field shows the name of the self-signed digital certificate you created.

Note: If you have only one personal certificate, it is set as the default certificate for the database. If you have more than one personal certificate, choose which one is the default certificate. You can change the default certificate by first highlighting the certificate and then selecting **View/Edit**. Then, select the checkbox at the bottom of the screen to set this certificate as the default.

Planning for creating a CA-signed production certificate on a CORBA C++ client: Before you begin

Use this procedure to plan for the signed SSL certificates that you need to get from a certificate authority (CA) to properly enable SSL security between a server and C++ clients that use SSL mutual certificate authentication. You can use this procedure to get a CA-signed certificate for a client.

You need to create a certificate for a C++ client only if the client is enabled to create secure connections with a server based on SSL using client certificates. In this case, WebSphere assumes that you have created and installed a unique certificate for the client (and another for the server).

In a production WebSphere network, the production certificates are authenticated to verify the principal using the certificate. The principal is authenticated by a CA when the CA signs the principal's certificate. Because of the diligence that is expected of the CA, as described in "Certificate authorities", the authentication process for principals can take a significant amount of time. Commercial CAs often require up to a week to complete their authentication process. Even on-site CAs can take up to several days to complete their authentication process.

As a result, when you plan to add a new application server, you must plan for the certificates that you will need in advance of actually creating the server or client.

On the certificate signing request that you send to the CA, you need to specify the common name for the certificate. This is the primary, universal identity for the certificate that uniquely identifies the principal that it represents. For a server, a common convention is to use the server name. For a client, a common convention is to use a unique name to represent the C++ secured client.

For some CAs, including the fully qualified name of your host in the common name is required. For example, some CAs will not sign your certificate unless the domain portion of the host name is owned by your organization. When you plan the common name for a certificate request, check the format that your CA requires.

On the certificate signing request that you send to the CA, specify the name and address of your organization. Some certificate authorities require that you completely spell out the state or province fields. For example, you need to specify California as opposed to CA. Thus, check the format requirements for your CA.

If you do not get a production certificate (from a CA) before you want to start using the SSL security based on the CA-signed certificate, you can start with either of the following, less secure, alternatives:

- Create and use your own self-signed test certificate to perform some early tests. This provides an alternative to the test certificate provided with WebSphere. However, like the test certificate, this does not provide appropriate security for production use. Thus, replace it with a CA-signed certificate that legitimately represents your client for production use.
- Use the test certificate provided with WebSphere to perform some early tests. However, given the lack of security implied by that test certificate, replace it as soon as possible with a CA-signed or self-signed certificate that legitimately represents your client.

When you have received a signed certificate from a certificate authority, you can reconfigure the server and client so that they can use the certificate. From then on, the clients can access the server with the security provided by the certificate.

Note: If your client certificate is compromised or even if some other server in its trust-basis is compromised and you have to produce a replacement certificate, you can experience the same delay again until a new certificate is received. For more information about getting and installing server certificates, see "Creating SSL certificates for a CORBA C++ client" on page 39.

Creating and sending a certificate signing request on a CORBA C++ client: Before you begin

If you want to create a request for a certificate authority (CA)-signed certificate from a key database file, you must have created the key database already. The request is issued against that key database and the certificate must be integrated into that database. For information about creating a key database, see "Creating a key database for a CORBA C++ client" on page 38.

Steps for this task

Use this procedure to create a Certificate Signing Request (CSR). This request is sent to a CA to get a signed certificate for a C++ client that uses SSL mutual certificate authentication. You only need to complete this procedure if you want to get a signed test or production certificate from a CA.

This procedure creates the CSR file in the $WAS_HOME\etc directory. It automatically creates a corresponding private key for the client that remains in your keyring file database. You do not transmit the certificate's private key to the CA, therefore the private key remains entirely in your possession at all times.

To create a Certificate Signing Request (CSR), complete the following steps:

1. Start the IBM Key Management tool and use it to open the key database file or cryptographic token from which you want to create the certificate request. If you want to create a request from a key database file, complete the following steps:
   a. Start the IBM Key Management tool as described in "Starting the IBM Key Management tool" on page 39.
   b. Open the key database file (filename.kdb) for the client for which you want to request a CA-signed certificate. To open the key database file, either click **Open a key database file** or select **Key Database File > Open** from the menu bar. Type the name and location of the key database file at the prompt.
   c. Click **OK**. This opens the **Password Prompt** window.
   d. At the prompt, type the password that you specified when you created the CMS key database file.
   e. Click **OK**.
2. Select **Personal Certificate Requests** from the pull-down under Key database content in the middle of the window. This updates the **IBM Key Management** window to list any existing personal certificate requests.
3. Click **New**. The **Create New Key and Certificate Request** window is displayed.
4. Fill in the following certificate attributes:

   Key Label
   The key label is used to uniquely identify the certificate within the key database file. For a CORBA C++ client, there typically is only one

certificate in each key database file, so you can assign any label value. However, it is good practice to use a unique label, perhaps related to the server or client name.

Key size
Key size is the size of the key used to digitally sign and authenticate certificates. The default is 1024. For 128-bit cipher algorithms, the value can be either 512 or 1024. For 56-bit cypher algorithms, the value must be 512.

Common Name
This is the primary, universal identity for the certificate that uniquely identifies the principal that it represents.

Notes:

a. For some CAs, it is required that you include the fully qualified name of your host in the common name. For example, VeriSign does not sign your certificate unless the domain portion of the host name is owned by your organization. Also, some CAs have restrictions on the characters that you can use for the common name in a certificate signing request (CSR). For example, your CA might require that the common name be a fully qualified domain name without the characters ?*', ??', ?:', ' ' (space), or the strings ?http://? or ?:port number?. Check the format that your CA requires before continuing to complete your CSR.
b. Any slash character used after host_name in the common name must be a back-slash (\), even on Unix hosts.

Organization
This is the name of your organization.

Note: Some Certificate Authorities (CAs) might require that you complete the "optional" fields in a certificate signing request (CSR) and that you completely spell out the state or province. Check with your intended CA for any such restrictions before continuing to complete your CSR. For example, your CA might require that the location, state/province, and zip code fields be completed for all organizations outside the US or Canada.

Organization Unit
(Optional) This is the name of your organization unit.

Locality
(Optional) This is the name of the location (city).

State/Province
(Optional) This is the name of the state/province.

Zipcode
(Optional) This is the zip code.

Country
This menu is the two-letter identifier of the country in which the server belongs.

The name of the file in which to store the certificate request
Type the full path name of the file in which you want to store the CSR. Typically, this is something like the following:

Websphere_key_dir\common_name.arm, where: Websphere_key_dir is the WebSphere default keyrings directory (for example, $WAS_HOME\etc).

common_name
This is the common name of the client for which you are getting a certificate. The standard extension used for a file in which you want to store a CSR is .ARM.

Results

When you have filled in all of the required fields for the certificate, click **OK**. When the CSR file is created, you are notified and prompted to get the certificate signed.

What to do next

Send the file to a CA to request a new digital certificate, or cut and paste the request into the request forms of the CA's Web site. After the CA sends you a new CA-signed certificate, you need to add it to the key database from which you generated the request. Continue with the next step in the overview procedure article, "Enabling SSL certificate security between a CORBA C++ client and an EJB server" on page 38.

Integrating a CA-signed certificate into a CORBA C++ client key database file: Before you begin

You must have requested and received a new signed certificate from a certificate authority as described in "Creating and sending a certificate signing request on a CORBA C++ client" on page 43. After the CA sends you a new signed certificate, you need to add it to the key database file from which you generated the request.

Steps for this task

To receive a CA-signed certificate into a key database file, follow these steps:

1. If you receive an e-mail from a certificate authority containing your new CA-signed certificate, save that mail in a file, for example, filename.ARM.
2. Start the IBM Key Management tool and use it to open the key database file or cryptographic token from which you created the certificate request. If you created a request from a key database file, complete the following steps:
   a. Start the IBM Key Management tool as described in "Starting the IBM Key Management tool" on page 39.
   b. Open the key database file (filename.kdb) for the client for which you want to request a CA-signed certificate. To open the key database file, either click **Open a key database file** or select **Key Database File > Open** from the menu bar. Type the name and location of the key database file at the prompt.
   c. Click **OK**. This opens the **Password Prompt** window.
   d. At the prompt, type the password that you specified when you created the CMS key database file.
   e. Click **OK**.
   f. Select **Personal Certificate Requests** from the pull-down under Key database content in the middle of the window.

g. To receive your signed certificate into the key database file, click **Receive**. The **Receive Certificate from a File** window is displayed.
h. In the **Receive Certificate from a File** window, type the **Certificate file name** and **Location** for the new digital certificate, or click **Browse** to select the name and location.
i. To receive your certificate, click **OK**. The **Enter a Label** window is displayed.
j. Type a label, such as `Production Certificate for MyWeb at My Company`, for the new digital certificate and click **OK**. The **IBM Key Management** window is displayed. The **Personal Certificates** field shows the label of the new digital certificate you added.

What to do next

Continue with the next step in the overview procedure article, "Enabling SSL certificate security between a CORBA C++ client and an EJB server" on page 38.

Extracting a certificate from a CORBA C++ client key database file: Before you begin

The key database file must already exist and contain the certificate to be extracted.

Use this procedure to extract a certificate (which includes its public key) from the (source) key database file to be added as a signer certificate in the (target) key database file.

This procedure forms the first stage of copying a certificate from one key database file to another. If the target key database file already contains the signer certificate of the certificate authority used to sign the certificate that is to be copied, you do not need to add the certificate to the target key database file. In general, you need to complete this procedure only for a self-signed certificate to support SSL between a client and a server, as in the following cases:

- For a CORBA C++ client, if the client and target server are configured to enable SSL client certificate association.
- The C++ client is to use the client certificate to create secure connections with the server.

  Note:

  1. Extracting a certificate from one key database file and adding it to another key database file is not the same as exporting the certificate and then importing it. Exporting a certificate copies all of the certificate information, including its private key, and is normally only used if you want to copy a personal certificate into another key database file as a personal certificate.
  2. If a certificate is self-signed, you need to extract the certificate (which includes its public key) and add it into the target key database file.
  3. If a certificate is CA-signed, verify that the CA certificate used to sign the certificate is listed as a signer certificate in the target key database file. For example, to check that the CA certificate for a server certificate is in a client key database file, complete the following steps:
     a. Write down the label names of the CA certificates from the client key database's signer certificates.
     b. Verify the client's signer certificates against the list of signer certificates in the server key database file.

c. If the CA certificate used to sign the client certificate is not listed in the server key database file, you can use this procedure to extract the CA certificate from the client key database file and add it to the server key database file.

To extract a certificate from a key database file (into a temporary file), complete the following steps:

Steps for this task

1. Start the IBM Key Management tool as described in "Starting the IBM Key Management tool" on page 39.
2. Open the key database file (filename.kdb) for the server or client for which you want to request a CA-signed certificate. To open the key database file, either click **Open a key database file** or select **Key Database File > Open** from the menu bar. Type the name and location of the key database file at the prompt.
3. Click **OK**. This opens the **Password Prompt** window.
4. Type the password used to create the key database file.
5. Click **OK**.
6. The title bar of the **IBM Key Management** window shows the name of the key database file that you selected and indicates that the key database file is open and ready.
7. Beneath Key Database Context, select **Personal Certificates** (the default) from the **Certificate types** menu. To copy a signer certificate from the key database file, click **Signer**.
8. Select the certificate you want to extract.
9. Click **Extract certificate**. (If you selected Signer, click **Extract**.) The **Extract a Certificate to a File** window is displayed. Proceed with the remaining steps.
10. Click **Data type** and select a data type, such as **Base64-encoded ASCII data** (the default). The data type needs to match the data type of the certificate stored in the certificate file. The IBM Key Management tool supports Base64-encoded ASCII files and binary DER-encoded certificates.
11. Type the certificate file name and location where you want to store the certificate, or click **Browse** to select the name and location.
12. Click **OK**. The certificate is written to the specified file and the **IBM Key Management** is displayed.

What to do next

Continue with the next step in the overview procedure article, "Enabling SSL certificate security between a CORBA C++ client and an EJB server" on page 38.

Adding a signer certificate into a CORBA C++ client key database: Before you begin

- The key database file must already exist. If you have not created the key database, see "Creating a key database for a CORBA C++ client" on page 38.
- The signer certificate to be added to the client key database must already have been extracted from the server key database into a temporary file.

Use this procedure to add a signer certificate (which includes its public key) into the key database file for a client.

This forms the second stage of copying a certificate from one key database to another. If the client key database already contains the signer certificate of the CA used to sign the certificate that is to be copied, you do not need to add the certificate to the key database. In general, you need to complete this procedure only for a self-signed certificate to support SSL between a client and a server, as in the following cases:

- For a CORBA C++ client, if the client and target server are configured to enable SSL certificate client association.

To add a signer certificate to a client key database, complete the following steps:

Steps for this task

1. Start the IBM Key Management tool as described in "Starting the IBM Key Management tool" on page 39.
2. Open the client key database file. For example, filename.kdb, for a C++ client.
3. Type the password for the key database file, then click **OK**. The title bar of the IBM Key Management window shows the name of the key database file that you selected and indicates that the key database is open and ready.
4. In the **Key Database Content** field, select **Signer** from the menu.
5. Click **Add**. This opens the **Add CA Certificate from a File** window.
6. Specify the data type, certificate file name, and location of the file that you specified when you extracted the certificate to be added to the key database.
7. Click **OK**. This opens the Enter a Label window.
8. Specify a name for the certificate.
9. Click **OK**. At this point, the target key database should contain the new certificate (as shown in the IBM Key Management window).
10. If no longer needed, close the IBM Key Management window.

What to do next

Continue with the next step in the overview procedure article, "Enabling SSL certificate security between a CORBA C++ client and an EJB server" on page 38.

SSL security for CORBA C++ clients

Secure Sockets Layer (SSL) is an authentication protocol introduced as an IETF standard. WebSphere supports SSL-based mutual authentication between IBM WebSphere Application Servers and CORBA C++ clients.

Both CORBA C++ clients and EJB servers have a key database file, which is a CMS key database file generated by the IBM Key Management Tool. In the client's key database file, a portion contains the server certificate's public key (or its CA certificate as a signer). In the server's key database, there is a truststore file that contains each client's certificate public key (or its CA certificate as a signer). SSL mutual authentication is performed so that the client uses the server's certificate to authenticate the server and the server uses the client's certificate to authenticate the client.

The SSL support provided by WebSphere for CORBA C++ clients uses the GSKit SSL library at the C++ clients and IBM JSSE at the server. Both SSL libraries are shipped with WebSphere.

To enable SSL certificate-based authentication, you must create a server certificate for each server you want to authenticate and a client certificate for each client that

you want to authenticate. A server certificate, along with its corresponding private key, must be placed in a key database file at the server. The server uses this key database file to present itself to any clients that want to authenticate the server. Similarly, the client certificate and key must be placed in a key database file at the client. The client uses its key database file to present itself to servers that want to authenticate the client.

For more information about WebSphere C++ client use of SSL, see "CORBA C++ client: SSL and certificates".

CORBA C++ client: SSL and certificates

The Secure Sockets Layer (SSL) protocol is popular in the Internet industry, primarily because of its use of public-key certificates as a means of authenticating principals. These certificates represent a possession-based authentication scheme; you are deemed to be who you claim to be (you are authentic) because you possess an appropriate certificate. The certificate identifies you and, through the encryption techniques used to create it, can be proved to be legitimate.

With SSL and public-key certificates, you trust the Certificate Authority (CA) that signed any certificates presented to you. If you do not trust the CA, then you do not trust the certificate, and by extension you do not trust the principal it represents. You only need to know about the relatively small number of CAs that you trust. As such, you can avoid building a large, central database of registered users (a user registry), which is essential in an environment that might consist of millions of end-users, such as the Internet.

Note: An important thing to understand is that, because SSL-based authentication is based on possession, anyone who can copy your certificate (actually the private-key that protects your certificate) is able to masquerade as you. For this reason, it is very difficult to use SSL-based authentication to perform delegation, that is, to perform down-stream method requests under your identity and authority. Further, since most enterprise information systems need to control access to their resources on an individual, group, or role basis, you often have to create some amount of central (or at least centrally managed) user database information in the form of Access Control Lists (ACLs).

SSL uses certificates for public-key cryptography. Public-key cryptography uses two different cryptographic keys: a private key and a public key. Public-key cryptography is also known as asymmetric cryptography because you can encrypt information with one key and decrypt it with the complement key from a given public-private key pair. GSKit 5 provides the standard SSL support (software cryptography) as in previous versions of WebSphere.

A certificate is your key into a resource. Certificates are signed by an issuing CA and validated either on an individual basis or on a group basis. The larger the grouping, the more certificates are impacted if the certificate becomes compromised.

CORBA C++ client: Structure of a certificate

WebSphere supports the mutual authentication of servers and SSL-enabled CORBA C++ clients, based on server certificates and client certificates.

A certificate is composed essentially of two major parts: the certificate itself (the public part) and its corresponding private key. As with public-key encryption, you can freely give out the certificate (the public part), if you keep secure the private-key part.

The public portion of the certificate is also composed of two parts: information that identifies you, for example, your name and address, and a certificate chain. The certificate chain is the certificate that identifies the authority that issued (signed) your certificate, the certificate of the authority that signed their certificate (authorized them to be a Certificate Authority), and so on. The certificate chain ends with one or more self-signed certificates, each an authority that authorized itself to be a Certificate Authority. These are known as the root authorities. For more information about certificate chains, see "CORBA C++ client: Certificate chains" on page 51.

Even when using public-key certificates to authenticate servers within the SSL-based authentication model, those servers also have security credentials. Creating a certificate for a server is secondary and must only be done if SSL-enabled clients communicate with the server. In this case, WebSphere assumes that you have created and installed a unique SSL certificate for each server. There are many choices to make about the procedures you use to generate and maintain server certificates. "Creating SSL certificates for a CORBA C++ client" on page 39 describes one way to create and administer server certificates, but there are many ways for you to tailor these procedures to match the specific needs of your enterprise and its administration policies.

WebSphere provides a test certificate that you can use during development or testing so that you can avoid any delays in setting up security for your application servers.

Note: It is very important that you understand that this is an insecure certificate; it is self-signed with a relatively weak key and does not uniquely distinguish the servers where it is used. Therefore, this test certificate should not be used in a production environment where security integrity is required.

CORBA C++ client: Certificate authorities

A certificate authority (CA) is someone that is assigned the responsibility for signing your certificates. The process of signing the certificate is an act that warrants the authenticity of the principal represented by that certificate. In other words, the certificate authority has done whatever it takes to ensure that the requesting principal is who they claim to be, and sealed that authenticity by digitally signing the certificate. The CA signs the principal's certificate with their own (the CA's) certificate.

Anyone can be a certificate authority. Most often you either trust a particular administrative group within your enterprise to be the CA, or in some cases you might prefer to delegate this responsibility to any one of a number of commercial CAs.

In the normal process, if you want to obtain a digitally-signed certificate that represents who you are, you begin by producing a Certificate Signing Request (CSR) using your local software. You then submit this CSR to the CA, along with any accompanying information that is needed to authenticate you. The CA does what they have to do to verify your authenticity and, if this is successful, signs your certificate and returns it to you. Often, this process can be completed electronically through either e-mail or through the world-wide-web.

Each CA has their own process. What the CA does to verify you depends on a number of conditions. Ultimately, the reputation of the CA is absolutely essential to their business success. If they fail to properly authenticate a requesting principal properly, accidentally handing out a certificate that the principal then uses to

misrepresent themselves, then the CA's reputation is at stake. You are likely to loose faith in that CA and no longer trust the certificates that they sign. Consequently, principals will soon stop using that CA to sign their certificates, because you have stopped recognizing their authority. As a result, CAs typically go to great lengths to ensure the authenticity of their requesting principals. They may perform background checks, verify credit histories, post office registries, business records, and even criminal histories. This process is in many ways equivalent to obtaining a passport; the Passport Office requires that you provide some proof of your legitimacy, such as a birth certificate before issuing you a passport. Other countries accept that the passport proves who you are, trusting that the Passport Office has provided this assurance through its own validation checks and sealed that validity in the physical packaging of your passport.

If you establish your own internal CA (for example, within your systems management group), then you can use less complicated procedures to authenticate certificates. For example, you can verify the requesting principal in your employment records, or based on a previously defined planning manifest, perhaps by listing all of the server principals that you plan to deploy in your enterprise as part of some major application delivery plan.

CORBA C++ client: Certificate chains

Often a certificate authority (CA) obtains its authority to sign certificates from another CA. This is particularly common for in-house CAs. In other words, you might deem a certificate signed by a particular CA to be legitimate, not because of who the CA is, but by virtue of that CA having been authenticated by another CA. In this case, the certificate of the requesting principal is signed with the certificate of its CA. That CA's certificate is signed with the certificate of the authorizing CA; the CA that authorized the first CA to sign the certificate for the requesting principal. This is referred to as a certificate chain. A certificate chain can be long.

Each successive certificate in a certificate chain represents the next higher certificate group. You can even create arbitrary, intermediate CA certificates that you use to sign such groups of principal certificates. Certificate groups are an important element in the organization of hierarchical trust relationships. For example, you can combine a set of servers into a server group. Assuming you assign each server its own certificate (each representing the server principal of each server) you can sign each of those server certificates with the same common group certificate. The certificate of that server group can then be combined with the signing certificates of other server groups, and all signed with another common certificate representing the super-group. This can go on until eventually there is one or more certificates that are self-signed. These self-signed certificates are referred to as root-certificates; they basically represent the root of the certificate hierarchy below them.

When verifying the validity of a certificate, you must decide who in the certificate chain you are going to trust. You could form your trust in individual certificates, in some intermediate Certificate Authority, or the root authority. We refer to this as the trust-basis for validating certificates. If your trust basis is in individual certificates, then you must retain a list of each individual certificate that you want to recognize. If your trust basis is in the root authority, then you only have to retain the certificate of that authority.

If you ever lose trust in the certificate authority, basically if any certificate issued by that authority is compromised, then you must change the certificate of that authority, and reissue every certificate issued by that authority previously. If your trust was in the root authority, this can be a major exercise. Alternatively, you can

reach some balance by establishing your trust-basis in some intermediate authority. (This reduces the impact from losing the trust-basis in that intermediate authority to only the certificates issued by that authority.)

Specifying run-time properties for CORBA C++ clients and servers

This topic provides an overview of how to specify the run-time properties for C++ clients and servers. There are three ways to specify Object Request Broker (ORB) run-time properties:

- Properties can be stored in a file whose name is specified by setting the WASPROPS environment variable.
- Properties can be specified by setting environment variables.
- Properties can be set by passing argument strings to the `CORBA::ORB_init()` function.

Each of these methods is described in detail below.

Specifying properties in a file. To specify ORB run-time properties in a file, complete the following steps:

1. Create a simple text file, for example, client.props.
2. Add the desired run-time properties to the file. The properties and values that you choose depend on your use of the ORB and are selected from the properties listed in the reference topic Runtime properties for CORBA clients and servers.

   Here is an example of a properties file that might be used for a client:

   Note: The lines beginning with a pound sign (#) are comment lines and are ignored.

   ```
   # Set the bootstrap host and port.
   com.ibm.CORBA.bootstrapHostName=host1.company.com
   com.ibm.CORBA.bootstrapPort=9000

   # Increase the request timeout from 3 minutes to 5 minutes (300 seconds).
   com.ibm.CORBA.requestTimeout=300

   # Load the transactions service during process initialization.
   com.ibm.CORBA.transactionEnabled=yes
   ```

   that lines beginning with a pound sign (#) are comment lines and are ignored.
3. Set the WASPROPS environment variable to the name of the properties file. An example follows:

   ```
   export WASPROPS=/dir1/dir2/client.props
   set WASPROPS=c:\dir1\dir2\client.props
   ```

   In fact, you can specify multiple properties files by setting WASPROPS to a blank-separated list of filenames. This would be useful if you need to store commonly used properties in one file (for example, common.props), store client-specific properties in another file (for example, client.props), and store server-specific properties in yet another file (for example, server.props). For the client, you might set WASPROPS like this:

   ```
   export WASPROPS=/dir1/dir2/common.props /dir1/dir2/client.props
   set WASPROPS=c:\dir1\dir2\common.props c:\dir1\dir2\client.props
   ```

   For the server, you might set WASPROPS like this:

```
export WASPROPS=/dir1/dir2/common.props /dir1/dir2/server.props
set WASPROPS=c:\dir1\dir2\common.props c:\dir1\dir2\server.props
```

Specifying properties with environment variables. To specify ORB run-time properties with environment variables, you first must set the WASGETENV environment variable to 1 (one) like this:

```
export WASGETENV=1
set WASGETENV=1
```

This indicates to the ORB to obtain properties first from the environment, then from any properties file or files provided to the ORB.

Next, you need to set environment variables that correspond to the desired run-time properties. To determine the correct variable name from the run-time property name, follow these steps:

1. Remove the com.ibm.CORBA. prefix (requestTimeout) starting with the full property name (for example, com.ibm.CORBA.requestTimeout).
2. Change the result to uppercase (REQUESTTIMEOUT).
3. Add the WAS prefix (WASREQUESTTIMEOUT).
4. Use the resulting string as the environment variable name, like this:

   ```
   export WASREQUESTTIMEOUT=300
   set WASREQUESTTIMEOUT=300
   ```

5. Convert any periods (.) to underscores (_) in the string from the preceding step. For example, if you need to set the com.ibm.CORBA.logger.fileDetail property, use the WASLOGGER_FILEDETAIL environment variable.

The following shows how the previous two properties might be set using environment variables:

```
export WASGETENV=1
export WASREQUESTTIMEOUT=300
export WASLOGGER_FILEDETAIL=no

set WASGETENV=1
set WASREQUESTTIMEOUT=300
set WASLOGGER_FILEDETAIL=no
```

Specifying properties with CORBA::ORB_init() argument strings. You can set properties by passing argument strings to the CORBA::ORB_init() function using the argc and argv parameters. The CORBA::ORB_init() function supports two ways to set properties:

- To set a property, use an argument string in the form, -Dproperty_name=property_value. For example, to set the request timeout value, you might use the argument string, -Dcom.ibm.CORBA.requestTimeout=300.
- To pass the name of a properties file to the ORB, use an argument string in the form, -PropertiesFile=file_name. For example, if you stored your properties in a file called /dir1/dir2/client.props, you might use the argument string, -PropertiesFile=/dir1/dir2/client.props.

The following is a code fragment that illustrates how to use these argument strings:

```
.
.
.

char *argList[] = {
   "-Dcom.ibm.CORBA.requestTimeout=300",
```

```
   "-Dcom.ibm.CORBA.logger.fileDetail=no",
   "-PropertiesFile=/dir1/additional.props"
   };

int argCount = 3;

CORBA::ORB_ptr orb_ptr = CORBA::ORB_init ( argCount, argList, "DSOM" );

.
.
.
```

Run-time properties for CORBA clients and servers

This topic provides reference information about the properties that you can set to control the run-time environment of CORBA C++ clients and servers. Each property is listed in the following form:

property_name=value_type
[default_value] A description of the property where

- property_name is the name of the property.
- value_type is the type of value that the property can have.
- [default value] is the default value of the property (only shown if the property has a default value).

Client and Server general ORB run-time properties

You can specify the following general ORB run-time properties for both clients and servers:

com.ibm.CORBA.bootstrapHostName=host_name
[current host] The name of the host on which the bootstrap agent runs. This host name is used with the **com.ibm.CORBA.bootstrapPort** property to access the bootstrap agent.

com.ibm.CORBA.bootstrapPort=port_number
[2809] The number of the port that the bootstrap agent uses to communicate with clients and servers. This property is an integer port number, in the range 0 through 65536. This port number is used with the **com.ibm.CORBA.bootstrapHostName** property to access the bootstrap agent.

com.ibm.CORBA.requestTimeout=number_seconds
[180] The time, in seconds, that a request waits for a response before timing out and issuing an error that indicates NO RESPONSE. If this property is set to 0 (zero), requests wait indefinitely until a response is received. This property is integer value greater than or equal to 0.

com.ibm.CORBA.initServices=list_of_service_names
This property specifies the set of *services* which should be loaded by the ORB during process initialization. This property is a blank-separated list of words which represent service names. For each service name listed in this property, the ORB expects to find a corresponding dllName property (see below) which specifies the name of the DLL or shared library to be loaded for that service during process initialization.

com.ibm.CORBA.service_name.dllName=library_name
This property specifies the name of the DLL or shared library to be loaded by the ORB during process initialization to enable the use of a particular service. For example, if you wanted to use services called my_service1 and my_service2, then you would set the following properties:

```
com.ibm.CORBA.initServices=my_service1 my_service2
com.ibm.CORBA.my_service1.dllName=service1.dll
com.ibm.CORBA.my_service2.dllName=service2.dll
```

Note: The above example assumes that the `my_service1` service is implemented in service1.dll, and the `my_service2` service is implemented in service2.dll.

com.ibm.CORBA.eMNumThreads=`number_threads`
[3] The size of the ORB's event manager thread pool. The event manager is the component which handles all incoming TCP/IP communications. Increasing the value of this property may improve performance for servers which receive an excessive number of connections from clients. This property is an integer value greater than or equal to 3.

com.ibm.CORBA.maxHops=`number_of_location_forwards`
[5] The maximum number of location forwards (i.e. *hops*) that the client or server should follow before aborting object location. This property is an integer greater than or equal to 1.

com.ibm.CORBA.processAlias=`string`
[DefaultClient] This property is used to provide a logical name for the client or server process. This property can be set to any name which is meaningful to the user. For servers, the ORB will, by default, set the process alias so that it matches the name of the implementation def used for the server, so it should not be necessary to set this property for servers. For clients, however, the ORB does not set a default value for this property. Note that the process alias is used only by the RAS message logger in the message header to indicate the logical name of the process which logged the message.

com.ibm.CORBA.processType=`string`
[client] This property is used to specify whether the process is a client or server. The valid values for this property are **client** and **server**. This property is only used by the RAS message logger in the message header to indicate the type of process (client, server) which logged the message. Note that the default value of this property is **client**, but the ORB will automatically set this property to **server** if the **CORBA::ORB::BOA_init()** function is called (which means that the process must be a server).

Server-specific ORB run-time properties

You can specify the following ORB run-time properties for servers:

com.ibm.CORBA.exportedHostName=`host_name`
The hostname string that should be included in object references (IORs) exported by the server. The value is a TCP/IP hostname of up to 256 ASCII characters. Normally, the C++ ORB uses the dotted-decimal form of the hostname, but this property can be used to override the dotted-decimal form with an alternate name. This might be useful in situations where the server is operating behind a firewall, and you do not want the dotted-decimal hostname published outside the firewall.

com.ibm.CORBA.serverListenPort=`port_number`
[0] The port number on which the server should listen for incoming requests from clients. This enables the server to support a static firewall scenario, in which the firewall enables use of a set of secure ports. If you leave this property to default to 0 (zero), the server is automatically assigned a number for it's listening port.

com.ibm.CORBA.iiopVersion=iiop_version
[1.2] The default GIOP/IIOP protocol version that the ORB uses to export object references. This property effectively specifies the highest level of GIOP/IIOP that the ORB should support. It can be used to downgrade the level of GIOP/IIOP used by the WebSphere ORB in order to enhance interoperability with non-WebSphere ORBs. This property can be set to 1.0, 1.1, or 1.2 (for IIOP 1.0, IIOP 1.1, or IIOP 1.2, respectively).

com.ibm.CORBA.numWorkerThreads=number_threads
[3] The size of the ORB thread pool in which servant objects process method invocations. This property is an integer greater than 0. When the ORB receives a request, it activates a thread from the appropriate pool for the target object to service the request.

com.ibm.CORBA.workerThreadStackSize=number_bytes
[65536] The size, in bytes, of the thread stack used when creating new threads in the server's thread pool. This property is an integer greater than 65536 bytes.

com.ibm.CORBA.implDbDir=directory_name
This property specifies the name of the directory containing the ORB's implementation repository. If this property is not specified, then the ORB will store the implementation repository in the **implrep** directory within the ORB's top-level installation directory (identified by the **WASORBTOP** environment variable).

com.ibm.CORBA.irUserid=string
This property specifies the user id to be used by the Interface Repository (IR) server when accessing a remote database. This property is only used if the IR database is configured as a remote database.

com.ibm.CORBA.irPassword=string
This property specifies the password to be used by the Interface Repository (IR) server when accessing a remote database. This property is only used if the IR database is configured as a remote database.

com.ibm.CORBA.TCPIP.lsdPort
[2809] The listening port to be used by the location service daemon (wasorbd).

com.ibm.CORBA.TCPIP.lsdHostName
[current host]The name of the host on which the location service daemon (wasorbd) is running.

Client and server codeset translation run-time properties

You can specify the following codeset-related ORB run-time properties for both clients and servers:

com.ibm.CORBA.translationEnabled=yes_no
[no] This property specifies whether or not the client or server should perform codeset translation for character data received in remote messages. This property can have the following values:

no Codeset translation is not performed, and the other codeset-related properties are ignored.

yes Codeset translation is performed. Also, consider the following points:

- If you do not specify a value for the com.ibm.CORBA.nativeWCharSet property, then the ORB

assumes that the application will not use wide character data because a wide character code set was not specified.

- Codeset translation is not supported by the IIOP 1.0 protocol.

com.ibm.CORBA.nativeCharCodeset=codeset_name_or_number
This property specifies the native codeset used by the application for character and string data. If this property is not set, then the ORB will attempt to determine the native character codeset itself. If this property is set, the value should be one of the following:

- A decimal integer value representing an OSF (Open Software Foundation) codeset number (for example, 65537).
- A hexadecimal integer value representing an OSF codeset number (for example, 0x0010001).
- A string specifying the primary name of the codeset (for example, ISO-8859-1)

com.ibm.CORBA.nativeWCharCodeset=codeset_name_or_number
This property specifies the native codeset used by the application for wide character data This property needs to be set only if the application actually uses wide character data within remote method invocations. If set, the value should be a decimal or hexadecimal OSF codeset number, or a codeset name, similar to the **nativeCharCodeset** property above. **Note:**If this property is set, but does not match the native process codeset of the operating system, codeset translations of wide character data will likely fail.

com.ibm.CORBA.charCCS=codeset_list
This is a blank-separated list of the codesets to and from which the process can translate character data. Each codeset is a decimal or hexadecimal OSF codeset number, or a codeset name.

com.ibm.CORBA.wcharCCS=codeset_list
This is a blank-separated list of the codesets to and from which the process can translate wide character data. Each codeset is a decimal or hexadecimal OSF codeset number, or a codeset name.

Client and server trace run-time properties

You can set the following trace-related ORB run-time properties for both clients and servers. Please note that these trace properties should only be used in situations where you need to diagnose a problem or otherwise capture very detailed information about the operation of the ORB and the application. The use of these trace properties will result in performance degradation, so they should not be used during normal operation.

com.ibm.CORBA.orbCommunicationsTraceLevel=trace_level
[none] This property controls the amount of trace data that is logged for ORB communications between the current process and other processes. This property has one of the following values. Each succeeding value increases the amount of information that is logged.

none or 0 (zero)
Trace data is not logged.

basic or 1
This setting results in a hexadecimal dump of the contents of each GIOP message that is sent or received by the process.

intermediate or 2
This setting includes the **basic** setting above, plus brief formatting of each GIOP message sent or received by the process. Brief formatting includes some basic information about the GIOP message: the GIOP version number, the message type (e.g. request, reply, etc.), the byte order (i.e. big endian, little endian, etc.), the message length, and the request id value. This is usually enough information to perform problem diagnosis in most situations.

advanced or 3
This setting includes the **intermediate** setting above, plus detailed formatting of each GIOP message sent or received by the process. Detailed formatting includes the formatting of object references, tagged components, tagged profiles, and service contexts contained in GIOP messages.

com.ibm.CORBA.orbMethodDispatchTraceLevel=`trace_level`
[none] This property is used to enable the tracing of methods dispatched by the object adapter within a server. This property has one of the following values.

none or 0 (zero)
Trace data is not logged.

basic or 1
This setting turns on method dispatch tracing.

com.ibm.CORBA.orbNLSTraceLevel=`trace_level`
[none] This property controls the amount of codeset translation-related trace data that is logged by the ORB. This property has one of the following values:

none or 0 (zero)
Trace data is not logged.

basic or 1
This setting enables the tracing of basic operations related to codeset conversions, including information about codeset negotiation between a client and server, as well as the logging of configuration information (i.e. the native codesets configured for the client or server).

intermediate or 2
This setting provides the **basic** setting above, plus detailed information about codeset conversion operations.

Client and server message logging properties

The following properties are supported by the message logger implementation provided with the ORB. These properties can be set for both clients and servers.

com.ibm.CORBA.logger.directoryName=`directory_name`
[service] The name of the directory which contains the message log and trace files. The value of this property is appended to the top-level installation directory, as specified by the **WASORBTOP** environment variable.

com.ibm.CORBA.logger.logFileName=`file_name`
[activity.txt] The name of the message log file produced by the message logger. This file is stored in the directory specified by the **com.ibm.CORBA.logger.directoryName** property.

com.ibm.CORBA.logger.maxLogFileSize=number_kbytes
[1024] The maximum size (in kilobytes) of the message log file. A value of 0 indicates no limit. Once the message log file reaches its maximum size, it is renamed to an alternate name, then the ORB continues to log messages to the orginal message log file. The alternate name is obtained by adding a **.prev** suffix to the message log filename. For example, if messages are being logged to a file called **activity.txt**, then when this file exceeds the maximum file size, it is renamed to **activity.txt.prev**, then the ORB continues logging messages to **activity.txt**.

com.ibm.CORBA.logger.logToFile=yes_no
[no] This property specifies whether or not the message logger should write messages to the message log file. The valid values are **yes** and **no**.

com.ibm.CORBA.logger.logToConsole=yes_no
[no] This property specifies whether or not the message logger should display messages on the console (i.e. the process's standard output). The valid values are **yes** and **no**.

com.ibm.CORBA.logger.logFilter=log_filter_value
[0] This property is used to selectively filter messages based on the message severity. The following values can be used:

0 All messages (informational, warning, error) are logged. This effectively turns off filtering.

1 Only warning and error messages are logged. This setting filters out informational messages.

2 Only error messages are logged. This setting filters out informational and warning messages.

com.ibm.CORBA.logger.fileDetail=yes_no
[yes] This property specifies whether detailed or brief messages are written to the message log file. This property can have the following values:

no Brief messages are written to the message log. A brief message consists of only the timestamp and message text.

yes Detailed messages are written to the message log. A detailed message includes the timestamp and message text, as well as other information such as the process id, the thread id, hostname, message severity, and source filename and line number which logged the error.

com.ibm.CORBA.logger.consoleDetail=yes_no
[no] This property specifies whether detailed or brief messages are displayed on the console (i.e. the process's standard output). This property can have the following values:

no Brief messages are diplayed on the console.

yes Detailed messages are displayed on the console.

com.ibm.CORBA.logger.traceToConsole=yes_no
[no] This property specifies whether or not the message logger should display trace messages on the console (i.e. the process's standard output). Valid values for this property are **yes** and **no**.

com.ibm.CORBA.logger.traceToFile=yes_no
[yes] This property specifies whether or not the message logger should write trace messages to a trace file. Valid values for this property are **yes** and **no**.

com.ibm.CORBA.logger.traceFileDetail=yes_no
[yes] This property specifies whether detailed or brief trace messages are written to the trace file. This property can have the following values:

no Brief trace messages are written to the trace file.

yes Detailed trace messages are written to the trace file.

com.ibm.CORBA.logger.traceConsoleDetail=yes_no
[no] This property specifies whether detailed or brief trace messages are displayed on the console (i.e. the process's standard output). This property can have the following values:

no Brief trace messages are diplayed on the console.

yes Detailed trace messages are displayed on the console.

com.ibm.CORBA.logger.multipleTraceFiles=yes_no
[no] This property specifies whether or not multiple trace files should be produced for each process (one trace file per thread of execution). If this property is set to **yes**, then the message logger will produce multiple trace files per process (i.e. one trace file per thread of execution within the process). If the property is set to **no**, then the message logger will produce one trace file per process (i.e. the trace messages for all threads are logged to a single file for that process).

Client and server transaction support properties

You can specify the following transaction support properties for clients:

com.ibm.CORBA.transactionEnabled=yes_no
[yes] This property specifies whether or not the transaction service is enabled for this client. The possible values for this property are **yes** or **no**.

com.ibm.CORBA.transactions.defaultTimeout=number_seconds
[300] The default time, in seconds, after which a top-level transaction is rolled back if it has not completed. Because a transaction may hold locks on database records, it is important to ensure that all transactions complete within a reasonable period of time. This is especially important in a distributed environment where a transaction may be originated by a non-recoverable client. If such a client dies without ending all the transactions it started, then those transactions should have a period of time after which they are automatically rolled back by the server on which they were created.

This timeout value is the time from when the originator started the top-level transaction to the time when the originator must request that the transaction be committed or rolled back. It is an integer number of seconds greater than 0. If you set this property to 0 (zero), the top-level transaction never times out in the lifetime of the server on which the transaction was created.

If the application server's default transaction timeout is set to 0 (zero), transactions started using the CosTransactions::Current interface do not have a time limit set. An application program can set a time limit by calling the set_timeout() operation on the CosTransactions::Current object, passing the time limit required as a parameter.

com.ibm.CORBA.transactions.deferredBegin=deferbegin
[always] This property is used to control whether clients should attempt to defer the begin of transactions until the first remote business method is

called. In general, it is desirable for clients to have a transaction created on the same application server as at least one of the Enterprise JavaBean objects. The transaction service provides the ability to defer the creation of a transaction until the first remote business method is called, allowing the transaction service on the remote application server to create the transaction during the processing of that first business method. However, the transaction service on the remote application server must be capable of supporting this function.

This property can be set to one of the following values:

always
Always defer the creation of a transaction. This setting can be used even if IBM WebSphere Application Servers are started with their default property settings. IBM WebSphere Application Servers handle deferred begins by default, but do not indicate that they support this capability.

never Never defer the creation of a transaction. When a client application requests that a transaction be started, a transaction factory is obtained using a factory finder. The factory finder may be specified with the factory finder property. If a value is not specified for the factory finder property, an arbitrary application server is chosen. This setting may be detrimental to performance when communicating with application servers that can support deferred begin, because transactions may be created on an application server that is not otherwise involved in the transaction.

serverDependent
Defer the creation of a transaction depending on the capability of the remote server. The transaction's client code determines the capability of the remote transaction service to support the deferred begin protocol. The client determines the capability of the remote server from information contained in the target object's proxy object, so no remote flows are required for this test.

IBM WebSphere Application Servers that are started specifying the property **com.ibm.ejs.jts.jts.ControlSet.nativeOnly=false** export this information in the target object's proxy. **Note:** This is not the default startup property for the application server.

com.ibm.CORBA.transactions.factoryFinder=`string`
The name to be used to find a transaction factory for transactional clients. The value is the fully-qualified path name from the local root, which can be used in a **resolve** to get the factory desired. For example, one possible value to specify is:

```
com.ibm.CORBA.transactionfactoryFinder=node/servers/xyzServer
```

You can specify any transaction factory that is bound into the name space. Through the use of this property, you can direct the search for a particular transaction factory. The above example finds a factory on the local node (host) in server `xyzServer`. The format of the property value may be either of the following:

```
node/servers/servername
```

or

```
domain/nodes/nodename/servers/servername
```

where nodename is the name of one of the nodes in the configured WebSphere domain and servername is the name of a server. If a null value is supplied, a search starts on the local bootstrap node and if no factory is found, the search then proceeds throughout the domain, searching all configured nodes and servers for an available transaction factory. Thus a null default value on a large configuration may incur a performance overhead.

Note: This property is only used if transactions are not deferring the start of a transaction until the first business method.

Client security properties

In addition to configuring security properties as shown below, you must also configure the following property to include **security**:

```
com.ibm.CORBA.initServices=transactions olt security
```

com.ibm.CORBA.securityEnabled=yes_no

[no] This property specifies whether or not the client should be enabled with security service. This property can have the following values:

no security service is disabled.

yes security service is enabled.

If you do not specify a value, the ORB assumes that the application will not use security service.

com.ibm.CORBA.security.dllName=dll_name

This property specifies the name of the DLL or shared library to be loaded by the ORB during process initialization to enable the use of security service. The valid security dll/shared lib names for different platforms are:

```
NT  - wasscc1m.dll
AIX  - libwasscc1.so
Solaris - libwasscc1.so
Linux - libwasscc1.so
HP - libwasscc1.sl
```

com.ibm.CORBA.securityTraceLevel=0_1

[0] This property specifies whether or not trace data is logged for the security service for the current client process. This property has one of the following values:

0 Indicates trace is off.

1 Indicates trace is on.

com.ibm.ssl.protocol=protocol_level

[SSLv3] This property is used to specify the SSL protocol level. Each value must be separated by a space. Possible values: SSLv2, SSLv3, TLS.

com.ibm.ssl.keyFile=key_file_name

This property specifies the name of the key database file to use when opening an SSL connection. The full path of the key file must be specified. For information on SSL support, see "Supporting SSL by WebSphere for CORBA C++ clients" on page 37.

com.ibm.ssl.keyPassword=password

This property is used to specify the password needed to open the key

database file. For information on how to protect a plain password, see "Protecting a plain password inside the client security property file" on page 64.

com.ibm.ssl.enabledv2CipherSuites=enabled_cipher_suites for SSLv2
[SSL_RC4_128_WITH_MD5 and SSL_RC4_128_EXPORT40_WITH_MD5] This property is used to specify the ciphers to use when opening a SSL connection with SSLv2 protocol. Each cipher must be separated by a space. Note that value of this property always overrides the ciphers set by com.ibm.CSI.performMessageIntergrity and com.ibm.CSI.performMessageConfidrntialality. Possible values:

```
SSL_RC4_128_WITH_MD5
SSL_RC4_128_EXPORT40_WITH_MD5
SSL_RC2_CBC_128_CBC_WITH_MD5
SSL_RC2_CBC_128_CBC_EXPORT40_WITH_MD5
SSL_DES_64_CBC_WITH_MD5
SSL_DES_192_EDE3_CBC_WITH_MD5
```

See the SSL v2 specification for further details about these ciphers.

com.ibm.ssl.enabledv3CipherSuites=enabled_cipher_suites for SSLv3 or TLS
[SSL_RSA_WITH_RC4_128_MD5 and SSL_RSA_EXPORT_WITH_RC4_40_MD5] This property is used to specify the ciphers to use when opening a SSL connection with SSLv3 or TLS protocol. Each cipher must be separated by a space. Note that value of this property always overrides the ciphers set by com.ibm.CSI.performMessageIntergrity and com.ibm.CSI.performMessageConfidentiality. Possible values:

```
SSL_RSA_WITH_NUL_MD5
SSL_RSA_WITH_NUL_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
```

See the SSL v3 specification for further details about these ciphers.

com.ibm.CSI.performTLClientAuthenticationRequired=yes_no
[no] This property is used to determine if transport layer client authentication is required. If required, every secure socket opened between client and server authenticates using SSL mutual authentication. When performing client authentication using SSL, the client key file must have a personal certificate configured. Without a personal certificate, the client cannot authenticate to the server over SSL. If the personal certificate is a self-signed certificate, the server must contain the client's public key in the server's trust file. If the personal certificate is a CA granted certificate, the server must contain the CA's root public key in the server's trust file. When this property is specified, the associated property com.ibm.CSI.performTLClientAuthenticationSupported is ignored.

com.ibm.CSI.performTLClientAuthenticationSupported=yes_no
[yes] This property is used to determine if transport layer client authentication is supported. When performing client authentication using SSL, the client key file must have a personal certificate configured. Without a personal certificate, the client cannot authenticate to the server over SSL. If the personal certificate is a self-signed certificate, the server must contain the client's public key in the server's trust file. If the personal certificate is a CA granted certificate, the server must contain the CA's root public key

in the server's trust file. This property is only valid when SSL is supported or required. If the associated property com.ibm.CSI.performTLClientAuthenticationRequired is enabled, this property is ignored.

com.ibm.CSI.performTransportAssocSSLTLSRequired=yes_no
[no] This property is used to determine if SSL is required. When SSL is required, this client must use SSL to communicate to a server. If SSL is not supported by a server, this client does not attempt a connection to that server. When this property is enabled, the associated property com.ibm.CSI.performTransportAssocSSLTLSSupported is ignored.

com.ibm.CSI.performTransportAssocSSLTLSSupported=yes_no
[yes] This property is used to determine if SSL is supported. When SSL is supported, this client may use either SSL or TCP/IP to communicate to a server. If SSL is not supported then the client must communicate over TCP/IP to the server. It is recommended that SSL be supported so that any sensitive information is encrypted and digitally signed. When the associated property com.ibm.CSI.performTransportAssocSSLTLSRequired is enabled (set to yes), this property is ignored. In this case, SSL will always be required.

com.ibm.CSI.performMessageIntegrityRequired=yes_no
[yes] This property is used to determine if 40-bit ciphers must be used to make SSL connections. If a target server does not support 40-bit ciphers, a connection to that server fails. This property is only valid when SSL is enabled. When this property is enabled, the associated property com.ibm.CSI.performMessageIntegritySupported is ignored.

com.ibm.CSI.performMessageIntegritySupported=yes_no
[yes] This property is used to determine if 40-bit ciphers may be used to make SSL connections. If a target server does not support 40-bit ciphers, a connection may be made using only digital signing ciphers. This property is only valid when SSL is enabled. This property is ignored if the associated property com.ibm.CSI.performMessageIntegrityRequired is enabled.

com.ibm.CSI.performMessageConfidentialityRequired=yes_no
[no] This property is used to determine if 128-bit ciphers must be used to make SSL connections. If a target server does not support 128-bit ciphers, a connection to that server fails. This property is only valid when SSL is enabled. When this property is enabled, the associated property com.ibm.CSI.performMessageConfidentialitySupported is ignored.

com.ibm.CSI.performMessageConfidentialitySupported=yes_no
[yes] This property is used to determine if 128-bit ciphers may be used to make SSL connections. If a target server does not support 128-bit ciphers, a connection may be made at a lower encryption strength. This property is only valid when SSL is enabled.

Note: If none of the properties below are configured, the default cipher strength will always be "confidentiality required", which uses 128-bit ciphers:

```
com.ibm.CSI.performMessageIntegrityRequired
com.ibm.CSI.performMessageIntegritySupported
com.ibm.CSI.performMessageConfidentialityRequired
com.ibm.CSI.performMessageConfidentialitySupported
```

Protecting a plain password inside the client security property file: The C++ SSL security client provides a password utility (PasswordEncoder4cpp) after an

IBM WebSphere Application Server Enterprise C++ CORBA SDK or C++ CORBA client installation. You can use the password utility provided by WebSphere to encode the key file password in the client's security property file for further protection. Security service can decode the password at run time and use the decoded password when opening the key file. To run the utility, see passwordEncoder4cpp utility.

Results

If encoding completes without any errors, then the plain password in the original property file is replaced by an encoded password. If you encounter one of the following errors, take the recommended user action to correct the problem:

ERROR: invalid target file == C:\property\my.props
This indicates that the target file does not exist or the path is incorrect.

User Action
Check the name and location of the property file and retry.

ERROR: no password properties in specified list were found in target file
This indicates that the ″com.ibm.ssl.keyPassword″ password property does not exist.

User Action
Provide the password attribute in the property file and retry.

ERROR: cannot load properties from target file
This indicates that the client property file is corrupted and cannot be loaded.

User Action
Reconstruct the property file or use the backup copy.

ERROR: invalid password encoding exception
This indicates that the utility has a problem encoding the password.

User Action
The valid characters for a password are a-z, A-Z, and 0-9.

ERROR: cannot create backup file
This indicates that the utility has a problem creating a backup file.

User Action
Make sure that you are able to create a backup file in the same location as the property file.

passwordEncoder4cpp utility: The passwordEncoder4cpp password utility is used to encode the key file password in the client's security property file for further protection.

The C++ SSL security client provides the PasswordEncoder4cpp password utility after an IBM WebSphere Application Server Enterprise C++ CORBA SDK or C++ CORBA client installation. Security service can decode the password at run time and use the decoded password when opening the key file.

Syntax

- For Windows platforms:
  - Change to the `%WAS_HOME%\bin` directory.
  - Execute `PasswordEncoder4cpp.bat` *filename*

  where *filename* is the name of the client security property file.

- For Unix platforms:
  - Change to the `$WAS_HOME/bin` directory.
  - Execute `PasswordEncoder4cpp.sh` *filename*

  where filename is the name of the client security property file.

A backup file with a .bak extension is generated to save the original file.

Note: For information on encoding errors, see "Protecting a plain password inside the client security property file" on page 64.

Resolving CORBA run-time errors

If you encounter a problem while running a CORBA application within WebSphere, consult the following topics, which can help identify the cause of run-time problems:

- To set the directories and maximum sizes for message and trace logs and to turn on various tracing features, see "Specifying run-time properties for CORBA C++ clients and servers" on page 52.
- For information on reading a message log, see "Reading a message log" on page 67.
- For general information on problem determination, see "CORBA problem determination".

CORBA problem determination

When IBM WebSphere Application Server is run, it records information about its activities in its message log. The message log on each host captures events that show a history of the activities on that host. Some of the entries in the log are informational and others report system exceptions, such as returned CORBA exceptions.

If you encounter run-time errors, it is often useful to read the message log and try to diagnose the problem yourself. If you still need assistance from IBM to help you diagnose problems, you can provide the message log to your IBM service personnel.

There is one message log for each host machine. The file is called `activity.txt` by default, and it resides in the IBM WebSphere Application Server service subdirectory. You can set the `com.ibm.CORBA.logger.directoryName` run-time property to specify an alternate location. You can also specify the maximum size of the message log with the `com.ibm.CORBA.logger.maxLogFileSize` run-time property. The message log file is automatically created when the first log entry needs to be written.

Because the message log is an accumulation of information, it always contains extraneous data. Some message log entries report serious failures, but many of them only report on the execution of activities, expected exceptions, or warnings of potentially dangerous situations. For example, in most instances, lower level components write an entry in the message log when they decide to throw an exception, even when the caller of the lower level component is prepared to handle the exception and continue processing on a normal code path. Although all of these entries on activities, handled exceptions, and warnings can make it difficult to read the log, sometimes they provide useful data to help you determine the exact cause of the problem that you are diagnosing.

If you need to do low-level debugging of problems identified in the message log, you can turn on tracing for appropriate components and then study the detailed information generated. For more information about setting trace-related run-time properties, see “Specifying run-time properties for CORBA C++ clients and servers” on page 52 and “Run-time properties for CORBA clients and servers” on page 54.

Note: tracing must be used only in situations where you need to diagnose a problem or otherwise capture very detailed information about the operation of the ORB and the application. The use of the trace capabilities result in performance degradation, so they should not be used during normal operation.

If you turn on tracing for a component type, extra detailed information is recorded in one or more trace logs for the host. Multiple trace files can be generated if needed. The trace log files are stored in the IBM WebSphere Application Server's service subdirectory by default. Set the `com.ibm.CORBA.logger.directoryName` run-time property to specify an alternate location. Trace log files are automatically created and given unique names that are determined by using the process id, thread id, and current time stamp.

Reading a message log

The IBM WebSphere Application Server message logs and trace logs are written as simple text files. These files can be viewed with a text editor or any other text browser-type utility.

It is easier to locate the cause of a problem in smaller message logs. Therefore, consider reducing the size of the message log before attempting to read it. For more information about this, see “Message log: Hints and tips” on page 72. Also, for information about the contents of a message log entry, see “Message log entry: Description” on page 68.

When reading a message log, identify the group of entries that are related to the problem or error that you want to resolve. A group of entries forms a bracket, as follows:

```
The start of the bracket
       Initial failure, which is a single entry in the log.
Results of the initial failure
       A number of entries in the log.
The end of the bracket
       Last result of the failure, which is a single entry in the log.
```

In general, when you are reading the message log, start with its last entry and then work backwards reading the previous entry, then the one before that, and so on.

To find the bracket of entries for a problem that you are diagnosing, complete the following steps:

Steps for this task

1. Identify the end of the bracket. The first step in reading the message log is to identify the last entry that reported the problem that you want to diagnose (that is, the end of bracket entry). This is essential for identifying the cause of the problem. Start with the latest entry in the message log and search backwards for the entry that reports the problem. The last entries of the log do not always relate to the problem that you are trying to solve.

   When the entry related to the failure has been identified, you have found the end of the bracket. Remember the process id, thread id, process type, and

process alias associated with the end of bracket entry. This helps to associate other related message entries with this one.

2. Find relevant entries. Examine each entry with a matching process id, thread id, process type and process alias and focus on the message entries that have a time stamp that is relatively the same as the end bracket entry.
3. Find the initial failure. When you have found the first entry for the cause of the problem, you can take action to resolve it. Depending on the situation, you also might want to read one or two entries before the initial failure's entry. This is in case there is additional data to help you diagnose the problem.

Message log entry: Description: This section describes message log entries. The information is the same whether the messages are logged to a file or displayed on a console, whether the messages are the result of a tracing event or a severe error event. The messages are logged as plain text, and can be viewed with any text editor or browser utility.

There are several run-time properties that control message logging. See "Run-time properties for CORBA clients and servers" on page 54 for a complete list of these properties. In particular, the `com.ibm.CORBA.logger.fileDetail` and `com.ibm.CORBA.logger.consoleDetail` properties control the amount of detail provided in message log entries (that is, brief or detailed). Detailed messages contain all of the information known about the message, some of which would be useful only to IBM support personnel. Brief messages contain only the timestamp and the message text.

Example of detailed message log entry

The following is an example of a detailed message log entry:

```
ComponentId:  393316
ProcessId:    981
ThreadId:     857
FunctionName: e:/l0203/src/eborb/lib/or/callstrm/callsiop.cpp
ProbeId:      4180
SourceId:     @(#) 1.22 src/eborb/lib/or/misc/wasderr.cpp,
              eborb, ebroker,
              l0203.02 1/22/02 09:32:01 [1/22/02 17:40:58]
Manufacturer: IBM Corporation
Product:      WebSphere Application Server
Version:      5.0
ProcessType:  daemon
ProcessAlias: WASDAEMON
HostName:     gaston
TimeStamp:    2002-01-25 12:54:26.61339573
Severity:     1
Message Text: A SystemException occurred: INITIALIZE, minor code 0x49420036
(SOMDERROR_SOMDDAlreadyRunning) at e:/l0203/src/eborb/lib/or/callstrm/callsiop.cpp line 4180.
Cannot open the Location Service Daemon's listening port at address gaston:2809.
The Location Service Daemon may already be running.
```

Example of a brief message log entry

The following is an example of a brief message log entry:

```
TimeStamp:      2002-01-25 12:58:17.188243117
A SystemException occurred: INITIALIZE, minor code 0x49420036 (SOMDERROR_SOMDDAlreadyRunning)
at e:/l0203/src/eborb/lib/or/callstrm/callsiop.cpp line 4180.
Cannot open the Location Service Daemon's listening port at address gaston:2809.
The Location Service Daemon may already be running.
```

Location of the message log file

By default, messages are logged in the file `$WASORBTOP/service/activity.txt`, where WASORBTOP is an environment variable which specifies the top level

installation directory of the ORB. Note that alternate names for both the directory and filename can be specified by setting the com.ibm.CORBA.logger.directoryName and com.ibm.CORBA.logger.logFileName ORB run-time properties. For more information on this, see "Run-time properties for CORBA clients and servers" on page 54.

Note: Many different processes may be logging messages to the same message log file. Each message entry includes the process id and thread id associated with the program that logged the message.

Message log file rollover

It is possible that the message log file could grow to an undesirable size. This might be caused by a program logging a large number of messages or by many programs logging messages to the same file over a long period of time.

To help manage the size of this file, an automatic rollover capability is provided by the logger.

Note: This capability applies only to the message log file. It is not provided for trace log files.

Before a message is actually logged to the message log file, a check is made to determine if the current log file size is larger than the currently configured maximum size. The ORB run-time property com.ibm.CORBA.logger.maxLogFileSize is used to specify this maximum size, in kilobytes. The default value is 1024 KB, or 1 MB. If you set this property to 0 (zero), you effectively disable the automatic rollover capability.

If the message log file is larger than the maximum size, then:

- The message log file is renamed by appending .prev to its name. For example, $WASORBTOP/service/activity.txt is renamed to $WASORBTOP/service/activity.txt.prev.

  If this is not the first rollover, then a file named $WASORBTOP/service/activity.txt.prev already exists. This file is overwritten due the rename operation in the previous step.
- The new message to be logged is then written to a newly-created file named $WASORBTOP/service/activity.txt.

Communications trace log entries

The com.ibm.CORBA.orbCommunicationsTraceLevel run-time property controls the tracing of GIOP messages sent or received by the ORB. The ORB supports four levels of tracing: none, basic, intermediate and advanced. The following sections provide examples of the basic, intermediate, and advanced trace levels.

Example of basic communications trace log entry

The following is an example of a brief trace log entry with the com.ibm.CORBA.orbCommunicationsTraceLevel property set to **basic**:

```
TimeStamp:    2002-01-28 15:06:31.398705663
File/function e:/l0203/src/eborb/lib/or/trans/transip.cpp:1466
              has logged trace data:

0000  47 49 4F 50 01 02 01 00 - 7C 00 00 00 01 00 00 00  GIOP....|.......
0010  03 00 00 00 00 00 00 00 - 43 00 00 00 4A 4D 42 49  ........C...JMBI
```

```
0020  00 00 00 13 00 00 00 00 - 0A C3 63 C2 BD 40 1C 55  ........c.@.U
0030  E0 00 02 E2 09 35 5C A0 - 00 00 00 24 00 00 00 1F  ....5\ ...$....
0040  41 52 53 55 00 10 00 0F - 00 0C 00 04 65 42 6F 61  ARSU........eBoa
0050  00 04 01 54 FA 12 00 12 - 00 00 00 52 03 00 00 00  ...T.......R....
0060  0F 00 00 00 67 65 74 50 - 72 69 6E 74 65 72 4C 69  ....getPrinterLi
0070  73 74 00 00 01 00 00 00 - 1C 4D 42 49 08 00 00 00  st.......MBI....
0080  01 00 00 00 00 00 45 FF                           ......E.
```

The basic trace level provides a hexadecimal dump of the contents of the GIOP message.

Example of intermediate communications trace log entry

The following is an example of a brief trace log entry with the com.ibm.CORBA.orbCommunicationsTraceLevel property set to **intermediate**:

```
TimeStamp:    2002-01-28 15:05:00.960148555
File/function e:/l0203/src/eborb/lib/or/trans/transip.cpp:1466
              has logged trace data:

0000  47 49 4F 50 01 02 01 00 - 7C 00 00 00 01 00 00 00  GIOP....|.......
0010  03 00 00 00 00 00 00 00 - 43 00 00 00 4A 4D 42 49  ........C...JMBI
0020  00 00 00 13 00 00 00 00 - 22 8F 8B B9 BC E5 1C 55  ..............U
0030  E0 00 02 FA 09 35 5C A0 - 00 00 00 24 00 00 00 1F  .....5\ ...$....
0040  41 52 53 55 00 10 00 0F - 00 0C 00 04 65 42 6F 61  ARSU........eBoa
0050  00 04 01 54 FA 12 00 30 - 00 00 00 18 01 00 00 00  ...T...0........
0060  0F 00 00 00 67 65 74 50 - 72 69 6E 74 65 72 4C 69  ....getPrinterLi
0070  73 74 00 00 01 00 00 00 - 1C 4D 42 49 08 00 00 00  st.......MBI....
0080  01 00 00 00 00 00 45 FF                           ......E.

***** GIOP Message *****
GIOP Version:      1.2
Byte Order:        LittleEndian (Intel)
More Fragments:    No
Message Length:    124 (0x0000007C)
Message Type:      REQUEST
Request ID:        1 (0x00000001)
Response flags:    0x00000003 (reply msg required=Yes, twoway request=Yes)
Method name:       getPrinterList
```

The intermediate trace level adds formatting of the various headers within the GIOP message for easier readability.

Example of advanced communications trace log entry

The following is an example of a detailed trace log entry with the com.ibm.CORBA.orbCommunicationsTraceLevel property set to advanced:

```
ComponentId:  393316
ProcessId:    364
ThreadId:     748
FunctionName: e:/l0203/src/eborb/lib/or/trans/transip.cpp
ProbeId:      1466
SourceId:     @(#) 1.12 src/eborb/lib/or/trans/transip.cpp, eborb,
              ebroker, k0149.03 10/23/01 11:02:49 [1/4/02 09:13:55]
Manufacturer: IBM Corporation
Product:      WebSphere Application Server
Version:      5.0
ProcessType:  client
ProcessAlias: DefaultClient
HostName:     gaston
TimeStamp:    2002-01-28 15:09:43.357868559
Severity:     3
Message Text:
File/function e:/l0203/src/eborb/lib/or/trans/transip.cpp:1466
```

```
             has logged trace data:

0000  47 49 4F 50 01 02 01 00 - 7C 00 00 00 01 00 00 00  GIOP....|.......
0010  03 00 00 00 00 00 00 00 - 43 00 00 00 4A 4D 42 49  ........C...JMBI
0020  00 00 00 13 00 00 00 00 - 0C 4B E9 C5 BE 00 1C 55  .........K.....U
0030  E0 00 02 FF 09 35 5C A0 - 00 00 00 24 00 00 00 1F  .....5\ ...$....
0040  41 52 53 55 00 10 00 0F - 00 0C 00 04 65 42 6F 61  ARSU........eBoa
0050  00 04 01 54 FA 12 00 1E - 00 00 00 12 03 00 00 00  ...T............
0060  0F 00 00 00 67 65 74 50 - 72 69 6E 74 65 72 4C 69  ....getPrinterLi
0070  73 74 00 00 01 00 00 00 - 1C 4D 42 49 08 00 00 00  st.......MBI....
0080  01 00 00 00 00 00 45 FF                           ......E.

***** GIOP Message *****
GIOP Version:      1.2
Byte Order:        LittleEndian (Intel)
More Fragments:    No
Message Length:    124 (0x0000007C)
Message Type:      REQUEST
Request ID:        1 (0x00000001)
Response flags:    0x00000003 (reply msg required=Yes, twoway request=Yes)
Method name:       getPrinterList

Target Address begins at offset: 20 (0x00000014)

Service context list:
   [0]  id = 0x49424D1C [IOP::CPPOrbLevelContext],
        length = 8 (0x00000008), data offset = 127 (0x0000007F)
        Base ORB Major Version: 0x0000FF45
        Base ORB Minor Version: 0x00000000
        Extended ORB Version:   0x00000000

Parameters begin at offset: 136 (0x00000088)
```

The advanced trace level adds more detailed formatting of the contents of the GIOP message, such as formatting of object references, service contexts, tagged profiles and tagged components.

Location of trace log files

By default, trace messages are written to a file whose name is of the form $WASORBTOP/service/process_alias/trace_file_name, where:

- **WASORBTOP** is an environment variable that specifies the ORB's top level installation directory.
- **service** is the default directory name used by the message logger. This can be changed by setting the `com.ibm.CORBA.logger.directoryName` run-time property.
- **process_alias** is the program's process alias, or logical name. For clients, the default value for this is DefaultClient, and for a server it is the server's implementation definition name (for example, MyServer1, or AppServer25). The `com.ibm.CORBA.processAlias` run-time property can be used to explicitly set the process alias value.
- **trace_file_name** is a unique filename based on the time of process initialization and the process id. More specifically, the filename is composed of a 10-digit integer representing the time that the process was initialized, followed by a 6-digit integer representing the process id. A three character extension is then added to the end. This extension will be .txt if the `com.ibm.CORBA.logger.multipleTraceFiles` property is set to no, otherwise it will be an indication of which thread is associated with the trace file. The first thread within the process is identified by the extension .101, then the second is identified by the extension .102, etc.

Message log: Hints and tips: In most problem determination situations, you need to quickly pinpoint the message log entries related to the problem that you are investigating. One way to do this is to reduce the message log to a more manageable size by setting its size, or by creating smaller messages logs.

Setting the size of the message log

Before starting client or server processes on a host, set the `com.ibm.CORBA.logger.maxLogFileSize` run-time property to the desired number of kilobytes. For more information about specifying run-time properties, see "Specifying run-time properties for CORBA C++ clients and servers" on page 52.

Creating smaller message logs

Smaller message logs can speed upyour problem determination process. If the run-time error can be reproduced by rerunning your application, consider performing the following steps to create a set of small message logs:

1. Rename or delete your existing message log.

   This ensures that you start your problem diagnosis with an empty message log, which minimizes the amount of extraneous information that you need to consider when diagnosing problems.
2. Restart the servers associated with the application.
3. When the servers have started, rename the message log to another filename, such as `serverinit.log`.

   This allows you to keep the server initialization messages together in a small message log.
4. Run your test.
5. After the test, rename the message log to another filename, such as `testrun.log`.

   This message log contains messages that directly relate to the test run.

You now have a set of message logs that are small enough to be manageable and useful for comparison. For example, if `testrun.log` shows that a client could not find a factory, then serverinit.log might show you why that factory was not registered.

Managing the CORBA Interface Repository

The CORBA Interface Repository (IR) is the component of the Object Request Broker (ORB) that provides persistent storage of interface definitions. It provides access to a collection of object definitions that are specified in the Interface Definition Language (IDL). The IR is populated with data by programs that are emitted by the IDL compiler (through the use of the `-eir` option). These programs are produced by the IDL compiler according to the IDL files that describe the object or objects.

The IR is used mainly to support the Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI) capabilities of the ORB. However, it can be used in any situation where a program needs to dynamically retrieve information about object definitions.

Steps for this task

1. "Installing the CORBA Interface Repository server" on page 73
2. "Populating the CORBA Interface Repository" on page 74

3. "Accessing the CORBA Interface Repository" on page 74

Installing the CORBA Interface Repository server

Before you begin

The CORBA Interface Repository (IR) requires IBM DB2®. Be sure you have DB2 installed before proceeding. Support for the CORBA Interface Repository server (wasirsvr) in WebSphere is provided by the CORBA C++ Software Development Kit (SDK). While installing the CORBA C++ SDK, you have the option to install the Interface Repository, as described in the following steps.

Steps for this task

1. Install the base IBM WebSphere Application Server.
2. Install C++ SDK CORBA server support. Follow the steps for your platform.
3. Choose Interface Repository support when installing the CORBA C++ SDK. The IR database and tables are created automatically during installation.

Results

To verify the IR installation, run the **irdump** utility from the command line. The resulting output should show the default CORBA Object entry. For example:

```
=============
InterfaceDef:          ::Object
=============
```

For more information on the irdump utility, see "irdump utility".

irdump utility: The irdump utility lists the contents of the Interface Repository.

In default mode, irdump begins at the root of the Interface Repository (IR) database and outputs everything to standard output. Optionally, a module or interface name may be specified and only that entity outputs to standard out.

Running irdump when the Interface Repository is empty produces the following output:

```
=============
InterfaceDef:          ::Object
=============
RepositoryID:          IDL:Object:1.0
Defined in:            The Repository
Version:               1.0
** Num of Base Interfaces: 0
    Interface is empty.
```

If the Interface Repository resides on the same computer where irdump is being run, irdump will access the Interface Repository directly.

If the Interface Repository resides on another computer and the Interface Repository server (wasirsvr) is running there, the irdump utility can be directed to that server's Interface Repository.

The command line arguments are passed to ORB_init, so any properties that may be set on ORB_init can be passed when invoking irdump. For example if wasirsvr is running on "host1" and listening on port 727, irdump can be run by using the following command:

```
irdump.exe -ORBInitRef InterfaceRepository=corbaloc::host1:727/InterfaceRepository
```

Populating the CORBA Interface Repository

The Interface Repository (IR) is populated with data by programs that are emitted by the IDL compiler (through the use of the `-eir` option). These IR population programs are produced by the IDL compiler according to the IDL files that describe the object or objects.

Steps for this task

1. Emit the _IR.cpp for application idl. From the C++ SDK environment, given an application whose interfaces are defined in idl file `XXX.idl`, run the IDLC program to emit the IR C++ source file associated with this idl. Specify ″ir″ to make idlc invoke the IR emitter.

   For example:

   ```
   idlc -eir XXX.idl
   ```

   The generated source file is `XXX_IR.cpp`.
2. Build a program. Compile and link the generated `XXX_IR.cpp` source file into a program, for example: `XXX_IR`.
3. Run the program, `XXX_IR`, on the computer where the IR resides.

Results

To verify that the application interface information has been populated (loaded) into the IR, run the irdump utility from the command line. The resulting output shows the application's interface, operations, attributes, and so forth.

For more information on the irdump utility, see irdump utility article in the InfoCenter.

Accessing the CORBA Interface Repository

CORBA C++ clients and servers can access the Interface Repository (IR) through a well-defined API. A program can call the get_interface() method on any object to obtain its interface definition from the IR or get access to the IR by calling resolve_initial_references(″InterfaceRepository″).

To enable remote access, a C++ application server must be running on the computer where the IR resides. Any C++ application server automatically is able to serve the contents of the IR. If a C++ application server is not available, you can run the C++ IRServer program, wasirsvr.

If a C++ program is run on the same computer where the IR resides, the IR is accessed by the program directly. Otherwise, the IR is accessed remotely using typical CORBA client-server communication.

You can use either of the following APIs to access the IR, regardless of whether the program is running remotely or locally with respect to the IR:

get_interface() method
: The client can access the specific IR information for an application object by calling the get_interface() method on the application CORBA object.

resolve_initial_references(″InterfaceRepository″) method
: In general, you can obtain the root object of the IR by calling the resolve_initial_references(″InterfaceRepository″) method. However, if the IR is remote from the client and the resolve_initial_references() method is used, you must configure both the program using the IR and the server to use the same port. To do this, complete the following steps:

1. Configure the server to listen on a specific port. For example, to start the IR server, wasirsvr, tell it to listen to port 727 by entering the following command:

   ```
   wasirsvr -Dcom.ibm.CORBA.serverListenPort=727
   ```

2. Configure your program to look for the InterfaceRepository service on the server's host and port. Specify the **-ORBInitRef** option and a corbaloc URL when you start the program. These arguments are passed to ORB_init within the program. For example, if the server is running on "host1", you could run the client utility, irdump, using the following command:

   ```
   irdump.exe -ORBInitRef InterfaceRepository=corbaloc::host1:727/InterfaceRepository
   ```

   For more information on the wasirsvr command, see "wasirsvr command".

   For information on how to stop a server that is running, see "WSStopServer command" on page 76.

   For more information on the irdump utility, see "irdump utility" on page 73.

wasirsvr command: The wasirsvr command starts the CORBA Interface Repository server.

The Interface Repository server provides access to the Interface Repository for CORBA clients. When configured and run on a computer where the Interface Repository database (WASORBIR) resides, the Interface Repository server hosts the Interface Repository root object.

Syntax

```
wasirsvr or wasirsvr.exe
```

An alternate way to access the Interface Repository is to use a C++ client or server running on the same computer where the Interface Repository resides. Also, a CORBA client may use the CORBA get_interface method on a CORBA object if that object is hosted on a server with access to the Interface Repository.

The Interface Repository server enables CORBA clients and servers to access an Interface Repository by calling the ORB::resolve_initial_references method with the argument, "InterfaceRepository" as defined by CORBA.

The Interface Repository server must be configured to listen on a specific port. The client must be configured to look for the Interface Repository service on the server's host and port.

Start wasirsvr with a specific listening port by setting the com.ibm.CORBA.serverListenPort property. For example to start the server and tell it to listen on port 727, enter the following command:

```
wasirsvr -Dcom.ibm.CORBA.serverListenPort=727
```

On the client, call ORB_init with the -ORBInitRef argument, followed by "InterfaceRepository=" and a corbaloc URL that specifies the server's host and listening port.

For example, if wasirsvr is running on ″host1″, the client program, irdump, can be run by the command:

```
irdump.exe -ORBInitRef InterfaceRepository=corbaloc::host1:727/InterfaceRepository
```

Note: If the Interface Repository resides on the same computer as the client, the client's ORB accesses the Interface Repository directly without using the server.

Examples

```
1-H:/d0237.01/src: wasirsvr
CORB1162I:
Interface Repository server listening...
```

For further information on accessing the CORBA Interface Repository, see "Accessing the CORBA Interface Repository" on page 74.

For further information on the irdump utility, see "irdump utility" on page 73″.

WSStopServer command: The WSStopServer command line utility is used to stop a server that is running.

The WSStopServer command line utility contains a WSServerShutdown object. The WSServerShutdown object enables the server to terminate whenever desired. When the server is about to terminate, the WSStopServer command line utility (on Windows, WSStopServer.exe) tells the ORB to shut the server down. The WSStopServer executable receives the server alias. The executable uses the alias to send a message to the thread spun off by the WSServerShutdown object, requesting that the server terminate. The WSServerShutdown object forces the execute_request_loop() method to return control to the server, which then terminates.

Syntax

```
WSStopServer server_alias
```

where *server_alias* is the server name created by the regimpl utility.

CORBA programming model

The CORBA programming model describes the artifacts that you develop and implement to enable client applications to interact with server applications in a CORBA environment. In this context, the word client refers to any program that makes a remote method request on a servant object. The server, a server process, hosts a servant object (in CORBA terminology) through which the client accesses business functions. On a CORBA C++ server, the servant object implements the business functions. On an EJB server, the business logic is implemented by an enterprise bean.

The CORBA programming model, as a distributed-object programming model, is characterized as follows:

Objects
CORBA objects are defined with the OMG Interface Definition Language (IDL). IDL is compiled to generate client stubs and server skeletons, which map an object's services from the server environment to the client.

Communications protocol
The specification is the General Inter-ORB Protocol (GIOP). The Internet

Inter-ORB Protocol (IIOP) is one implementation of this specification. Together, these protocols specify the message formats and data representations used between CORBA clients and servers.

Object references
CORBA Interoperable Object References (IOR) provide a platform and vendor-independent object reference.

Naming service
The CORBA CosNaming service is located (bootstrapped) with resolve_initial_references(). CosNaming binds a CORBA object to a public name.

The following image shows the artifacts that you develop and implement for the CORBA programming model:

CORBA programming model

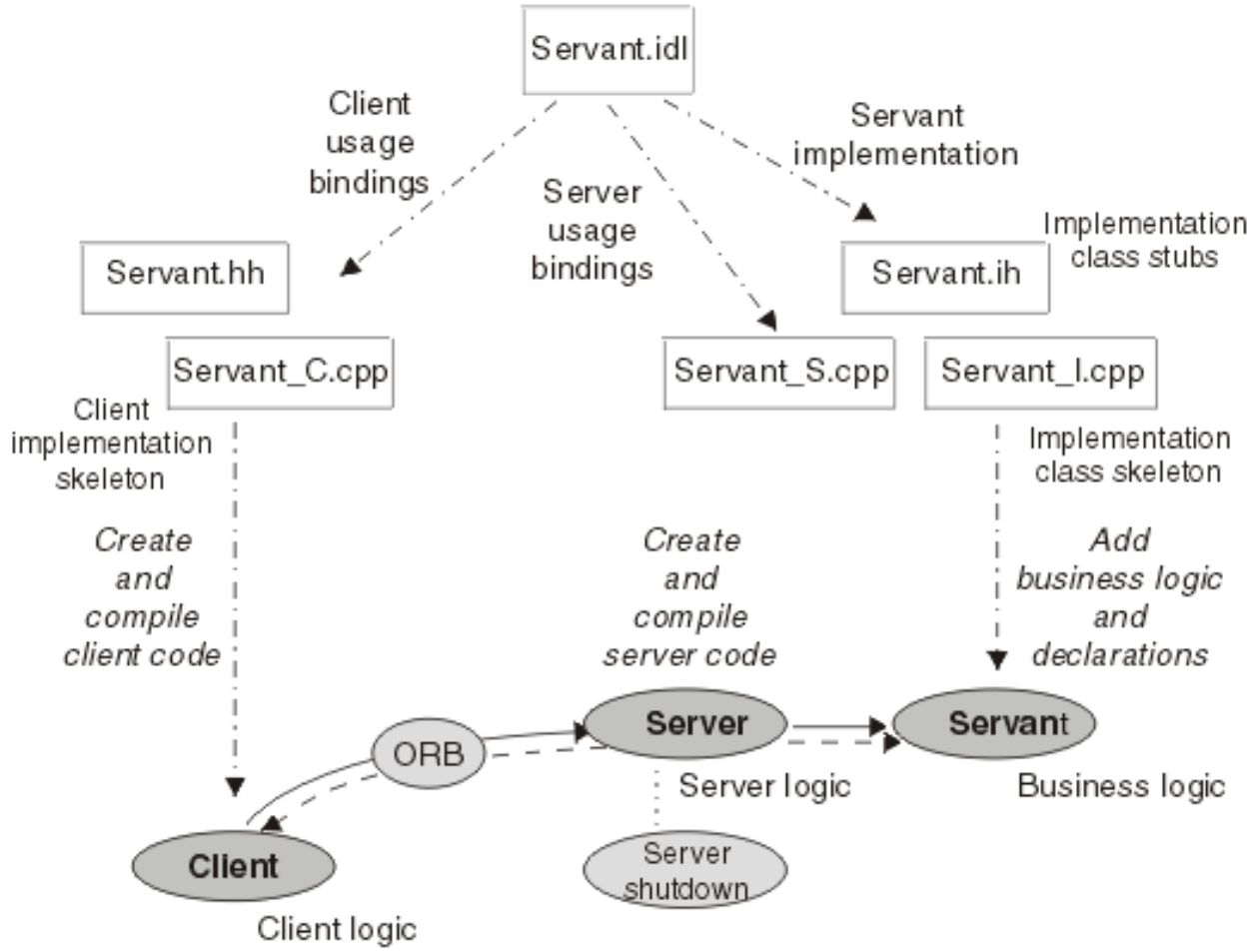


Figure 2. CORBA programming model

The CORBA programming model comprises the following two interrelated parts:

- The server programming model describes the interfaces and processes used to develop CORBA server objects that make up the business logic and business data inherent in a server application. Application programmers use the server programming model if they are developing CORBA server objects that perform

business functions used in the implementation of client objects. For more information about the server programming model, see the topic "CORBA server programming model" on page 82.

- The client programming model describes how client applications use the objects provided by server applications. Application programmers use the client programming model if they are developing CORBA clients that access servant implementations that are either CORBA server objects or enterprise beans. For more information about the client programming model, see the topic "CORBA C++ client programming model" on page 82.

In IBM WebSphere Application Server, the CORBA client, and server programming models are used as follows:

- The CORBA client programming model is used for WebSphere C++ clients to access a WebSphere EJB server.
- The CORBA client programming model also is used for WebSphere C++ clients to access a WebSphere C++ server or another CORBA ORB acting as a CORBA server.
- The J2EE server programming model is used for WebSphere EJB servers.
- The CORBA server programming model is used for WebSphere C++ servers and other CORBA servers.

CORBA concepts

A CORBA environment is based on client applications finding and using objects that provide a desired function. The objects typically represent something in the real world, for example, shopping carts, and are hosted by servers (usually EJB servers). The type of object is defined by its interface and the semantics defined for that interface. There can be many instances of an object (with the same interface and semantics) that represent different entities. CORBA provides the Interface Definition Language (IDL) to define object interfaces and Object Request Brokers (ORBs) to provide access to objects through a distributed environment. The binding of an object's interface to a specific implementation is handled in the server environment.

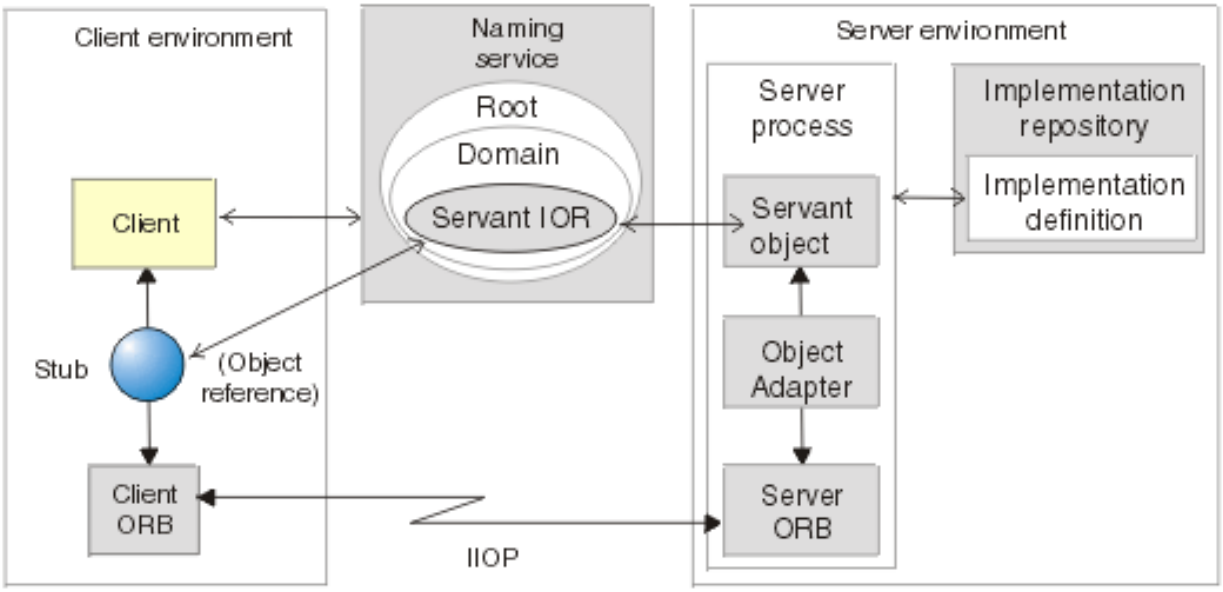


Figure 3. Conceptual view of a CORBA C++ environment

A CORBA environment comprises the following elements:

- Client programming languages
- Client proxy
- Client run-time environment
- CORBA-compliant Object Request Broker (ORB)
- Internet Inter-ORB protocol (IIOP)
- Implementation repository
- Initial references
- Interface definition language (IDL)
- Interoperable naming service (INS)
- Interoperable object reference (IOR)
- Naming service
- Object
- Object adapter
- Object reference
- Server
- Server implementation (servant) object
- WebSphere Application EJB Server

Client programming languages

WebSphere CORBA clients can be developed in C++. Other CORBA clients can be developed in C++, ActiveX, Java, or other languages supported by the CORBA client programming model.

Client proxy

To the client, the object on the server appears as if it resides in the client program. This is accomplished by using a client-side proxy (stub) object. The proxy object has the same interface as the server-side object it represents, but does not directly implement the object's methods. Instead, the proxy object translates a method

invocation into a format that is communicated from the client to the server using their respective ORB infrastructures. The server finds the target servant object, which executes the actual method implementation.

Client run-time environment

The client run-time environment enables CORBA client applications to access server implementation objects.

The client environment can be either of the following:

- An IBM WebSphere CORBA C++ client or server (written with the CORBA C++ SDK)
- A non-IBM, ORB-based CORBA client or server

CORBA-compliant Object Request Broker (ORB)

The CORBA-compliant ORB enables clients to communicate with the application server.

The ORB sends local client requests across a network by using the Internet Inter-ORB Protocol (IIOP). The IIOP is a TCP/IP-based communications protocol with CORBA-defined message exchanges. Separate ORBs reside at each end of the communication channel.

IIOP

The client and server ORBs communicate using the CORBA Internet Inter-ORB protocol (IIOP), which is a TCP/IP-based protocol with CORBA-defined message exchanges.

CORBA uses the General Inter-ORB Protocol (GIOP) to define the format of messages and uses IIOP to encode and decode GIOP messages.

Implementation repository

The implementation repository is a persistent data store of ImplementationDef objects that each represent a logical CORBA C++ server that has been registered. The implementation repository is used by the location service daemon.

Initial references

An initial reference is a well-known object reference associated with an identifier. CORBA provides mechanisms to configure, list, and get (or resolve) initial references. Obtaining an initial reference of the Naming Service also is called ″bootstrapping″.

Interface definition language (IDL)

The CORBA IDL provides clients and servers with a platform-independent and language-neutral mechanism to base their communications.

Using IDL, application developers can specify the public interface to a CORBA class or enterprise bean (as the servant class). For a CORBA server implementation, the application developer typically creates the IDL ″by hand″. For an enterprise bean, a tool is used to create the IDL from the interface or class file. The IDL definition of a servant is used to generate the client proxy (stub). An IDL compiler generates the code necessary to use an interface with a specific programming language.

Serializable objects used in an EJB's interface are expressed in IDL as CORBA valuetypes. Therefore every Java serializable object passed by a CORBA client as a parameter or return value for an enterprise bean must be reimplemented in the

language of the client. To simplify the development of CORBA clients of enterprise beans, minimize the range of Java serializable objects used in the EJB's interface.

Interoperable naming service (INS)
The Interoperable Naming Service is a CORBA-defined syntax for specification and use of object URLs.

An object URL can be used when calling string_to_object or working with initial references. Use of INS URLs can be used by a client as an alternative to explicit bootstrapping. However, in generating the object, the ORB accomplishes bootstrapping internally.

Interoperable object reference (IOR)
An IOR allows objects to be communicated across process boundaries. IORs provide platform-independent and vendor-independent object references. A client can convert an IOR into a string, externalize it by saving it to a file, and then terminate. When the client is activated again, the IOR can be read from the file and converted back into an object reference.

Naming service
The WebSphere naming service forms the lookup directory for a distributed system. It provides an interface for binding and resolving names to object references.

When an object is created, its object reference can be bound to a name in the naming service. Any application with access to the naming service can use the name to find the associated object reference. The naming service implements the CosNaming service, the standard naming service defined by the CORBA services specification.

A CORBA client can establish access to a root naming context for the naming service by using a URL as specified by Interoperable Name Service (INS) and calling string_to_object or by calling resolve_initial_reference(″NameService″) on the client ORB.

Object
From the client's point of view, a CORBA object is an entity with an object reference that provides the operations defined in its interface.

These operations are always available to the client from the time the object instance is created until the time it is released.

Object adapter
The object adapter acts as a mediator between the communications framework of the server-side ORB and the servant objects that reside on that server.

When the server-side ORB receives a request, the ORB passes the request to the object adapter. The object adapter identifies the target of the request and dispatches it to that servant object. The object adapter class provides methods that allow the server application to participate in the exporting and importing of object references and the selection of threads to which remote requests are dispatched.

Object reference
An object reference contains information that is used to identify a target object.

A client-side proxy object contains information to locate the target server and the target servant object within that server.

Server
The C++ application server provides the run-time environment in which a servant object can exist. It initializes the ORB and object adapter, creates a servant object, and registers it in an appropriate naming context in the naming service. The server invokes the object adapter's request processing, enabling it to dispatch requests to the servant object that the server hosts. If the server is shut down, it removes the servant object from the run-time environment and cleans up the resources used to support the servant object.

Server implementation (servant) object
The servant implementation object (also known as a servant object) is the executing CPU and memory resource that performs an object's operation. This object is visible to the server only.

WebSphere Application EJB Server
The WebSphere Application EJB Server supports the communications protocol, the ORB, object references, and RMI-IIOP. In addition, it implements the Java Naming and Directory Interface (JNDI) using a directory that also supports CORBA CosNaming bindings. This enables WebSphere enterprise beans to be visible to CORBA clients.

For more information, see the CORBA and OMG Web sites.

CORBA C++ client programming model
The CORBA client programming model describes how CORBA clients access enterprise beans or CORBA server objects. Application programmers use the CORBA client programming model to develop tier-1 (client) or tier-2 (server) CORBA applications. (A CORBA server can act as a client to another server.)

The information about the CORBA client programming model, provided in the following topics, is based on developing CORBA C++ clients:

- Initializing the C++ ORB
- Locating the root naming context (bootstrapping)
- Locating a servant object
- Using a servant object
- Locating the EJB home
- Using an enterprise bean
- Handling CORBA exceptions
- Coding tips for proper CORBA memory management
- CORBA client to WebSphere EJB server

Examples of client programming are provided in the WSLoggerClient sample, accessible from the Samples Gallery, which is installed with IBM WebSphere Application Server.

CORBA server programming model
This topic describes the CORBA server programming model, which describes the interfaces and processes used to develop CORBA server objects that make up the business logic and business data inherent in a server application. Application programmers use the server programming model if they are developing CORBA server implementation objects, known as servant objects, that perform business functions used in the implementation of client objects.

The concepts about the server programming model are derived from the following general procedure for developing a CORBA server. For task information about developing a CORBA server, see "Developing a CORBA C++ server" on page 20. Examples of server programming are given in the WSLoggerServer sample, accessible from the Samples Gallery, which is installed with IBM WebSphere Application Server.

1. Specify the business logic implementation interface for the servant (*servant*.idl).

   In an Interface Definition Language (IDL)file, you define the public interface to the methods provided by the business logic. This defines the information that a client must know to call and use a servant object. For more information about the IDL definition of an implementation, see "Interface Definition Language (IDL)" on page 148.

2. Compile the servant IDL (using **idlc**).

   Compiling the servant IDL file produces the usage binding files to implement and use the servant object within a particular programming language. For example, this creates an implementation template that provides a native, server language class template into which method behavior can be inserted. WebSphere supports CORBA servers implemented in C++.

3. Add declarations for class variables, constructors, and destructors to the servant class definition (*servant*.ih).

   The implementation class interface header (*servant*.ih) created by **idlc** contains a skeleton class definition, but lacks declarations for class variables, constructors, and destructors. You need to add the missing declarations.

4. Complete the servant implementation (*servant*_I.cpp).The implementation class (*servant*_I.cpp) created by **idlc** contains a skeleton implementation definition, which you need to complete by adding the business logic that the servant provides.

5. Create the server main code (server.cpp).

   Create the server code to define the methods that the server implements. In particular, create the main() function, which controls the server run time by performing the following tasks:

   a. Validating user input
   b. Initializing the server environment
   c. Accessing naming contexts
   d. Creating a servant object
   e. Binding the servant object to the appropriate naming context
   f. Creating a server shut down object
   g. Going into a wait loop
   h. Servicing requests

6. Build the server object and server code.

   Like any other programming model, you need to build the modules that the server host can use to run the server and the servant.

7. Store a logical definition for the server in the system implementation repository (using regimpl).

   Each server needs a unique logical definition in the implementation repository of the host on which the server runs. The logical definition defines the server alias that is used to control the server.

CORBA object services

CORBA object services interoperate by delivering context information, with messages, that establish service state and other parameters. Some older Object Request Brokers (ORBs) do not support the passing of this context or use proprietary context data that cannot interoperate with another server.

Conversely, because a service context is *not* part of the message normally seen at the programmer's level, solutions that involve a break in the normal flow of a message do not automatically propagate a service context. Such solutions include wrapper classes or messages manually propagated across co-existent ORBs. If context propagation is required under such circumstances, it must be explicitly or manually managed in the code. If available, request interceptors provide a useful way to propogate contexts.

Naming service

For CORBA applications, WebSphere supports the CORBA CosNaming service, which binds CORBA objects to a public name. Clients are ″bootstrapped″ according to the CORBA programming model. CORBA-compliant Interoperable Object References (IORs) must be obtained and server objects must be bound into the CORBA CosNaming service. (For CORBA client access to enterprise beans, the EJB home must be bound into the CORBA CosNaming service.)

For more information about the naming service, see "CORBA naming service".

Transaction service

WebSphere supports the CORBA object transaction service (OTS) as defined by the EJB specification. WebSphere follows the CORBA transaction service specification for propagating transaction contexts and forwards the transaction context to the server. For interoperation with other ORBs, incoming contexts are honored and outgoing transaction contexts are generated, as appropriate.

For more information about the transaction service, see "CORBA transaction service" on page 85.

Security service

Security Service has been implemented to support C++ client applications accessing protected enterprise beans. To access the protected enterprise beans inside a secure WebSphere domain, a C++ client needs to propagate its identity over a transport protocol (often a secure one) to a server for authentication and authorization check. Successful authentication and authorization allows the client to invoke methods on the protected beans.

For more information about the security service, see "CORBA security service" on page 86.

CORBA naming service

WebSphere supports the CORBA CosNaming service, which binds CORBA objects to a public name. Clients are ″bootstrapped″ according to the CORBA programming mode. CORBA-compliant Interoperable Object References (IORs) must be obtained and server objects must be bound into the CORBA CosNaming service. (For CORBA client access to enterprise beans, the EJB home is bound automatically into the CORBA Naming Service and, therefore, can be accessed through the CosNaming interfaces.)

The naming service provides a mapping between names and object references. When an object is created, it is assigned an object reference, which can be bound

with a ::CosNaming::Name name into the namespace managed by the naming service. Any client (or any other object) with access to the naming service can use the associated ::CosNaming::Name name to retrieve the object reference.

The namespace is hierarchical and similar in structure to a file system tree. The nodes of the namespace are CORBA::objects (either NamingContext objects or leaf objects). A NamingContext object, or naming context, can contain zero or more bindings of name-object reference pairs. Each object, bound by name into a naming context, can be a leaf object or a subordinate NamingContext in the tree. Subordinate NamingContexts similarly can contain bindings of other NamingContexts and leaf objects.

For example, a servant object called WSLoggerObject1 is bound into naming context called WSLoggerContext, which was created by a CORBA server for the servant objects that it hosts. The WSLoggerContext naming context is bound into the domain naming context called domain, which is bound into the root naming context for the naming service. This might be represented by the domain.WSLoggerContext.WSLoggerObject1 object reference and represented by the following hierarchy:

```
       /        (root)
       |
    domain
       |
WSLoggerContext
       |
WSLoggerObject1
```

This also can be represented by the name string "/domain/WSLoggerContext/WSLoggerObject1".

CORBA transaction service

WebSphere supports the object transaction service (OTS) and follows the CORBA transaction service specification for propagating transaction contexts. It forwards the transaction context from a client to the server. An Object Request Broker (ORB) uses incoming transaction contexts to either handle transactions transparently or ignore transaction contexts that it does not understand.

For transactional support, a CORBA client of an enterprise bean must rely on one of the following options:

- Container-managed transactions, where the container automatically starts and ends each new transaction
- Bean-managed transactions
- Client-initiated transactions

WebSphere CORBA C++ clients and servers provide a client-side transaction service only. They can act as a transactional client only to a server which supports a transaction service (for example, a WebSphere EJB server). Objects on a WebSphere CORBA C++ server are not recoverable.

In many cases the enterprise bean infrastructure within WebSphere automatically initiates transactions, even if the application code does not. If an enterprise bean calls a CORBA server from within a transaction, the following might happen:

- (Best) the CORBA server resources are coordinated with the transaction, or there are no resources to coordinate.
- The server does not recognize the transaction context, so it is ignored. The application writer must recognize this and code accordingly.

- The server crashes due to the presence of the WebSphere transaction context. The application writer must either design the enterprise bean to not run from within a transaction context, or use coexistence to ensure that the transaction context is not automatically propagated. To disable automatic creation of transaction contexts, deploy the WebSphere Enterprise JavaBean with a transaction attribute other than the default TX_REQUIRED. Use TX_NOT_SUPPORTED, TX_SUPPORTS, or an equivalent transaction attribute that does not force the creation of transaction context.

If an EJB client is to use a CORBA server in the scope of a transaction, consider the following transaction timeout issue. If the enterprise bean is executing a loop, if the server object takes an excessive amount of time to execute, and the enterprise bean executes within the scope of a single transaction, then built-in transaction timeouts can be exceeded. This causes unexpected failures that have nothing to do with CORBA architectural issues. This is natural behavior but not necessarily expected if you are not familiar with the issues. This can be avoided by creating a new transaction each time through the loop.

For clients to other ORBs, you can use the co-existent ORB solution to start other transactions.

CORBA security service

WebSphere provides a security service that supports CORBA C++ clients to access protected enterprise beans over SSL. To access the protected beans, the client is required to prove its identity (by authentication) and role (by authorization) to the secure EJB server. All request messages are also protected.

The security service uses the SSL transport protocol for both client authentication and message protection. Once the client is authenticated, the client's identity may be used for matching the role required by the server's authorization policy with respect to the protected beans. With identity assertion, the server also can assert a client's identity for authorization checking or identity propagation in downstream requests.

WebSphere CORBA C++ clients and servers provide a client-side security service only. They can act as a secure client only to a server that supports SSL and CSIv2 (for example, a WebSphere EJB server).

The following figure describes a typical C++ client security topology:

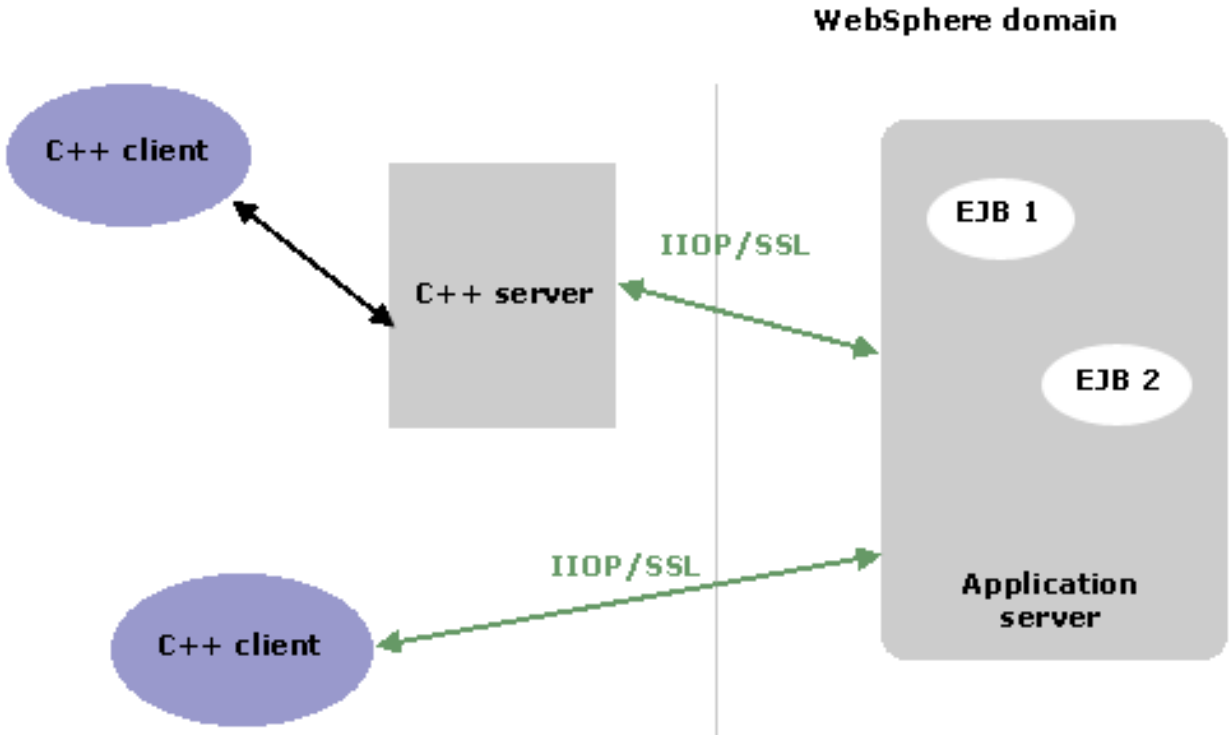


Figure 4. C++ client security - Topology diagram

SSL There are different levels of protection for a SSL connection. Client authentication is also optional. Before a client request is dispatched, the security service determines an effective security policy by coalescing both client and server configurations. The effective policy is then used to set the required level of protection that meets the SSL requirements of both client and server. Once the coalesced requirement is set, the ORB then attempts to establish the appropriate SSL connection.

Note: The client configuration is based on the client's security properties while the server configuration is read from the Interoperable Object Reference (IOR). The evaluation of effective security policy is executed at every method request.

Common Secure Interoperability Version 2 Security Protocol (CSIv2)
There are two authentication protocols implemented for the WebSphere EJB server: Secure Association Service (SAS) and Common Secure Interoperability Version 2 (CSIv2). Both protocols are based on the Interoperable Inter-ORB Protocol (IIOP). Because CSIv2 is the strategic protocol, the security service is implemented to support only CSIv2 at the transport layer.

Client Authentication with SSL

Client authentication with SSL is enabled by default. When enabled, the C++ clients must already be configured with a valid SSL certificate, and the certificate's public key must already be imported into the server's truststore file. Using SSL with client authentication is especially important since the server might assert the client's identity for further downstream requests. If the client authentication fails during the SSL handshake, the connection fails immediately and the request is rejected. If the client authentication succeeds and the connection is established, the client's identity is then available at the server side.

Identity Assertion

Extracted identity can be stored into an identity token for identity assertion purposes. Identity assertion is used to assert a caller identity that is different than the authenticated one after a trust is established. With identity assertion, the target can assert a client's identity for authorization check or identity propagation in downstream requests. Check the CSIv2 sections for further information.

To use the security service, configure properties in the C++ client security properties file, scclient.props.

CORBA communication protocols (GIOP/IIOP)

The CORBA architecture provides the General Inter-ORB Protocol (GIOP) to define message formats between objects in a distributed environment. The Internet Inter-ORB Protocol (IIOP) is an implementation of GIOP.

GIOP includes a Common Data Representation (CDR) that resolves differences between native hardware architectures within such an environment. Different hardware architectures can have variations in byte ordering and alignment for multi-byte data types within the address space. GIOP provides the means for resolving the differences across platforms. GIOP supports a number of simple data types, compound data types, object references, exceptions, and other features, depending upon the version of the specification (as shown in the following table). Nevertheless, early ORBs can encounter interoperability problems related to byte ordering.

If you suspect a GIOP-related interoperability problem, it is reasonably safe to adopt GIOP Version 1.0, as all of the major ORBs support GIOP at this level. WebSphere supports client ORBs that use GIOP Version 1.0, 1.1, or 1.2. The WebSphere ORB accepts fragmented GIOP messages, but it does not send fragmented messages.

GIOP feature	Data types	GIOP version			WebSphere ORB	
		1.0	1.1	1.2	Java	C++
Simple data types	octet, char, short, unsigned short, long, unsigned long long long, unsigned long long, float, double, boolean, string	Yes	Yes	Yes	Yes	Yes
	long double	Yes	Yes	Yes	-	-
	fixed	Yes	Yes	Yes	-	-
	wchar, wstring	-	Yes	Yes (Footnote 1: There are differences in the wchar and wstring encoding between GIOP 1.1 and 1.2, which creates interoperability challenges. For detailed information, review the portion of the CORBA specification relating to GIOP and CDR.)	1.1	1.1, 1.2
	enum	Yes	Yes	Yes	Yes	Yes
Compound data types	struct, union, array, sequence	Yes	Yes	Yes	Yes	Yes
	valuetype	-	-	Yes	Yes	Yes
	CORBA::Object	Yes	Yes	Yes	Yes	Yes
	any	Yes	Yes	Yes	Yes	Yes
	context	Yes	Yes	Yes	Yes	Yes
	exception	Yes	Yes	Yes	Yes	Yes
GIOP Message types	Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError	Yes	Yes	Yes	Yes	Yes
	Fragment	Yes	Yes	Yes	Yes	Yes
Bi-directional	-	-	-	Yes	-	-

Resolving unsupported CORBA data types

If a client Object Request Broker (ORB) does not support a data type required by a server object, such as an enterprise bean or C++ servant object, you can use a variety of techniques to resolve this, including the following:

- Removing a data type that is not needed from the IDL for a server object
- Using a wrapper to hide an unsupported data type
- Using the dynamic invocation interface to call the server

Removing a data type that is not needed from the IDL for a server object

If a client Object Request Broker (ORB) does not support a data type defined in the Interface Definition Language (IDL) file for a server object and the client does not need to use the associated feature, you can remove the data type from the IDL used to create the client. For example, you can use the following steps to enable the client to use a version of the IDL file to access the server object:

1. Generate the IDL file that represents the server object. For example, to generate IDL for an enterprise bean, run `rmic -idl` on the EJB's home and remote interfaces.
2. Make a copy of the IDL file to use with the client.
3. Edit the IDL for the client to remove all references to unsupported data types. This can involve removing exceptions, objects, attributes, and methods (but not individual parameters or return types).
4. Compile the IDL and link the client with the generated bindings.

Using a wrapper to hide an unsupported data type

You can use a wrapper to hide unsupported data types needed by a server object behind a thin intermediate server object. The wrapper can be a CORBA object or a session bean. If a wrapper is being used to resolve an ORB that does not support valuetypes, then implement the wrapper as a CORBA object to avoid the various extraneous valuetypes generated by the EJB-to-IDL compiler.

A wrapper provides an alternate interface, a supported interface, and delegates its implementation to the original server object. The CORBA client accesses the intermediate wrapper and the wrapper is deployed on a server that can directly access the target server object. The wrapper interface must be designed such that it provides access to the target object's interface without using valuetypes (for an EJB server) or other unsupported data types.

A wrapper may be the only way to get client access working for some vendor ORBs.

When using wrappers, consider the following points:

- Management: The wrappers must be installed and managed in a CORBA server.
- Lifecycle: For an enterprise bean, the EJB container manages the lifecycle automatically. If the wrapper is a CORBA object in the same server as the enterprise bean, the wrapper's lifecycle must be managed explicitly in your code.

  Note: For Session Beans, you can manage the wrapper's lifecycle by unexporting and destroying the wrapper when the bean is removed. Because not all beans, for example, entity beans, are eventually removed, the unexport and destruction of the wrapper might have to be explicitly exposed to the client's programming model.In this situation, it is better to put CORBA wrappers in a CORBA server and enterprise beans in a different EJB server.
- Data types: The wrappers must convert between enterprise bean types and IDL types unless the client uses only primitive types. For example, EJB object

references must be converted into IDL object references. Also, Java serializable objects must be converted into IDL equivalents.

The client marshalls data into an opaque octet stream and passes it to the IDL wrapper. The IDL wrapper unmarshalls the data, inflates Java objects by value if necessary, and passes the data onto the target server object. For a target enterprise bean, this can be done in a session bean, which is free to use RMI-IIOP and valuetypes while interacting with entity beans.

- Hand coding: Both the wrapper interface design and the wrapper object must be hand-coded.

Using the dynamic invocation interface to call the server

As a last resort, the CORBA Dynamic Invocation Interface (DII) enables a client to make a call to a server without using IDL. Instead, the client makes a call by constructing the method parameters dynamically and storing them as CORBA::Any data types. This mode of access can be useful in the following cases:

- Accessing remote objects, such as enterprise beans, when the representative IDL might contain unsupported data types with the following characteristics:
  - The data types that prevent the IDL from being compiled by the client Object Request Broker's (ORB's) IDL compiler.
  - The data types are not required for the methods or features that must be accessed.
- The client does not have prior knowledge of the IDL definitions.
- The repository ID of an EJB interface is redefined, for example, some older ORBs do not support the `rmi:` prefix.

CORBA valuetype considerations

The Java language to Interface Definition Language (IDL) specification maps Java serializables to CORBA valuetypes (pass-by-value objects). Therefore, every Java serializable object that is passed between a client and server (for example, by a CORBA client as a parameter or return value for an enterprise bean) must be reimplemented in the language of the client. (The implementation for the valuetype must be defined and provided in the language run time for both the client and the server.) Implementation of Java serializable objects as valuetypes in C++ or another language can be a significant development effort.

Valuetypes were introduced by the CORBA 2.3 specification and many other Object Request Brokers (ORBs) do not implement the specification or do not implement it fully.

To aid application development, IBM WebSphere Application Server provides a valuetype library that contains the C++ valuetype implementation for some commonly used Java classes in the java.lang, java.io, and java.util packages (for example, Integer, Float, Vector, Exception, OutputStream, and so on). For more information about the valuetype library provided with IBM WebSphere Application Server, see "CORBA valuetype library for C++" on page 92.

Java language to IDL specification

An enterprise bean is implemented in Java with no hint of the CORBA architecture in its programming model. The enterprise bean specification requires that the server implementation be restricted to using those Java language constructs defined as the RMI/IDL subset by the Java language to IDL specification.

By following the Java language to IDL specification, you can create CORBA clients implemented in any programming language for which there is a defined mapping and for which ORB supporting valuetypes are available.

When valuetypes are not supported

For languages other than Java, such as C++, the CORBA architecture is often the only viable option for accessing enterprise beans.

For session bean interfaces that only use primitive data types, you can use generated IDL files to access the enterprise beans even if the client ORB does not support valuetypes. However, the IDL generated from such an enterprise bean still can include valuetype declarations for exceptions or other entities.

If you decide that the features supported by valuetypes are not needed, consider using the strategies outlined in "Resolving unsupported CORBA data types" on page 89.

CORBA valuetype library for C++

The Java Language to IDL specification maps Java serializable objects to CORBA value types. Therefore every Java serializable object to be passed by a CORBA client as a parameter or return value for an enterprise bean must be reimplemented in the language of the client. Implementation of Java serializable objects as value types in C++ or another language can be a significant development effort.

To aid application development, IBM WebSphere Application Server provides a valuetype library that contains C++ valuetype implementations for some commonly used Java classes in the java.lang, java.io, java.util, javax.ejb, java.sql, and java.math packages, for example, Integer, Float, Vector, Exception, OutputStream, and so on. The valuetype library supports the WebSphere C++ Object Request Broker (ORB).

These classes represent an established hierarchy in the Java language and are implemented to preserve the inheritance relationship that exists in certain Java packages. These classes enable CORBA programmers to use the WebSphere C++ classes in the same way they use their Java counterparts. Constructors in the original Java classes do not need to be mapped to the IDL definitions and the C++ bindings. When mapped, constructors become create (or init) methods on the factory classes.

The IDL compiler always provides a pointer type definition for each type. For example, for a valuetype class T, the pointer type definition is `typedef T * T_ptr`. Unlike mapping for interfaces, the reference counting for valuetype must be implemented by the instance of the valuetypes. The IDL compiler also generates a _var class, which you can use instead of the _ptr. The _var class for a valuetype automates the reference counting, that is, it automatically manages the memory associated with the dynamically allocated object reference. When the T_var object is deleted, the object associated with T_ptr is released. When a T_var object is assigned a new value, the old object reference pointed to by T_ptr is released after the assignment takes place. A casting operator also is provided to enable you to assign a T_var to a type T_ptr.

Data type mappings: The WebSphere CORBA valuetype library for C++ provides mappings for the following primitive data types:

Java	IDL Type	C++ Type
boolean	boolean	CORBA::Boolean
byte	octet	CORBA::Octet
char	wchar	CORBA::Wchar
double	double	CORBA::Double
float	float	CORBA::Float
int	long	CORBA::Long
int	unsigned long	CORBA::ULong
long	long long	CORBA::LongLong
long	unsigned long long	CORBA::ULongLong
byte	char	CORBA::Char
short	short	CORBA::Short
short	unsigned short	CORBA::UShort
void	void	CORBA::void

Objects behave somewhat differently, as shown in the following examples (Java type-> IDL type-> C++ type):

- Array

  ```
  byte[]-> ::org::omg::boxedRMI::seq1_octet-> ::org::omg::boxedRMI::seq1_octet*
  ```

- String

  ```
  java.lang.String-> ::CORBA::WstringValue-> ::CORBA::WstringValue*
  ```

- Standard Java objects

  ```
  Java.util.Enumeration-> abstract valuetype Enumeration-> ::java::util:: Enumeration
  ```

The IDL definition for the Enumeration valuetype (as generated by the **rmic -idl** utility) is as follows:

```
module java {
  module util {
    abstract valuetype Enumeration;
  };
};
```

Run-time type information: Some of the classes in the WebSphere value type library contain methods that accept instances of a superclass. For such cases, the library use a C++ dynamic_cast to determine the type of the passed object. For example:

```
CORBA::Boolean equals(CORBA::ValueBase& arg0)
 {
    ...
  OBV_java::lang::Integer*  argInteger = dynamic_cast<OBV_java::lang::Integer*>(& arg0) ;
  ...
  }
```

This functionality allows you to perform type inquiries just as you would in Java using the ″instance of″ operator.

Note: For this code to work, a polymorphic hierarchy must exist, that is, at least one virtual function must be implemented in the class hierarchy.

Another possible approach is to use the ″typeid()″ operator of the type_info class. For example:

```
#include <typeinfo>
#include <iostream>
using namespace std;
class Test1 { __ };
class Test2 : Test1 {_..};
```

```
void main(void)
{
Test2* ptr = new Test2();
cout << typeid(*ptr).name() << endl; //yields the string "class Test2"
}
```

Depending on the compiler that is used, you must enable certain options in order for this functionality to work properly. For example, for MSVC++, the /GR option must be added to the compiler settings.

Application programming interface: The WebSphere valuetype library for C++ implements the methods listed in "CORBA valuetype library for C++: Methods implemented". Because the implemented classes are derived from generated classes, the member functions they contain differ slightly from those in the java classes. For example, in java, the `java.io.FilterOutputStream` class extends the abstract class `java.io.OutputStream`, so it must provide definitions for all abstract methods specified in the superclass. However, in the valuetype library hierarchy `java_io_FilterOutputStream_Impl` is derived from `java_io_OutputStream_Impl`; a concrete class that defines the methods of the generated class `::java::io::OutputStream`.

The types used in the signatures of these methods are derived from the OMG Specification. The semantics of each of the valuetype methods conforms exactly to those of their Java counterparts. For a more detailed function specification of each method, see Sun's Javadoc.

For each valuetype, there is a corresponding factory class. You must use the creation methods of a factory class (class name with _init or _factory suffix) to create instances of a valuetype (unlike the normal practice of using constructors to create objects in Java). Except in two cases, each creation method of the factory classes corresponds to a constructor in the Java counterparts of the valuetypes.

In addition to the valuetype classes, a utility class called VtlUtil is defined to provide several common methods to print debugging messages, handle exceptions, get registered factory objects, and make transformation between C++ strings and the ::CORBA::WstringValue objects.

Note: You can reuse a registered factory object with the com::ibm::ws::VtlUtil::getFactory() method instead of creating a new factory every time.

The vtlib.h header file contains the definitions of all the factory classes and the VtlUtil class. These classes are defined in the com::ibm::ws name space.

For an example of using a registered factory object, the com::ibm::ws::VtlUtil::getFactory() method, and the creation methods of a factory, see ″Example: C++ value type library″ in the InfoCenter.

CORBA valuetype library for C++: Methods implemented: The WebSphere valuetype library for C++ implements the following methods:

- java::io::IOException
- java::io::IOException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  java::io::IOException* create__ ()
virtual  java::io::IOException* create__CORBA_WStringValue (::CORBA::WStringValue* arg0)
```

- java::lang::Boolean

```
::CORBA::Boolean             booleanValue ();
::CORBA::Boolean             equals (const ::java::lang::Object& arg0);
::CORBA::Boolean             getBoolean (::CORBA::WStringValue* arg0);
::CORBA::Long                  hashCode ();
::CORBA::WStringValue* toString ();
::java::lang::Boolean*         valueOf (::CORBA::WStringValue* arg0);
```

- java::lang::Boolean_init

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Boolean* create__CORBA_WStringValue(::CORBA::WStringValue* arg0);
java::lang::Boolean* create__boolean(CORBA::Boolean arg0);
```

- java::lang::Byte

```
::CORBA::Octet       byteValue ();
::CORBA::Long        compareTo (const ::java::lang::Object& arg0);
::CORBA::Long        compareTo__java_lang_Byte (::java::lang::Byte* arg0);
::java::lang::Byte*     decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean   equals (const ::java::lang::Object& arg0);
::CORBA::Long        hashCode ();
::CORBA::Octet        parseByte__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::Octet        parseByte__CORBA_WStringValue__long (::CORBA::WStringValue*
                                                           arg0, ::CORBA::Long arg1);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__octet (::CORBA::Octet arg0);
::java::lang::Byte*    valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Byte*    valueOf__CORBA_WStringValue__long (::CORBA::WStringValue* arg0,
::CORBA::Long arg1);
```

- java::lang::Byte_init

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Byte* create__ ();
java::lang::Byte* create__octet(::CORBA::Octet arg0);
java::lang::Byte* create__CORBA_WStringValue(::CORBA::WStringValue* arg0);
```

- java::lang::Character

```
::CORBA::WChar       charValue ();
::CORBA::Long          compareTo (const ::java::lang::Object& arg0);
::CORBA::Long          compareTo__java_lang_Character (::java::lang::Character* arg0);
::CORBA::Long          digit (::CORBA::WChar arg0, ::CORBA::Long arg1);
::CORBA::Boolean     equals (const ::java::lang::Object& arg0);
::CORBA::WChar       forDigit (::CORBA::Long arg0, ::CORBA::Long arg1);
::CORBA::Long          getNumericValue (::CORBA::WChar arg0);
::CORBA::Long          getType (::CORBA::WChar arg0);
::CORBA::Long          hashCode ();
::CORBA::Boolean     isDefined (::CORBA::WChar arg0);
::CORBA::Boolean     isDigit (::CORBA::WChar arg0);
::CORBA::Boolean     isISOControl (::CORBA::WChar arg0);
::CORBA::Boolean     isIdentifierIgnorable (::CORBA::WChar arg0);
::CORBA::Boolean     isJavaIdentifierPart (::CORBA::WChar arg0);
::CORBA::Boolean     isJavaIdentifierStart (::CORBA::WChar arg0);
::CORBA::Boolean      isJavaLetter (::CORBA::WChar arg0);
::CORBA::Boolean      isJavaLetterOrDigit (::CORBA::WChar arg0);
::CORBA::Boolean      isLetter (::CORBA::WChar arg0);
::CORBA::Boolean      isLetterOrDigit (::CORBA::WChar arg0);
::CORBA::Boolean      isLowerCase (::CORBA::WChar arg0);
::CORBA::Boolean      isSpace (::CORBA::WChar arg0);
::CORBA::Boolean      isSpaceChar (::CORBA::WChar arg0);
::CORBA::Boolean      isTitleCase (::CORBA::WChar arg0);
::CORBA::Boolean      isUnicodeIdentifierPart (::CORBA::WChar arg0);
::CORBA::Boolean       isUnicodeIdentifierStart (::CORBA::WChar arg0);
::CORBA::Boolean       isUpperCase (::CORBA::WChar arg0);
::CORBA::Boolean       isWhitespace (::CORBA::WChar arg0);
::CORBA::WChar         toLowerCase (::CORBA::WChar arg0);
::CORBA::WStringValue* toString ();
::CORBA::WChar         toTitleCase (::CORBA::WChar arg0);
::CORBA::WChar         toUpperCase (::CORBA::WChar arg0);
```

- java::lang::Character_init

```
virtual CORBA::ValueBase *create_for_unmarshal();
virtual  java::lang::Character* create (::CORBA::WChar arg0);
```

- java::lang::Double

```
::CORBA::Long           compareTo (const ::java::lang::Object& arg0);
::CORBA::Long           compareTo__java_lang_Double (::java::lang::Double* arg0);
::CORBA::LongLong    doubleToLongBits (::CORBA::Double arg0);
::CORBA::Double        doubleValue ();
::CORBA::Boolean      equals (const ::java::lang::Object& arg0);
```

```
::CORBA::Long           hashCode ();
::CORBA::Boolean       infinite ();
::CORBA::Boolean       isInfinite (::CORBA::Double arg0);
::CORBA::Boolean       naN ();
::CORBA::Boolean       isNaN (::CORBA::Double arg0);
::CORBA::Double         longBitsToDouble (::CORBA::LongLong arg0);
::CORBA::Double         parseDouble (::CORBA::WStringValue* arg0);
::CORBA::WStringValue*   toString__ ();
::CORBA::WStringValue*   toString__double (::CORBA::Double arg0);
::java::lang::Double*           valueOf (::CORBA::WStringValue* arg0);
```

- java::lang::Double_init

```
virtual CORBA::ValueBase *java_lang_Double_factory::create_for_unmarshal();
java::lang::Double *java_lang_Double_factory::create__double (::CORBA::Double arg0);
java::lang::Double *java_lang_Double_factory::create__CORBA_WStringValue
                                                    (::CORBA::WStringValue* arg0);
```

- java::lang::Exception
- java::lang::Exception_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  java::lang::Exception* create__ ()
virtual  java::lang::Exception* create__CORBA_WStringValue (::CORBA::WStringValue* arg0)
```

- java::lang::Float

```
::CORBA::Long          compareTo (const ::java::lang::Object& arg0);
::CORBA::Long          compareTo__java_lang_Float (::java::lang::Float* arg0);
::CORBA::Boolean     equals (const ::java::lang::Object& arg0);
::CORBA::Long          floatToIntBits (::CORBA::Float arg0);
::CORBA::Float          floatValue ();

::CORBA::Long          hashCode ();
::CORBA::Float          intBitsToFloat (::CORBA::Long arg0);
::CORBA::Boolean     infinite ();
::CORBA::Boolean     isInfinite (::CORBA::Float arg0);
::CORBA::Boolean     naN ();
::CORBA::Boolean     isNaN (::CORBA::Float arg0);
::CORBA::Float           parseFloat (::CORBA::WStringValue* arg0);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__float (::CORBA::Float arg0);
::java::lang::Float*      valueOf (::CORBA::WStringValue* arg0);
```

- java::lang::Float_init

```
virtual CORBA::ValueBase *java_lang_Float_factory::create_for_unmarshal();
java::lang::Float *java_lang_Float_factory::create__double (::CORBA::Double arg0);
java::lang::Float *java_lang_Float_factory::create__float (::CORBA::Float arg0);
java::lang::Float *java_lang_Float_factory::create__CORBA_WStringValue (::CORBA::WStringValue* arg0);
```

- java::lang::Integer

```
::CORBA::Long          compareTo (const ::java::lang::Object& arg0);
::CORBA::Long          compareTo__java_lang_Integer (::java::lang::Integer* arg0);
::java::lang::Integer*   decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean     equals (const ::java::lang::Object& arg0);
::java::lang::Integer*   getInteger__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Integer*   getInteger__CORBA_WStringValue__long (::CORBA::WStringValue*
                                                  arg0, ::CORBA::Long arg1);
::java::lang::Integer*   getInteger__CORBA_WStringValue__java_lang_Integer
                            (::CORBA::WStringValue* arg0, ::java::lang::Integer* arg1);
::CORBA::Long          hashCode ();
::CORBA::Long          intValue ();
::CORBA::Long          parseInt__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::Long          parseInt__CORBA_WStringValue__long (::CORBA::WStringValue* arg0,
                                                                ::CORBA::Long arg1);
::CORBA::WStringValue* toBinaryString (::CORBA::Long arg0);
::CORBA::WStringValue* toHexString (::CORBA::Long arg0);
::CORBA::WStringValue* toOctalString (::CORBA::Long arg0);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__long (::CORBA::Long arg0);
::CORBA::WStringValue* toString__long__long (::CORBA::Long arg0, ::CORBA::Long arg1);
::java::lang::Integer* valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Integer* valueOf__CORBA_WStringValue__long (::CORBA::WStringValue* arg0,
                                                                ::CORBA::Long arg1);
```

- java::lang::Integer_init

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Integer* create__long (::CORBA::Long arg0);
java::lang::Integer* create__CORBA_WStringValue (::CORBA::WStringValue* arg0);
```

- java::lang::Long

```
::CORBA::Long         ompareTo (const ::java::lang::Object& arg0);
::CORBA::Long         compareTo__java_lang_Long (::java::lang::Long* arg0);
::java::lang::Long*     decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean    equals (const ::java::lang::Object& arg0);
::java::lang::Long*     getLong__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Long*     getLong__CORBA_WStringValue__long_long (::CORBA::WStringValue*
                                      arg0, ::CORBA::LongLong arg1);
::java::lang::Long*     getLong__CORBA_WStringValue__java_lang_Long
                              (::CORBA::WStringValue* arg0, ::java::lang::Long* arg1);
::CORBA::Long             hashCode ();
::CORBA::LongLong     longValue ();
::CORBA::LongLong      parseLong__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::LongLong      parseLong__CORBA_WStringValue__long (::CORBA::WStringValue*
                                             arg0, ::CORBA::Long arg1);
::CORBA::WStringValue* toBinaryString (::CORBA::LongLong arg0);
::CORBA::WStringValue* toHexString (::CORBA::LongLong arg0);
::CORBA::WStringValue* toOctalString (::CORBA::LongLong arg0);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__long_long (::CORBA::LongLong arg0);
::CORBA::WStringValue* toString__long_long__long (::CORBA::LongLong arg0,
                                                          ::CORBA::Long arg1);
::java::lang::Long*    valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Long*    valueOf__CORBA_WStringValue__long (::CORBA::WStringValue* arg0,
                                                          ::CORBA::Long arg1);
```

- java::lang::Long_init

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Long* create__long_long(::CORBA::LongLong arg0);
java::lang::Long* create__CORBA_WStringValue(::CORBA::WStringValue* arg0);
```

- java::lang::Number

```
virtual ::CORBA::Long     intValue();
virtual ::CORBA::LongLong longValue();
virtual ::CORBA::Float    floatValue();
virtual ::CORBA::Double   doubleValue();
virtual ::CORBA::Octet    byteValue();
virtual ::CORBA::Short    shortValue();
```

- java::lang::Short

```
::CORBA::Long          compareTo (const ::java::lang::Object& arg0);
::CORBA::Long          compareTo__java_lang_Short (::java::lang::Short* arg0);
::java::lang::Short*     decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean    equals (const ::java::lang::Object& arg0);
::CORBA::Long          hashCode ();
::CORBA::Short         parseShort__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::Short         parseShort__CORBA_WStringValue__long (::CORBA::WStringValue*
                                                    arg0, ::CORBA::Long arg1);
::CORBA::Short         shortValue ();
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__short (::CORBA::Short arg0);
::java::lang::Short*   valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Short*   valueOf__CORBA_WStringValue__long (::CORBA::WStringValue* arg0,
                                                          ::CORBA::Long arg1);
```

- java::lang::Short_init

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Short* create__CORBA_WStringValue(::CORBA::WStringValue* arg0);
java::lang::Short* create__short(::CORBA::Short arg0);
```

- java::lang::Throwable

```
::CORBA::WStringValue* localizedMessage ()
::CORBA::WStringValue* message ()
::CORBA::Void setMessage (const ::CORBA::WStringValue& arg0);
::CORBA::WStringValue* getMessage () const;
::CORBA::WStringValue* toString ()
```

- java::lang::Throwable_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  java::lang::Throwable* create__ ()
virtual  java::lang::Throwable* create__CORBA_WStringValue (::CORBA::WStringValue* arg0)
```

- java::math::BigDecimal

```
::CORBA::Long  java::math::BigDecimal::signum ()
::CORBA::Long  java::math::BigDecimal::scale ()
::java::math::BigInteger*  java::math::BigDecimal::unscaledValue ()
```

```
::java::math::BigDecimal* java::math::BigDecimal_factory::create__java_math_BigInteger
  (::java::math::BigInteger* arg0)
::java::math::BigDecimal* java::math::BigDecimal_factory::create__java_math_BigInteger__long
  (::java::math::BigInteger* arg0, ::CORBA::Long scale)
```

Note: The fourth and fifth line of the previous example wrapped due to the width of the page.

- java::math::BigInteger

```
::CORBA::Long  java::math::BigInteger::signum ()
::java::math::BigInteger* java::math::BigInteger_factory::
create__org_omg_boxedRMI_seq1_octet (::org::omg::boxedRMI::seq1_octet* magnitude)
```

Note: The second line of the previous example wrapped due to the width of the page.

- java::sql::Date

```
::java::sql::Date* java::sql::Date_factory::create__long_long (::CORBA::LongLong time)
```

- java::sql::Time

```
::java::sql::Time* java::sql::Time_factory::create__long_long (::CORBA::LongLong time)
```

- java::sql::Timestamp

```
::CORBA::Long  java::sql::Timestamp::nanos ()
::CORBA::Void  java::sql::Timestamp::nanos (::CORBA::Long nanos)
::java::sql::Timestamp* java::sql::Timestamp_factory::create__long_long (::CORBA::LongLong time)
```

- java::util::Date

```
::java::lang::Object*  java::util::Date::clone ()
::CORBA::LongLong  java::util::Date::time ()
::CORBA::Void  java::util::Date::time (::CORBA::LongLong time)
::java::util::Date* java::util::Date_factory::create__ ()
```

- java::util::Vector

```
::CORBA::Long getCapacity() const;
::CORBA::Void setCapacity(::CORBA::Long cap);
java_util_Vector_Impl& operator=(const java_util_Vector_Impl& aVector);
const std::vector<java::lang::Object>& getVectorInstance() const;
const std::vector<java::lang::Object>::iterator& getObjectIterator() const;
std::vector<java::lang::Object>::iterator& resetObjectIterator();
::CORBA::Void setCapacityIncrement(::CORBA::Long incrementValue);
::CORBA::Long getCapacityIncrement();
::CORBA::Void addElement (const ::java::lang::Object& arg0);
::CORBA::Long capacity ();
::java::lang::Object* clone ();
java::util::Vector* cloneVector ();
::CORBA::Void copyInto (::org::omg::boxedRMI::java::lang::seq1_Object* arg0);
::java::lang::Object* elementAt (::CORBA::Long arg0);
::java::lang::Object* getElements ();
::CORBA::Void ensureCapacity (::CORBA::Long arg0);
::java::lang::Object* firstElement ();
::CORBA::Long indexOf__java_lang_Object__long (const ::java::lang::Object& arg0,
::CORBA::Long arg1);
::CORBA::Void insertElementAt (const ::java::lang::Object& arg0, ::CORBA::Long arg1);
::CORBA::Boolean isEmpty();
::java::lang::Object* lastElement ();
::CORBA::Long lastIndexOf__java_lang_Object__long (const ::java::lang::Object& arg0,
::CORBA::Long arg1);
::CORBA::Void removeAllElements ();
::CORBA::Boolean removeElement (const ::java::lang::Object& arg0);
::CORBA::Void removeElementAt (::CORBA::Long arg0);
::CORBA::Void setElementAt (const ::java::lang::Object& arg0, ::CORBA::Long arg1);
::CORBA::Void setSize (::CORBA::Long arg0);
::CORBA::Long size();
::CORBA::Void trimToSize ();
```

- java::util::Vector_init

```
virtual ::CORBA::ValueBase *create_for_unmarshal()
virtual  ::java::util::Vector* create__ ()
::java::util::Vector* create__long (::CORBA::Long arg0)
::java::util::Vector* create__long__long (::CORBA::Long arg0, ::CORBA::Long arg1)
```

- javax::ejb::CreateException
- javax::ejb::CreateException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  javax::ejb::CreateException* create__ ()
virtual  javax::ejb::CreateException* create__CORBA_WStringValue (::CORBA::WStringValue* arg0)
```

- javax::ejb::RemoveException
- javax::ejb::RemoveException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  javax::ejb::RemoveException* create__ ()
virtual  javax::ejb::RemoveException* create__CORBA_WStringValue (::CORBA::WStringValue*arg0)
```

- javax::ejb::FinderException
- javax::ejb::FinderException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  javax::ejb::RemoveException* create__ ()
virtual  javax::ejb::RemoveException* create__CORBA_WStringValue (::CORBA::WStringValue*arg0)
```

- javax::ejb::ObjectNotFoundException
- javax::ejb::ObjectNotFoundException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::ejb::RemoveException* create__ ()
virtual javax::ejb::RemoveException* create__CORBA_WStringValue(::CORBA::WStringValue*arg0)
```

- javax::ejb::DuplicateKeyException
- javax::ejb::DuplicateKeyException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::ejb::RemoveException* create__ () virtual javax::ejb::
RemoveException* create__CORBA_WStringValue(::CORBA::WStringValue*arg0)
```

Note: The second line of the previous example wrapped due to the width of the page.

- javax::ejb::EJBMetaData

```
::javax::ejb::EJBHome_ptr  getEJBHome ()
::javax::rmi::CORBA::ClassDesc* getHomeInterfaceClass ()
::javax::rmi::CORBA::ClassDesc* getRemoteInterfaceClass ()
::javax::rmi::CORBA::ClassDesc* getPrimaryKeyClass ()
void setEJBHome (::javax::ejb::EJBHome_ptr arg0);
void setHomeInterfaceClass (::javax::rmi::CORBA::ClassDesc* arg0);
void setRemoteInterfaceClass (::javax::rmi::CORBA::ClassDesc* arg0);
void setPrimaryKeyClass (::javax::rmi::CORBA::ClassDesc* arg0)
::CORBA::Boolean isSession ()
```

- javax::rmi::CORBA::ClassDesc
- com::ibm::ws::java_io_PrintStream_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  ::java::io::PrintStream* create__ ()
virtual  ::java::io::PrintStream* create__java_io_OutputStream ( ::java::io::OutputStream *arg0)
virtual  ::java::io::PrintWriter* create__java_io_Writer (::java::io::Writer *arg0)

  /**
*  Create a new print stream over a file output stream.
*
*  @param   The name of the file output stream to which values and objects will be
*                   printed.
*  @return   the pointer to the created PrintStream object.
    */
virtual  ::java::io::PrintStream* create__CORBA_WStringValue (::CORBA::WStringValue* arg0)
```

- com::ibm::ws::java_io_FilterOutputStream_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  ::java::io::FilterOutputStream* create__ ()
```

- com::ibm::ws::java_io_PrintWriter_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual  ::java::io::PrintWriter* create__ ()
virtual  ::java::io::PrintWriter* create__java_io_Writer ( ::java::io::Writer *arg0)
virtual  ::java::io::PrintWriter* create__java_io_OutputStream
( ::java::io::OutputStream *arg0)

   /**
*  Create a new print writer over a file output stream.
```

```
*
*  @param   The name of the file output stream to which values and objects will be
*                   printed.
*  @return   the pointer to the created PrintStream object.
    */
virtual  ::java::io::PrintWriter* create__CORBA_WStringValue (::CORBA::WStringValue* arg0)
```

Note: The fourth line of the previous example wrapped due to the width of the page.

- com::ibm::ws::javax_rmi_CORBA_ClassDesc_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::rmi::CORBA::ClassDesc *create__()
```

- com::ibm::ws::javax_ejb_EJBMetaData_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual ::javax::ejb::EJBMetaData *create__ ()
```

- com::ibm::ws::VtlUtil

```
   static const char*                        exceptionLogFileName;
   static const char*                        debugLogFileName;

  /**
*  debugOn set to 1 is the debugging mode.
*  debugOn set to 0 is the non-debugging mode.
*/
   static const int                             debugOn;   // = 0;

  /**
*  debugInfoToStdOut set to 1, the debugging messages will be printed to stdout.
*  debugInfoToStdOut set to 0, the debugging messages will not be printed to stdout.
*/
   static const ::CORBA::Boolean  debugInfoToStdOut;

  /**
*   debugInfoToFile set to 1, the debugging messages will be printed to the file defined by
*   debugLogFileName.
*   debugInfoToFile set to 0, the debugging messages will not be printed to a file.
*/
   static const ::CORBA::Boolean  debugInfoToFile;  // = false;

   /**
*  Print the debugging message string msg to the designated media when debugOn is true.
*
*  @param  msg the <code>char *</code> to be printed.
 * @return   void
    */
   static void  debug(char *msg);

  /**
* Concatenate strings msg1 and msg2. Print the result string to the designated media
* if debugOn is true.
*
*  @param  msg1 the <code>char *</code> to be printed.
*  @param  msg2 the <code>char *</code> to be printed.
 * @return   void
    */
    static void  debug(char *msg1, char *msg2);


  /**
 *   Print the debugging message string str and the attributes of the exception e to
 *   the designated media if debugOn is true.
    *
*  @param  msg the <code>char *</code> to be printed.
*  @param  e the <code>java::lang:: Throwable* </code> to be printed.
*  @return   void
    */
   static void  debug(char msg, java::lang::Throwable* e);


  /**
*   Print the attributes of the exception e to stderr and the designated log file defined by the
*   exceptionLogFileName.
    *
*  @param  e the <code> java::lang::Throwable*</code> to be printed.
*  @return   void
    */
    static void  handleException(java::lang::Throwable* e);

  /**
*   Print the attributes of the exception e and the message string msg to stderr and
```

```
*   the designated log file *  defined by the exceptionLogFileName.

*  @param  e the <code> java::lang::Throwable*</code> to be printed.
*  @param  msg the <code>char *</code> to be printed.
*  @return   void
    */
    static void  handleException(java::lang::Throwable* e, char *msg);

   /**
*   Transform the string str to a WStringValue object and return its pointer.
    *
*  @param  str the <code>char *</code> to be transformed.
*  @return   pointer to the transformed WStringValue object
    */

   static ::CORBA::WStringValue* toWStringValue(const char *str);

  /**
*   Concatenate strings str1 and str2, and transform the result string to a
WStringValue object and return
*   its pointer.
    *
*  @param  str1 the <code>char *</code> to be transformed.
*  @param  str2 the <code>char *</code> to be transformed.
*  @return   pointer to the transformed WStringValue object
    */
    static ::CORBA::WStringValue* toWStringValue(const char *str1, const char *str2);


 /**
*   Concatenate strings str1, str2,and str3, and transform the result string to a
*   WStringValue object and return its pointer.
    *
*  @param  str1 the <code>char *</code> to be transformed.
*  @param  str2 the <code>char *</code> to be transformed.
*  @param  str3 the <code>char *</code> to be transformed.
*  @return   pointer to the transformed WStringValue object
    */
    static ::CORBA::WStringValue* toWStringValue(const char *str1, const char *str2, const char *str3);

   /**
*  Transform the WStringValue object wsv to a string and return the pointer to the string.
    *
*  @param  wsv the pointer to <code>::CORBA::WstringValue </code> to be transformed.
*  @return   pointer to the transformed string.
    */
   static char*  WStringValueToString(::CORBA::WStringValue *wsv);


   /**
  /**
*  Returns the registered factory object for the valuetype that has the designated repository id.
*  If the factory object is not found, a NULL pointer will be returned.
*
*  @param   the repository id of the factory to be retrieved as defined in the Vtlib.idl file.
*  @return   the pointer to the registered factory.
    */
   static ::CORBA::ValueFactoryBase*  getFactory(const char * rid);


  /**
*   Each of the following methods returns the registered factory object for the named valuetype.
*  If the factory object is not found, a NULL pointer will be returned.
*
*  @return   the pointer to the registered factory.
    */
   static java::lang::Boolean_init*  getBooleanFactory();
   static java::lang::Byte_init*  getByteFactory();
   static java::lang::Character_init*  getCharacterFactory();
   static java::lang::Double_init*   getDoubleFactory();
   static java::lang::Float_init*  getFloatFactory();
   static java::lang::Integer_init*          getIntegerFactory();
   static java::lang::Long_init*               getLongFactory();
   static java::lang::Short_init*              getShortFactory();
   static java::lang::Throwable_init*          getThrowableFactory();
   static java::lang::Exception_init*         getExceptionFactory();
   static java::io::IOException_init*          getIOExceptionFactory();
   static javax::ejb::CreateException_init * getCreateExceptionFactory();
   static javax::ejb::RemoveException_init * getRemoveExceptionFactory();
   static java::util::Vector_init*  getVectorFactory();
   static com::ibm::ws::javax_rmi_CORBA_ClassDesc_factory* getClassDescFactory();
   static com::ibm::ws::java_io_PrintStream_factory*   getPrintStreamFactory();
   static com::ibm::ws::java_io_FilterOutputStream_factory*  getFilterOutputStreamFactory();
   static com::ibm::ws::java_io_PrintWriter_factory*   getPrintWriterFactory();
   static com::ibm::ws::javax_ejb_EJBMetaData_factory*  getEJBMetaDataFactory();
```

Example: C++ value type library: The following examples are provided to illustrate use of the valuetype library methods in a distributed environment.

Example: A client program that uses a remote object to call methods of ::java::util::Vector

```
//First obtain the stringified ior of an EJB deployed on an AE server

using namespace com::ibm::ws;

CORBA::Object_var vector_obj;
//init the orb
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "DSOM");

//Get the stringified ior from a file, then use it to obtain a valid object reference
    ifstream in;
    int fileIndex = 0;
    char* iorfile = "VectorSession.ior"; // the stringified ior of a session bean that
                                         // uses instances of the java serializable
                                         // java.util.Vector
    in.open(iorfile);
    if (in.fail()) {
       std::cerr << "Cannot open file " << iorfile << std::endl;
       return 1;
    }
    char iorstr[2048];
    //read ior from file
    in >> iorstr;

    in.close();

    std::cout << "using ior:'" << iorstr << "'" << std::endl;

    //get stringified ior
    vector_obj = orb->string_to_object(iorstr);
    if (CORBA::is_nil(vector_obj))
    {
        std::cerr << "string_to_object failed"<< std::endl;
        return 1;
    }


// Now, call the createVector method of the stub class ejbPackage::VectorSession to access
// an EJB method that returns an instance of the java serializable, java.util.Vector. The
// stub method then returns a pointer to a java::util::Vector.

::CORBA::Short initialElement = 999;
java::util::Vector *vPrt;

    try
    {
       vPrt = vector_obj ->createVector(initialElement);
       if (vPrt == 0)
       {
          VtlUtil::debug("In testVector: vector_obj ->createVector(arg) returned a null pointer\n");

       }
    } catch (...)
       {
          VtlUtil::debug("In testVector: vector_obj ->createVector(arg) has thrown an exception\n");
       }

//Next use the remote object to a method of  ::java::util::Vector.

/************************
 * Create and populate a
 * java::lang::Object
 ************************/
short inValue = 999;
::CORBA::Long incrementValue = 1;

java::lang::Object obj;
obj <<= inValue;   //rvalue is a ptr

/*************************************************************
 * Call the addElement method using the pointer obtaned remotely
 * via the createVector method. Add "numberToAdd" elements.
 *  Verify that the correct size is returned
 *************************************************************/
::CORBA::Long numberToAdd = 5;
    for (int i = 0; i < numberToAdd; i ++) {
```

```
       try {
          vPrt->addElement (obj);
          } catch (...)
        {
VtlUtil::debug("In testVector: In catch after pwPtr->addElement()\n");
        }
    }
```

Example: A client program that uses a remote object to call methods of ::java::lang::Boolean

```
using namespace com::ibm::ws;

const char *factoryName = "java::lang::Boolean_init";

// Use the utility method, com::ibm::ws::VtlUtil::getBooleanFactory to get a pointer to the registered
// java:lang::Boolean_init factory object.

java::lang::Boolean_init* fact = VtlUtil::getBooleanFactory();

if(fact == 0)
{
  VtlUtil::debug("VtlUtil::getFactory returned a null value ");
}
else
{
  VtlUtil::debug("VtlUtil::getFactory returned a valid value ");
}

//call create__boolean  to create a pointer to a java::lang::Boolean that contains true
java::lang::Boolean* booleanPtr = fact->create__boolean(1);
if(booleanPtr == 0)
{
 VtlUtil::debug("booleanPtr == 0");
 return failed;
}
else
{
 VtlUtil::debug("create__boolean returned a valid value: test succeeded");
}

CORBA::Object_var boolean _obj;
 ::CORBA::Boolean trueBooleanValue = boolean_obj->callBooleanValue(booleanPtr);
 int tempTrueBooleanValue = trueBooleanValue;
 if (tempTrueBooleanValue == 1)
 {
   VtlUtil::debug("tempTrueBooleanValue == 1");
}
```

Creating your own C++ valuetypes

To aid application development, IBM WebSphere Application Server provides a valuetype library that contains C++ valuetype implementations for some commonly used Java classes in the java.lang, java.io, java.util, and javax.ejb packages. For example, Integer, Float, Vector, Exception, and so on. However, you might want to create your own C++ valuetypes.

The following steps describe how to create a C++ valuetype from an existing sample class called vttest.Book

Steps for this task

1. If you have private variables in your class that are accessed using EJB getter and setter methods, rename the methods to not use ″get″ and ″set″ in their names. For example, rename getProperty() and setProperty() to readProperty() and writeProperty(). This change is necessary to work around a problem with rmic.
2. Use the following command to generate the IDL file of your Java class:

   ```
   rmic -idl vttest.Book
   ```
3. Use the following command to generate the Book.hh and Book_C.cpp files:

   ```
   idlc -mcpponly -mnohhguards -mdllname=vtlib_name -shh:uc  -Iinclude-path vttest/Book.idl
   ```

This outputs the client-side definition and client bindings files, Book.hh and Book_C.cpp. These two files are generated and must not be edited.

4. Use the following command to generate the Book.ih and Book_I.cpp files:

```
idlc -mcpponly -mnohhguards -mdllname=vtlib_name -eih:ic  -Iinclude-path vttest/Book.idl
```

This outputs the skeleton implementation files, Book.ih and Book_I.cpp.

5. In the generated Book.ih file, add an inheritance from OBV_<package>::<class> to the <module>_<class>_Impl definition.

For example:

```
class vttest_Book_Impl : virtual public ::vttest::Book, virtual
public ::CORBA:: DefaultValueRefCountBase
```

becomes:

```
class vttest_Book_Impl : virtual public ::vttest::Book, virtual
public OBV_vttest::Book, virtual public
::CORBA::DefaultValueRefCountBase
```

You need this additional inheritance to use the default implementation class, OBV_<module>::<class>. See step 7 for further details.

Note: The generated file <class>_C.cpp contains the syntax required for the class OBV_<module>::<class>.

6. Add a class factory definition in the form <module>::<class>_factory to the <class>.ih file.

For example:

```
class <module_<class>_factory : public ::<module>::<class>_init
{
  public:

     virtual ;
        ::CORBA::ValueBase*     create_for_unmarshal();
        ::<module>::<class>*    create();
        ::CORBA::ValueBase*     asValueBase(void* v);
};
```

For Book class, this becomes:

```
class vttest_Book_factory : public ::vttest::Book_init
{
   public:

     virtual:
        ::CORBA::ValueBase*      create_for_unmarshal();
        ::vttest::Book*          create();
        ::CORBA::ValueBase*      asValueBase(void* v);
};
```

A template can be found in the factory classes defined in the file Vtlib_i_vb.cxx in your samples directory. The following steps use java_lang_Boolean_factory as a template.

Note: This template includes implementation code, which is omitted here. The template also lacks the asValueBase definition, which is added here.

```
class java_lang_Boolean_factory : public java::lang::Boolean_init
{
public:
virtual ::CORBA::ValueBase*
  create_for_unmarshal();
virtual java::lang::Boolean*
```

```
  create__CORBA_WStringValue(::CORBA::WStringValue* arg0);
virtual java::lang::Boolean*
  create__boolean(::CORBA::Boolean arg0);
virtual ::CORBA::ValueBase*
  asValueBase(void* v);
};
```

The <class>_init class defined in the generated <class>.hh file shows the create constructors that must be included in the <class>_factory for your class. (The create consructors above are specific to the Boolean valuetype class.) The class Book_init in Book.hh contains only one create constructor:

```
virtual  Book* create ()=0;
```

Therefore, only that create constructor is added to the class definition for vttest_Book_factory:

```
virtual ::vttest::Book* create();
```

7. Add implementations for the factory class methods to <class>_I.cpp. Sample implementations can be copied from the file vtlib_i_vb.cxx in the samples directory, such as the following for Boolean_factory:

```
virtual CORBA::ValueBase *create_for_unmarshal()
{
 return new java_lang_Boolean_Impl();
}
 virtual  java::lang::Boolean* create__boolean (CORBA::Boolean arg0)
{
 java_lang_Boolean_Impl *ptr = new java_lang_Boolean_Impl();
 ptr->value(arg0);
 return ptr;
}
```

The create_for_unmarshal() method creates a new <class>_Impl object and returns it. The create methods create new <class>_Impl classes and initialize them with appropriate values. (Your create methods must match those in your class.) Since the methods in the implementation file are not virtual, the virtual keyword must be deleted. In their completed form, the implementations for the Book class look like this:

```
::CORBA::ValueBase* vttest_Book_factory::create_for_unmarshal()
{
 return new ::vttest_Book_Impl();
}
 ::vttest::Book* vttest_Book_factory::create()
{
 return new ::vttest_Book_Impl();
}
```

The instance variables of a Java class are mapped into C++ counterparts in the OBV_* namespace's default implementation. (In a previous step, you added an inheritance of OBV_<module>::<class>). Use these OBV_ getters and setters in your implementation class.

8. An implementation of the factory method asValueBase() also is required. The method is a cast from (void *) to (::CORBA::ValueBase *). It takes the following form:

```
::CORBA::ValueBase* <module>_<class>_factory::asValueBase(void* v)
{
 return (::CORBA::ValueBase*)( (::<module>::<class>*) v );
}
```

You must change the class name to match your own class. In the case of the Book class:

```
::CORBA::ValueBase* vttest_Book_factory::asValueBase(void* v)
{
 return (::CORBA::ValueBase*)((::vttest::Book*)v);
}
```

9. Register your factory with the ORB by adding additional class, <module>_<class>_factoryInit. For the Book class, the following class and object instantiation do this:

   class vttest_Book_factoryInit { public : vttest_Book_factoryInit() { ::CORBA::ORB_ptr _vtlibOrb; int _argc = 0; ::CORBA::ValueFactoryBase_var retfact; ::CORBA::ValueFactoryBase_var fact; // Get access to the ORB. _vtlibOrb = ::CORBA::ORB_init(_argc, NULL, ″DSOM″); // Create a Book factory. fact = new ::vttest_Book_factory(); // Register the factory. retfact = _vtlibOrb->register_value_factory((char *)::vttest::Book::Book_RID, fact.in()); } }; // Static instantiation of the class. static vttest_Book_factoryInit __vttest_Book_factoryInit;

10. Add code that creates a Book objectd. You can put this code in the client source. First, insert an include of .hh. For Book, add #include <Book.hh>. Next, add a function that creates a Book object.

```
::<module>::<class>* create<class>()
{
 static <module>_<class>_init *factory = NULL;
 if ( factory == NULL )
 factory = (<module>::<class>_init*) ::com::ibm::ws::VtlUtil::getFactory
(::<module>::<class>::<class>_RID);
 ::<module>::<class>* myPtr = factory->create();
 return myPtr;
}
```

Note: The fifth and sixth lines of the previous example are one continuous line. However, the example had to wrap to fit within the width of the page.

For the Book class, this is:

```
::vttest::Book* createBook()
{
 static ::vttest::Book_init *factory = NULL;
 factory = (::vttest::Book_init*) ::com::ibm::ws::VtlUtil::getFactory(::vttest::Book::Book_RID);
::vttest::Book* bookPtr = factory->create();
 return bookPtr;
}
```

What to do next

You have now completed creation of your own C++ valuetype. Instances of this class can be created and used locally by your client. Because this class is a valuetype, it also can be serialized and sent to a server, where it can be used and returned to your client.

CORBA internationalization considerations

When you code CORBA applications for international use, consider the issues discussed in the following topics:

- "Initialization of client programs"
- "Character set restriction" on page 107
- "Codeset conversions" on page 107
- "Passing object references between multiple platforms" on page 107
- "OMG char data type in IDL files" on page 107

Initialization of client programs

All C++ clients should have their locale information set correctly. To do this, add the following so that it is called prior to calling CORBA::ORB_init():

```
setlocale(LC_ALL, "" );
```

Character set restriction

When developing CORBA applications, use only the Portable Character Set (PCS) in your IDL string type parameters. The PCS consists of the following characters:

```
0 1 2 3 4 5 6 7 8 9
 : ; < = > ? @ [ \ ] ^ _ ` ' ~ { | } ! " # $ % & ( ) * + , - . / <space>
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Codeset conversions

One of the fundamental features of the Object Request Broker (ORB) is to provide a way for clients and servers to communicate. Client and server processes can exist on various platforms and can be configured to use different locale information.

The ORB is responsible for performing the necessary codeset conversions to allow clients and servers on different platforms to interoperate. This section describes the ORB codeset conversion capabilities for both character and wide character data.

Passing object references between multiple platforms

When passing an object reference that is stored in a file from one platform to another, any string-modified values must be passed with the appropriate code page conversion.

For example, when you transfer files between Microsoft Windows NT and IBM OS/390®, you must use an ASCII-aware mechanism (such as FTP in ASCII mode or an ASCII NFS mount). Do not use FTP in binary mode.

OMG char data type in IDL files

Successful communications between clients and servers can involve codeset conversions. But CORBA limits the size of a `char` data item to one octet during transportation. So if any `char` data item is expanded to more than one octet in length during code set conversion, a CORBA::DATA_CONVERSION exception is thrown.

Use the `char` data type for a parameter or return result when the parameter or return result can contain data from the Portable Character Set only. For more information, see "Character set restriction". Otherwise, use a string data type or char array.

CORBA programming reference

CORBA 2.3 specifies standard forms by which client code can manipulate data whose types are described using an Interface Definition List (IDL). Reference material on programming these standard forms is grouped under the following topics:

- "CORBA types and business objects"
- "Commonly used CORBA interfaces" on page 109
- "CORBA C++ bindings" on page 110
- "Storage management and _var types" on page 129
- "CORBA exceptions" on page 138

CORBA types and business objects

CORBA basic types

Most of the CORBA types map directly onto C++ types and can be used transparently to C++. The following basic C++ types map directly into CORBA types:

- Atomic data types:
  - Boolean
  - Char
  - Double
  - Float
  - Long
  - Octet (hexadecimal)
  - Short
  - ULong (unsigned long)
  - UShort (unsigned short)
- Enum (enumerations)
- LongLong (long long)
- Struct
- ULongLong (unsigned long long)
- WChar (wide character)

All of these types are scoped to the CORBA class and must be declared accordingly. Their use in C++ is transparent and straightforward. For example:

```
CORBA::Short aShortvariable;
...
aShortVariable = 12;
...
```

CORBA types that return object references

Other CORBA types are not as straightforward to use because they return object references to the caller. The following CORBA types return object references to the caller:

- Any
- Array
- Sequence
- String
- Union
- WString (wide string)

It is the responsibility of the caller to manage the object references and their associated memory. There are two facilities provided by CORBA to do this:

A_var This is the facility most frequently used by client code because it is a smart pointer that automatically releases its object reference when it is deallocated or assigned a new object reference. This is the safest and most straightforward approach to managing these types.

A_ptr This is a pointer type that provides the most basic object reference, which has similar semantics to a standard C++ pointer.

Note: Avoid declaring C++ Static variables as _var. The _var holds a reference to an object. During the end of the process, this object might reference another object

that was removed before end processing completes for this static type. As a result, the _var might reference an inappropriate address or null pointer and thereby cause an undesirable ending.

Commonly used CORBA interfaces

The following are the most commonly used CORBA interfaces:

- CORBA class interfaces
- CORBA::object interfaces
- CORBA::ORB interfaces

CORBA class interfaces

The CORBA interface provides the following commonly used class operations. These are used like a C++ class reference (for example `CORBA::is_nil(somePointer);`).

is_nil This operation returns a boolean that indicates if the input object reference is nil. This is useful for many operations involving object references, including those operations that do not throw exceptions when they fail - for example `CORBA::Object::_narrow()`.

release
This operation releases resources associated with an object or pseudo-object reference.This operation may or may not perform a C++ delete operation. A reference count is used by this operation and `CORBA::Object::_duplicate()`. When the reference count reaches zero then the appropriate delete operations are performed. Care must be taken when using the release and _duplicate operations to ensure that objects are not leaked or inadvertently deleted. Alternatively use the _var technique described for **string_dup** below.

string_dup
This operation copies a string. A common example of its use is when returning a string from an operation. Strings and wide strings, unlike the other basic CORBA types, have associated allocated memory. So care must be taken when using these variables. The resulting string should subsequently be freed by using the CORBA::string_free operation, or by assigning the string to a _var variable which will free the string appropriately.

CORBA::object interfaces

The CORBA interface provides the following object interfaces:

_duplicate
This operation duplicates an object reference. This is particularly useful when passing references to objects to resolve memory ownership issues. For every _duplicate that is performed on an object an equal number of release() must also be performed for proper memory management. An alternative to the _duplicate() and release() logic is to use _var support as described for **string_dup** in CORBA class interfaces.

_is_a This operation is used to determine whether an object reference supports a given IDL interface. If the object supports the interface the _narrow operation can be successfully performed.

_is_equivalent
This operation is used to determine whether two object references refer to the same object.

_narrow
This operation is used to narrow a more generic interface to a more specific interface. This operation will return an empty pointer without throwing an exception if the interface cannot be narrowed to the requested type. Care must be taken to check the returned value before using it.

_nil This operation returns a nil CORBA::Object. This object could be used for comparison operations.

_non_existent
This operation determines whether an object reference refers to a valid object. This will result in verification of the object reference only, no other operations are performed on the requested object.

CORBA::ORB interfaces

The CORBA interface provides the following ORB interfaces:

object_to_string
This operation converts an object reference to an external form that can be stored for later use or exchanged between processes. The string_to_object operation can be used to reconstruct the object reference.

string_to_object
This operation converts a stringified object reference to a reconstructed object reference. The object_to_string operation must have been used to create the input stringified data.

Note: Although object_to_string is the way to save object references for future usage, the returned data should only be used with string_to_object to reconstruct that object reference. Do not use the string for comparing equivalence of object references. The object_to_string operation may return different values at different times because various Object Services may be adding information to this IOR.

CORBA C++ bindings

C++ bindings are generated, based on CORBA 2.1 standard forms, to enable client C++ code to manipulate data whose types are described using IDL. C++ bindings that support the standard forms are called *compliant* and client code that uses (only) these forms is called *conformant*.

For more information about C++ bindings, see the following topics:

- "CORBA C++ bindings for constants"
- "CORBA C++ bindings for data types" on page 111
- "CORBA C++ bindings for interfaces" on page 126
- "CORBA C++ binding restrictions" on page 128
- "Name scoping and modules in the C++ bindings" on page 129

CORBA C++ bindings for constants

Constants can be defined within the Interface Definition Language (IDL) in either of the following ways:

- Within a module or interface.
- Globally, outside any module or interface.

If you declare an IDL constant within a module or interface, the constant is mapped as a static data item local to the C++ class for that module or interface. If you declare an IDL constant globally, the constant is mapped as a static data item global to that client application.

For example, consider the following IDL:

```
module M
{
    const string name = "testing";
};
```

After compiling the client bindings, a C++ client application can refer to the constant using the expression `M::name`.

If the same constant is declared globally, outside any module or interface, then (after compiling the client bindings) a C++ client application can refer to the constant using the expression `name`.

CORBA C++ bindings for data types

C++ bindings can be created for the following CORBA data types:

- "C++ bindings for CORBA Any type"
- "C++ bindings for CORBA Array types" on page 115
- "C++ bindings for CORBA Atomic data types" on page 118
- "C++ bindings for CORBA Enumerations" on page 118
- "C++ bindings for CORBA Sequence types" on page 118
- "C++ bindings for CORBA Strings" on page 122
- "C++ bindings for CORBA Struct types" on page 123
- "C++ bindings for CORBA Union types" on page 124
- "C++ bindings for CORBA WStrings" on page 125

C++ bindings for CORBA Any type: The purpose of the IDL ″any″ type is to encapsulate data of some arbitrary IDL type. The C++ bindings provide a C++ class named CORBA::Any that provides this functionality. A CORBA::Any class encapsulates a void* pointer and a CORBA::TypeCode object that describes the thing being pointed to by the void*.

The Any type can be used with many of the CORBA types. It is useful when different types can be used that are unknown to the receiver of the data or also used as a common storage mechanism for passing a variety of types. The Any type can be used with many of the CORBA types. However, it has a unique method of redirection for setting and retrieving data.

The following data types are handled in this manner:

- Double
- Enumerations
- Float
- Long
- Short
- ULong
- UShort
- Unbounded Strings
- Object References

For example:

```
::CORBA::Any anything;
anything <<= (::CORBA::Long) 123456;
::CORBA::Long anythingStart  = 123456;
::CORBA::Long anythingLongResult = 0;
policyVar->anything(anything);

::CORBA::Any_var anythingResult_var(policyVar->anything());
::CORBA::Any anythingResult(anythingResult_var);
anythingResult >>= anythingLongResult;
if ( anythingStart != anythingLongResult)
{
   cout << "Anything not set" << endl;
      return 1;
}
else
{
   cout << "Anything set correctly..." << endl;
}
```

There are also specialized structures provided for the following types for conversion with Any:

- Boolean
- Char
- Octet
- String

The data in an Any object is initialized and accessed using insertion (<<=) and extraction (>>=) operators defined by the C++ bindings. These operators are provided (using overloading) by CORBA::Any for each primitive data type, and are provided by the generated C++ bindings for each user-defined IDL type. As a result, there is usually no need to indicate a typecode when inserting or extracting data from a CORBA::Any (although the CORBA::Any class does provide methods for manipulate the data using an explicit TypeCode).

Types that cannot be distinguished by C++ overloading are inserted into and extracted from Any's using special *wrapper* classes. These wrapper classes are not transparent to the application; the application must explicitly create and use them when inserting or extracting ambiguous types into or from Any's. For primitive IDL types that do not map to distinct C++ types (boolean, octet, and char), the wrapper classes are defined within the CORBA::Any scope; they are called from_boolean, to_boolean, from_octet, to_octet, from_char, and to_char. For information on the scope, see the topic "IDL name scoping" on page 150. Because bounded strings cannot be distinguished in C++ from unbounded strings, CORBA::Any provides the from_string and to_string wrapper classes, for inserting/extracting bounded strings. For extracting object references from Any's as the base CORBA::Object type, CORBA::Any provides a to_object wrapper class.

For application-specific arrays, the bindings provide a special forany class, for inserting or extracting the array into or from an Any. For example, here is an IDL array definition:

```
typedef long LongArray[4][5];
```

A C++ class, similar to the following, is emitted for the LongArray:

```
typedef CORBA::Long LongArray[4][5];
typedef CORBA::Long LongArray_slice[5];
class LongArray_forany
{
```

```
   public:
   LongArray_forany();
   LongArray_forany(LongArray_slice*, CORBA::Boolean nocopy=0);
   LongArray_forany(const LongArray_forany );
   ...
};
void operator<<=(Any& , const LongArray_forany&  );
CORBA::Boolean operator>>=(const Any& , LongArray_forany&  );
```

To determine what kind of data is in Any, invoke the type method on a CORBA::Any to access a TypeCode that describes the data it holds. Alternatively, you can try to extract data of a particular type from the Any; the extraction operator returns a boolean to indicate success. If the extraction operation fails, the Any does not hold data of the type you tried to extract.

A CORBA::Any object always owns the data that its void* points to, and deletes (or releases) it when the Any is given a new value or deleted. The only question is whether this data is a copy of the data that was inserted into the Any.

When primitives (including strings and enums) are inserted, a copy is made and returned when the data is extracted.

When a reference to non-primitive (constructed) data is inserted into an Any, a copy is made. In this case, the caller retains ownership of the original data. When a pointer to non-primitive (constructed) data is inserted into an Any, no copy is made. In this case, the Any takes ownership of the storage and caller is forbidden from accessing the storage. When the data is extracted from the Any, the caller is given a pointer that locates the data, but the Any owns the data. The caller should not free this data or reference it after the Any has been given a new value.

The following are examples of the emitted insertion and extraction signatures:

```
void operator <<=(Any&, const T*;);               // insertion by pointer
void operator <<=(Any&, const T&);                // insertion by reference
CORBA::Boolean operator>>=(const Any&, const T*&); // extraction
```

In summary, when extracting data from an Any, the caller does own the data for primitive types, but does not own the data for constructed types. When inserting data into an Any, the caller retains ownership of the data for primitive types, for constructed types inserted by value, and for storage embedded within constructed types inserted by pointer. The caller does not retain ownership of the top-level contiguous storage for a constructed type inserted into an Any by pointer.

The followng is an example that illustrates the previously discussed aspects of CORBA::Any usage. The IDL that follows is used in the succeeding example. It defines a struct and an array that is inserted into an Any:

```
Module M
{
   Struct S
   {
      string str;
      longlng;
   };
   typedef long long1[2][3];
}
```

A C++ program illustrating Any insertion and extraction appears below:

```
#include stdio.h
#include any_C.cpp
main()
```

```
{
   CORBA::Any a;  // the Any that we'll be using
   // test a long
   long l = 42;
   a <<= l;
   if (a.type()->equal(CORBA::_tc_long))
   {
      long v;
      a >>= v;
      printf("the any holds a long = %d\n", v);
   }
   else
   printf("failure: long insertion\n");
   // test a string
   char *str = "abc";
   a <<= str;
   if (a.type()->equal(CORBA::_tc_string))
   {
      char *ch;
      a >>= ch;
      printf("the any holds the string = %s\n", ch);
      delete ch;
      a >>= ch;
      printf(" the any still holds the string = %s\n", ch);
      delete ch;
   }
   else
   printf("failure: string insertion\n");
   // test a bounded string -- note you do not use a typecode here
   char *bstr = "abcd";
   char *rstr;
   a <<= CORBA::Any::from_string(bstr, 6);
   if (a >>= CORBA::Any::to_string(rstr,6))
   printf("the any holds a bounded string<6> = %s\n", rstr);
   else
   printf("failure: bounded string insertion\n");

   // test a user-defined struct
   M::S *s1 = new M::S;
   char *saveforlater = CORBA::string_dup("abc");
   s1->str = saveforlater;
   s1->lng = 42;
   a <<= s1; // insertion by pointer
   if (a.type()->equal(_tc_M_S))
   {
      M::S *sp;
      a >>= sp;
      printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
   }
   else
   printf("failure: struct insertion by pointer\n");
   M::S s2;
   s2.str = CORBA::string_dup("def");
   s2.lng = 23;
   a <<= s2; // note: this deletes *s1, but not saveforlater
   printf("saveforlater still = %s\n", saveforlater);
   CORBA::string_free(saveforlater);
   if (a.type()->equal(_tc_M_S))
   {
      M::S *sp;
      a >>= sp;
      printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
   }
   else
   printf("failure: struct insertion by value\n");
   M::S_var s3 = new M::S;
   s3->str = CORBA::string_dup("ghi");
```

```
   s3->lng = 96;
   a <<= *s3;
   if (a.type()->equal(_tc_M_S))
   {
      M::S *sp;
      a >>= sp;
      printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
   }
   else
   printf("failure: struct insertion by ref to value\n");
   // test an array
   M::long1_var l1v =  M::long1_alloc();
   for (i=0;i<2;i++)
   for (j=0;j<3;j++)
   l1v[i][j] = (i+1)*(j+1);
   a <<= M::long1_forany(l1v);
   if (a.type()->equal(_tc_M_long1))
   {
      M::long1_forany l1s;
      a >>= l1s;
      printf("the any holds the array: ");
      for (i=0;i<2;i++)
      for (j=0;j<3;j++)
      printf("%d ",l1s[i][j]);
      printf("\n");
   }
   else printf("failure: array insertion\n");
}
```

Output from the above program is:

```
the any holds a long = 42
the any holds a string = abc
the any still holds a string = abc
the any holds a bounded string<6> = abcd
the any holds an M::S = {abc, 42}
saveforlater still = abc
the any holds an M::S = {def, 23}
the any holds an M::S = {ghi, 96}
the any holds the array: 1 2 3 2 4 6
```

C++ bindings for CORBA Array types: An Interface Definition Language (IDL) array type is mapped to the corresponding C++ array definition. There also is a corresponding _var type. For example, given the following IDL definition:

```
typedef long LongArray [4][5];
```

The C++ bindings are similar to the following definitions:

```
typedef CORBA::Long LongArray[4][5];
typedef CORBA::Long LongArray_slice[5];
typedef LongArray_slice* LongArray_slice_vPtr;
typedef const LongArray_slice* LongArray_slice_cvPtr;
class LongArray_var
{
   public:
   LongArray_var();
   LongArray_var(LongArray_slice*);
   LongArray_var(const LongArray_var&);
   LongArray_var & operator= (LongArray_slice*);
   LongArray_var & operator= (const LongArray_var &);
   ~LongArray_var();
   const LongArray_slice& operator[] (int) const;
   const LongArray_slice& operator[] (CORBA::ULong) const;
   LongArray_slice & operator[] (int);
   LongArray_slice & operator[] (CORBA::ULong);
   operator LongArray_slice_cvPtr () const;
```

```
   operator LongArray_slice_vPtr& ();
   const LongArray_slice* in() const;
   LongArray_slice* inout();
   LongArray_slice* out();
   LongArray_slice* _retn();
};
LongArray_slice * LongArray_alloc();
void LongArray_free (LongArray_slice*);
LongArray_slice * LongArray_dup (const LongArray_slice*);
LongArray_copy (LongArray_slice* to, const LongArray_slice* from);
```

As shown previously, array mappings provide the following funcations:

alloc function
: This is used for allocating storage. The alloc function dynamically allocates an array, which can be later freed using the free function.

dup function
: This is used for duplicating arrays. The dup function dynamically allocates an array and copies the elements of an existing array into it.

copy function
: This is used for copying array elements. The copy function copies elements from a previously allocated array to another previously allocated array.

free function
: This is used for freeing array storage. The free function frees an array allocated using the alloc or dup function and properly releases the elements of the array. A NULL pointer can be passed to the free function.

None of these functions throws exceptions.

The type of the pointer returned from LongArray_alloc is LongArray_slice*. The C++ bindings define ″slice″ types for all arrays declared in IDL to indicate the type of the array. In this case, the slice type is an array of Long. The slice type has one less array dimension than the array. Thus, the bindings for LongArray include the following typedef:

```
typedef Long LongArray_slice[5];
```

Hence, LongArray_slice* is the correct type for do describe an array of Long arrays.

As with structs and sequences, arrays use special auxiliary classes for automatic storage management of string and object reference elements. The auxiliary classes for strings and object references manage storage the same way the associated _var classes do.

When the array is allocated, the default constructor for each element is automatically invoked to construct the element. If the array's elements are object references, the elements are set to nil when the array is allocated. If the array's elements are strings or wstrings, the elements are set to the empty string.

When assigning a value to an array element that is an object reference, the assignment operator automatically releases the previous value, if any. When assigning an object reference pointer to an array element, the array assumes ownership of the pointer (no _duplicate is done) and the application must no longer access the pointer directly. If this is not the desired behavior, then the caller can explicitly _duplicate the object reference before assigning it. However, when assigning to an object reference array element from a _var object or from another

struct, union, array, or sequence member (rather than from an object reference pointer), a _duplicate is done automatically.

The following is an example that involves multidimensional arrays and array_vars from the IDL snippet at the end of this article:

```
typedef string s2_3[2][3];
typedef string s3_2[3][2];
```

The code at the end of this article uses the C++ arrays that correspond to the previous IDL snippet. In the following example, there is no need to explicitly use slice types when working with the array _var types. This is possible because the bindings declare the pointer held by an array _var type using the appropriate slice type. At the end, the program explicitly frees the storage pointed to by s2_3p (using an array delete operator), but does not do this for s3_2v. Instead, its pointer is deleted when the destructor for s3_2v is implemented. This is the purpose of the _var types.

```
#include arr_C.cpp
#include stdio.h
main()
{
   int i,j;
   char id[40];
   // create arrays
   s2_3_slice* s2_3p = s2_3_alloc();
   s3_2_var s3_2v = s3_2_alloc();
   // load the arrays
   for(i=0; i<2; i++)
   {
      for (j=0; j<3; j++)
      {
         sprintf(id, "s2_3 element [%d][%d]",i,j);
         // Use string_dup when assigning a char*
         // if you do not want the array to own the original:
         s2_3p[i][j] = CORBA::string_dup(id);
      }
   }
   for(i=0; i<3; i++)
   {
      for (j=0; j<2; j++)
      {
         sprintf(id, "s3_2_var element [%d][%d]",i,j);
         // Use string_dup when assigning a char*
         // if you do not want the array to own the original:
         s3_2v[i][j] = CORBA::string_dup(id);
      }
   }
   // print the array contents
   for(i=0; i<2; i++)
   {
      for (j=0; j<3; j++)
      {
         printf("%s\n", s2_3p[i][j]);
      }
   }
   for(i=0; i<3; i++)
   {
      for (j=0; j<2; j++)
      {
         printf("%s\n", s3_2v[i][j]);
      }
   }
```

```
   delete [] s2_3; // needed to prevent a storage leak.
   // Nothing is needed for s3_2v, because
   // it is a _var type.
}
```

Output from the above program is:

```
s2_3 element [0][0]
s2_3 element [0][1]
s2_3 element [0][2]
s2_3 element [1][0]
s2_3 element [1][1]
s2_3 element [1][2]
s3_2_var element [0][0]
s3_2_var element [0][1]
s3_2_var element [1][0]
s3_2_var element [1][1]
s3_2_var element [2][0]
s3_2_var element [2][1]
```

C++ bindings for CORBA Atomic data types: The atomic Interface Definition Language (IDL) data types (long, short, unsigned long, unsigned short, float, double, char, boolean, and octet) are mapped into types defined in corba.h, which are nested within the CORBA scope. See "IDL name scoping" on page 150 for more information. The first letter of the mapped type is capitalized. For example, to introduce and initialize a local variable named `Myvar` whose type corresponds to the IDL type named long, a C++ programmer might use the following expression:

```
CORBA::Long Myvar = 1;
```

The mapping for the IDL boolean type (CORBA::Boolean) defines only the values 0 and 1. The unsigned long and unsigned short IDL types are mapped to CORBA::ULong and CORBA::UShort, respectively.

C++ bindings for CORBA Enumerations: An Interface Definition Language (IDL) enum is mapped to a corresponding C++ enum. For example, given the following IDL:

```
module M
{
   enum Color
   {
      red, green, blue
   };
};
```

a C++ programmer might introduce a local variable of the corresponding C++ type and initialize it with the following code:

```
{
   M::Color MYCOLOR = M::red;
}
```

Note: The name of the enumeration value is `M::red` not `M::Color::red`. The enum construct does not introduce a nested scope; the enumeration value identifiers are in the same scope as the name of the enum. For this reason, choose the names of the enumeration values carefully so that they do not conflict with other IDL identifiers.

C++ bindings for CORBA Sequence types: An Interface Definition Language (IDL) sequence type is mapped to a C++ class that behaves like an array with a current length (how many elements are stored) and a maximum length (how much storage is currently allocated). The array indexing operator [] is used to read and

write sequence elements. The indexing begins at zero. It is the programmer's responsibility to check the current sequence length to prevent accessing the sequence beyond its bounds. The length and maximum of the sequence are not increased automatically to accommodate new elements; the programmer must explicitly increase them.

There are two kinds of sequences: bounded and unbounded. A bounded sequence has a maximum length that is part of the IDL sequence type and cannot be changed. An unbounded sequence has a flexible maximum length that can be set during construction and modified during processing. In either case, the maximum length determines the amount of buffer storage used by the sequence to hold its elements.

The sequence class contains a member function ULong length() to query the number of elements in the sequence. Another member function, length(ULong newLength), indicates the number of elements in the sequence. Increasing the length causes new elements to be added to the end of the sequence. These new elements are default constructed (similar to the default construction of fields for an IDL struct). Decreasing the length causes the elements at the end of the seqence to be released (if the sequence owns the buffer). The length of a bounded sequence cannot exceed the maximum length. If the length of an unbounded sequence is increased past the maximum, the sequence allocates a new buffer and copies the sequence elements to the new buffer.

The buffer, used by the sequence, can either be managed (owned) by the sequence or managed by the client code. By default, the sequence manages its own storage. The release() method indicates whether the sequence manages the buffer (release() returns TRUE if the sequence manages the buffer). Under most circumstances, it is advisable to let the sequence manage its own buffer.

There are a number of methods available to control the management of the sequence's buffer. The programmer must allocate a buffer using the allocbuf function. The allocated buffer is given to the sequence using either the specialized constructor *seq*(ULong max, ULong length, *Data** buffer, Boolean release) or the replace(ULong max, ULong length, *Data** buffer, Boolean release) method. During processing, the programmer can query the buffer directly using the const *Data** get_buffer() const method or can modify the buffer using the *Data** get_buffer(Boolean orphan) method. If the orphan flag is set to TRUE, the sequence yields ownership of the buffer to the client.

The destructor destroys each of the sequence elements if the sequence owns the buffer.

The copy constructor creates a new sequence with the same maximum and length as the input sequence and copies the sequence elements to the storage that the sequence owns. The assignment operator performs a deep copy and releases the previous sequence elements, if the sequence owns the buffer. This operator behaves as if the destructor was run and is followed by the copy constructor.

The allocbuf function allocates enough storage for the specified number of sequence elements. Each sequence element is initialized using its default constructor. The string elements are initialized to the empty string and the object reference elements are initialized to nil object references. NULL is returned if the storage cannot be allocated for any reason. If ownership of the allocated buffer is

not transferred to a sequence, the buffer should be subsequently freed using the freebuf function. The freebuf function ensures the following before the buffer is deleted:

- Each sequence element's destructor is run
- For strings, CORBA::string_free is called
- For object references, CORBA::release is called

Neither allocbuf nor freebuf throw CORBA exceptions.

As with structs, sequences use special auxiliary classes for automatic storage management of string and object reference elements. These auxiliary classes manage strings and object references just as the associated _var classes do.

If a storage-managed sequence's elements are object references, assignment to an element (using operator[]) automatically releases the previous value. If the source of the assignment is an object reference pointer, the sequence assumes ownership of the pointer and no _duplicate is done. If the source is an object reference (_var object, struct field, array element, etc.), a _duplicate is done automatically.

If a storage-managed sequence's elements are strings, assignment to an element (using operator[]) automatically frees the previous string. As with assigning to String_vars, assigning a char* to a string element does not make a copy, but assigns a const char *, a String_var, or another struct, union, array, sequence. A string member automatically makes a copy. Thus, never assign a string literal (such as "abc") to an element without an explicit cast to const char*. When assigning a char* that occupies static storage (rather than one that was dynamically allocated), the caller can use CORBA::string_dup to duplicate the string before assigning it.

There is a corresponding _var type defined for every sequence class.The _var type for a sequence provides an overloaded operator[] that delegates to the underlying sequence.

The following example illustrates loading and accessing the elements of a sequence. It illustrates a recursive sequence whose entries are structs of the same type that contain the sequence. The IDL that follows is used in the succeeding example:

```
struct S
{
   long sf1;
   sequence sf2;
};
typedef sequence Sseq;
```

The following is an example program that creates and loads a sequence of type Sseq and then prints out its contents:

```
#include seq_C.cpp
#include stdio.h
main()
{
   int i,j;
   Sseq seq;      // create an Sseq
   seq.length(3); // set length of seq to 3
   for (i=0; i<3; i++) { // index the three S structs in seq
   seq[i].sf1 = i;    // place a number in the i-indexed struct
   seq[i].sf2.length(i+1); // set length of the sequence in
   //   the i-indexed struct
   for (j=0; j<i+1; j++) // index the i+1 S structs in the sequence
   //  in the i-indexed struct
```

```
   seq[i].sf2[j].sf1 = (i+1)*10+j; // place a number in
   //   the j-indexed struct
}
// OK. Print out what you have created!
printf("seq = (%d sequence elements)\n", seq.length());
for (i=0; i<3; i++)
{
   printf("   struct[%d] = {\n", i);
   printf("      sf1 = %d\n", seq[i].sf1);
   printf("      sf2 = (%d sequence elements)\n",
   seq[i].sf2[j].length());
   for (j=0; j<i+1; j++)
   {
      printf("         struct[%d] = \n",j);
      printf("            sf1 = %d\n", seq[i].sf2[j].sf1);
      printf("            sf2 = (%d sequence elements)\n",
      seq[i].sf2[j].sf2.length());
      printf("         }\n");
      }
   printf("   }\n");
   }
}
```

Note: The previous program never explicitly constructs any data of type S, even though the sequences contain structs of this type. The reason is that when a sequence buffer is allocated, default constructors are run for each of the buffer elements. When the previous program sets the length of a sequence of S structs (either at the top level for the seq variable or for the sf2 field of an S struct in seq), the resulting buffer is populated automatically with default structs of type S.

The output from the previous program is:

```
seq = (3 sequence elements)
   struct[0] = {
      sf1 = 0
      sf2 = (1 sequence elements)
         struct[0] = {
            sf1 = 10
            sf2 = (0 sequence elements)
         }
   }
   struct[1] = {
      sf1 = 1
      sf2 = (2 sequence elements)
         struct[0] = {
            sf1 = 20
            sf2 = (0 sequence elements)
         }
         struct[1] = {
            sf1 = 21
            sf2 = (0 sequence elements)
         }
   }
   struct[2] = {
      sf1 = 2
      sf2 = (3 sequence elements)
         struct[0] = {
            sf1 = 30
            sf2 = (0 sequence elements)
         }
         struct[1] = {
            sf1 = 31
            sf2 = (0 sequence elements)
         }
         struct[2] = {
```

```
            sf1 = 32
            sf2 = (0 sequence elements)
         }
   }
```

C++ bindings for CORBA Strings: The mapping for strings is provided by corba.h, within the CORBA scope. See "IDL name scoping" on page 150 for more information. The user-visible types are CORBA::String and CORBA::String_var. CORBA::String is a typedef name for char*. The CORBA::String_var class performs storage management of a dynamically allocated CORBA::String. The following functions are for dynamic allocation and deallocation of memory to hold a string:

- CORBA::string_alloc
- CORBA::string_free
- CORBA::string_dup

A String_var object behaves as a char* except when it is assigned or goes out of scope, the memory it points to is automatically freed by CORBA::string_free. When a String_var is constructed or assigned from a char*, the String_var assumes ownership of the string and the caller must no longer access the string directly. If this is not the desired behavior, as when the char* occupies static storage, the caller can use CORBA::string_dup to copy the char* before assigning it. When a String_var is constructed or assigned from a const char*, another String_var, or a String element of a struct, union, array, or sequence, an automatic copy of the source string is done. The String_var class provides subscripting operations to access the characters within the embedded string.

C++ compilers do not treat a string literal (such as ″abc″) as a const char* upon assignment. Given both a const and a non-const assignment operator, the compiler chooses the non-const operator. As a result, when a string literal is assigned to a String_var, a copy of the string is not made into dynamically allocated memory. The pointer ″owned″ by the String_var points to memory that cannot be freed. Thus, string literals must not be assigned to a String_var without an explicit cast to const char*.

Note: If a string type is used as a field or element type, the bindings initialize the field or element to the empty string ″″ and not null. This rule applies to sequence elements, array elements, struct fields, union fields, and exception fields.

The following is an example using String_var objects:

```
// first some supporting functions for the examples
char* f1()
{
   return "abc";
}
char* f2()
{
   char* s=CORBA::string_alloc(4);strcpy(s,"abc");return s;
}
// then the examples
void main()
{
   CORBA::String_var s1;
   if (0) s1 = f1();// Wrong!! The pointer cannot be freed and
   // no copy is done.
   if (0) s1 = "abc";  // Also wrong, for the same reason.
   const char* const_string = "abcd"; // *const_string cannot be changed
   s1 = const_string; // OK. A copy of the string is made because
   // it is const, and the copy can be freed.
   CORBA::String_var s3 = f2();// OK. no copy is made, but f2
```

```
   // returns a string that can be freed
   CORBA::String_var s4 = CORBA::string_alloc(10); // also OK. no copy
   s4 = s3; // s4 will use string_free followed by string_dup
   long l4 = strlen(s4); // l4 will receive 3
   long l1 = strlen(s1); // l1 will receive 4
   if (l4 >= l1)
   strcpy(s4,s1); // OK, but only because of the condition.
   // note that s4's buffer only has size=4.
   s4 = const_string; // OK. s4 will use string_free followed by
   // string_dup. The copy is made because String_vars
   // must reference a buffer that can be modified.
}
// The s1, s3 and s4 destructors run successfully, freeing their buffers
```

C++ bindings for CORBA Struct types: An Interface Definition Language (IDL) struct type is mapped to a corresponding C++ struct whose field names correspond to those in the IDL declaration and whose field types support access and storage of the C++ types corresponding to the IDL struct field types. Dynamically allocated storage used to hold this type of C++ struct must be allocated and freed using the C++ new and delete operators.

When a struct is constructed, the default constructor for each field is invoked, object reference files are initialized to nil references, and string fields are initialized to an empty string. When a struct is deleted (or goes out of scope), the destructor for each field is invoked. All of the object references are released and all of the strings are freed. The copy constructor performs a deep copy including duplicating object references. The assignment operator acts as a destructor (releasing all memory) followed by a copy constructor.

When assigning a value to a struct field that is an object reference, the assignment operator for the struct field automatically releases the previous value, if any. When assigning an object reference pointer to a struct member, the struct member assumes ownership of the pointer (no _duplicate is made) and the application must no longer access the pointer directly. If this is not the desired behavior, then the caller can explicitly _duplicate the object reference before assigning it to the struct member. However, when assigning to an object reference struct member from a _var object or from another struct, union, array, or sequence member (rather than from an object reference pointer), a _duplicate is made automatically.

When assigning a value to a struct field that is a string or when the struct is deleted or goes out of scope, any previously held (non-null) string is automatically freed. Assigning a char* to a string field does not make a copy, but assigning a const char *, a String_var, or another struct, union, array, sequence string member does make a copy automatically. Never assign a string literal (for example, ″abc″) to a string struct member without an explicit cast to ″const char*″. When assigning a char* that occupies static storage (rather than one that was dynamically allocated), the caller can use CORBA::string_dup to duplicate the string before assigning it.

As with all constructed types, a _var type is provided for managing an instance of the C++ struct that corresponds to an IDL struct. When assigning one struct's _var to another, the receiving _var deletes its current pointer (thus running all contained destructors) and creates a new struct to hold the assignment result. The new struct is initialized using copy constructors for each of the contained fields. For example, if the source struct has an object reference field, the struct _var assignment automatically duplicates this reference.

The IDL that follows is used in the succeeding example, which shows both correct and incorrect ways to create and manipulate the corresponding C++ struct and the corresponding _var type :

```
Interface A
{
   struct S
   {
      string f1;
      A      f2;
   };
};
```

The following code illustrates both correct and incorrect ways to create and manipulate the corresponding C++ struct and the corresponding _var type.

```
{
   A::S_var sv1 = new A::S;
   A::S_var sv2 = new A::S;
   // sv1->f1 = "abc"; -- Wrong! f1 cannot free this pointer later
   sv1->f1 = CORBA::string_alloc(20);
   A_ptr a1 =  // get an A somehow
   A_ptr a2 =  // get an A somehow
   sv1->f2 = a1; // a1 still has ref cnt = 1
   sv2->f1 = CORBA::string_alloc(20);
   sv2->f2 = a2; // a2 still has ref cnt = 1
   sv1 = sv2; // This runs copy ctors, and increments a2's ref cnt.
   // Also, a1's ref count is decremented.
   sv1->f1 = sv2->f1;
}
```

C++ bindings for CORBA Union types: Union fields are not accessible directly to C++ programmers. Instead, the C++ mapping for Interface Definition Language (IDL) unions defines a class that provides accessor methods for the union discriminator and the corresponding union fields. The union discriminator accessor is named _d. The union field accessors are named using the IDL union field names and are overloaded to allow both reading and writing.

Setting a union's value using a field accessor automatically sets the discriminator and releases the storage associated with the previous value, if any. It is an error for an application to attempt to access the union's value through an accessor that does not match the current discriminator value. It is also an error for an application to use the discriminator modifier method to implicitly switch between different union members.

Unions that have an implicit default member (no explicit default case and not all possible discriminator values are used) have a _default method. The _default method sets the discriminator value to a legal default value and does not have a union member value.

A _var type is defined for managing a pointer to a union in dynamically allocated memory.

To illustrate the C++ bindings for IDL unions, consider the following IDL:

```
module A
{
   interface X
   {
   };
   union U switch (long)
   {
      case 1: long u1;
```

```
      case 2: string u2;
      case 3: X u3;
   };
};
```

The following code illustrates usage of the C++ bindings corresponding to the previous IDL:

```
{
   X_ptr x = // get an X somehow
   A::U_var uv = new A::U;
   uv.u2((const char*) "testing");  // sets the discriminator to 2
   // and copies the string
   if (u._d() == 2)// the condition evaluates to true
   u.u1(23); // frees the string, and sets the discriminator to 1
   if (u._d() == 1) // the condition evaluates to true
   u.u3(x);  // duplicates x and sets discriminator to 3
}
```

The default constructor of a union class does not initialize the discriminator or the union members, so the application must initialize the union before accessing it. The copy constructor and assignment operator perform deep copies. The assignment operator and destructor release all storage owned by the union.

With respect to memory management, accessor and modifier methods for union members work similarly to those for struct members. Modifier methods make a deep copy of their input when passed by value (for simple types) or by reference (for constructed types). Accessor methods that return a non-const reference can be used by the application to update a union member's value, but only for struct, union, sequence, and any members.

The modifier method for a string union member makes a copy when given a const char* or a String_var, but not when given a char*. As shown in the example above, a string literal must not be assigned to a union without an explicit ″const char*″ cast. The accessor method for a string union member returns a const char* and therefore, the string union member cannot be modified. This is done to prevent the string union member from being assigned to a String_var and resulting in memory management errors.

The modifier method for an object reference union member duplicates the input object reference and releases the previous object reference value, if any. The accessor method for an object reference union member does not duplicate the returned object reference because the union retains ownership of it.

The accessor method for an array union member returns a pointer to the array slice. Thus, the application can read or write the union-member array elements using subscript operators. If the union member is an anonymous array (one without an explicit type name), the union defines (typedefs) the slice type by cocatenating a leading underscore and appending ″_slice″ to the union member name.

C++ bindings for CORBA WStrings: The WString type provides support for wide strings. It is fairly comparable to using strings except for type declarations and assignments. The following example, uses the WString type:

```
#include wcstr.h // For WChar and WString support
...
const wchar_t* wcomments = L"This policy looks pretty good...";
wchar_t* wcommentsResult=::CORBA::wstring_alloc(wcslen(wcomments));
::CORBA::WString_var wcommentsResult_var(wcommentsResult);
policyVar->wcomments(wcomments);
```

```
if (!wcscmp(wcommentsResult_var, wcomments) )
{
   cout << "Wcomments not set" << endl;
      return 1;
}
else
{
   cout << "Wcomments set correctly..." << endl;
}
wcommentsResult = policyVar->wcomments();
```

CORBA C++ bindings for interfaces

The CORBA 2.1 C++ client bindings define a variety of C++ types corresponding to a single Interface Definition Language (IDL) interface. The following is a list of these interfaces:

- I
- I_ptr
- I_var

For example, an IDL interface I is mapped to C++ client types. The types named I and I_var are classes. The type I_ptr (the object reference) is defined in the IBM Object Request Broker (ORB) as an I*.

The class I is referred to as the interface class corresponding to the IDL interface named I. The IDL constructs defined within the IDL interface I are defined with public access within the C++ class I. For example, the operations within an IDL interface are mapped as C++ virtual member methods within the corresponding C++ class.

As with other user-defined IDL types, the I_var type is used to assist storage management. Specifically, an I_var type holds an I_ptr and can be used as if it were an I_ptr. When an I_var type is assigned a new value or when it goes out of scope, it releases the I_ptr it is holding at that time.

The CORBA specification prohibits CORBA-compliant applications from the following:

- Explicitly creating an instance of an interface class, as in:

  ```
  I my_instance;   // NOT ALLOWED!
  I_ptr my_instance = new I;  // NOT ALLOWED!
  ```

- Declaring a pointer (I*) or a reference (I) to an interface class.

Instead, the I_ptr, and I_var types must be used to hold object references and object references can be created only by client applications by invoking methods that return object references. The interface class I is used by client applications only as a name scope.

IDL operations defined in (or inherited by) interface I are invoked in C++ using the arrow (->) operator on either an I_ptr, IRef, or I_var type.

Nil object references of type I_ptr can be obtained using a static member function of I called _nil(). Operations cannot be invoked on nil object references. The CORBA::is_nil function is the only CORBA-conformant way to determine whether a given object reference is nil. CORBA::release can be invoked on a nil object reference, but is not needed. The _duplicate and _narrow functions defined by the C++ bindings can be given a nil object reference.

In the IBM C++ bindings, the CORBA-prescribed types are implemented as follows:

- The interface class for I is derived using virtual inheritance from the interface classes for I's IDL parents. When I has no IDL parents, its interface class is derived using virtual inheritance from CORBA::Object. Types, constants, and operations declared within the I interface are mapped to types, constants, and member functions declared within the corresponding interface class.
- The object reference type I_ptr is typedefed to I* (for example, an I_ptr points to an object of type I). However, CORBA specifies that treating an I_ptr as a C++ pointer (for example, using conversion to void*, using arithmetic and relational operators, testing for equality) is not conformant. However, this is not enforced by the bindings.
- An instance of I addressed is called a proxy and is created by a proxy factory object of class IProxyFactory.
- Nil object references are represented as NULL pointers. However, CORBA conformant applications must not assume this and must use the _nil() and is_nil() functions to manipulate nil object references.
- The I_var class introduces an instance variable of type I_ptr. The purpose of an I_var object is to handle release operations on the I_ptr that it holds.
- Other auxillary classes are generated for implementation purposes. An I_SeqElem class is generated, which is used to hold I elements in an array. Its function is similar to the I_var class as it automatically controls the releasing of the object_reference. The I_StructElem class is used for holding I items in a field of a struct, union, or exception. Its function is similar to I_SeqElem. The I_out class is used in the bindings when passing I items as out arguments.

For more details on C++ bindings for CORBA interfaces, see the following topics:

- "C++ bindings for CORBA: Managing CORBA object references"
- "C++ bindings for CORBA: Widening CORBA object references"
- "C++ bindings for CORBA: Narrowing CORBA object references" on page 128
- "C++ bindings for CORBA: Narrowing to a C++ implementation" on page 128

C++ bindings for CORBA: Managing CORBA object references: The mapping for interface I defines a static member function named _duplicate that takes as input an object reference of type I_ptr and returns an object reference of type I_ptr (This is potentially the same reference, when reference counting is employed, as is the case with IBM WebSphere Application Server C++ bindings). The CORBA::release function indicates that the caller will no longer access the object reference and that the resources associated with the object reference can be deallocated. (In the IBM WebSphere Application Server C++ bindings, an object reference is only deleted when its reference count falls to zero. This occurs only if CORBA::release is called for each _duplicate or _narrow performed on the object reference.)

Duplicating an object reference using _duplicate is analogous to copying a string before transferring ownership of it. Releasing an object reference is analogous to deleting a string that is no longer needed. Unlike strings, object references cannot be directly copied or deleted by the client programmer Object references are managed by the Object Request Broker (ORB) and can be duplicated or released by the application only.

C++ bindings for CORBA: Widening CORBA object references: If interface A is a (direct or indirect) base of interface B, the following assignments do not require an explicit C++ cast:

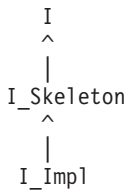
- B_ptr to A_var
- B_ptr to A_ptr
- B_ptr to Object_var
- B_ptr to Object_ptr
- B_var to A_ptr
- B_var to Object_ptr

B_var cannot be assigned to A_var or a compile-time error occurs. To assign B_var to A_var, use B::_duplicate on B_var to create B_ptr and assign B_ptr to A_var.

C++ bindings for CORBA: Narrowing CORBA object references: The mapping for an interface I defines a static member function named _narrow that takes as input an object reference of any type (for example, an Object_ptr) and returns an object reference of type I_ptr. If the referenced object (the actual implementation object corresponding to the proxy addressed by the input object reference) does not support the I interface, the result is NULL. Otherwise, the I_ptr addresses an object that also supports the I interface. In the case where the proxy addressed by the input argument does not support interface I and the actual implementation object does, the I_ptr returned by I::_narrow addresses a different proxy object than the input argument.

The _narrow static member function does an implicit _duplicate of the input argument. Therefore, the caller is responsible for releasing both the object reference input to _narrow and the return result.

C++ bindings for CORBA: Narrowing to a C++ implementation: Given an interface pointer to an object, it can be useful to narrow to the implementation pointer of the object. For example, given interface I, the C++ implementation hierarchy for I might look like the following:



You might convert a pointer to I into a pointer to I_Impl. There is no CORBA-prescribed mechanism for this conversion. Within the confines of the C++ language, you can use dynamic cast.

CORBA C++ binding restrictions

When a forward reference to an interface appears within an Interface Definition List (IDL) module, the IDL compiler issues an error message if the referenced interface is not defined within the module. When a similar unresolved forward reference appears at global (file) scope, a warning is issued that indicates the bindings being emitted will not include a mapping for the undefined interface. For information on the scope, see the topic “IDL name scoping” on page 150. The assumption is that the interface is defined by bindings other than those currently being generated. This approach supports IDL files with mutually-referential interfaces (as long as they appear at global scope). The following example illustrates how to organize the IDL files for such cases:

```
// file foo.idl
#ifndef foo_idl
#define foo_idl
interface Foo; // declare Foo so bar.idl can refer to it
```

```
#include bar.idl
interface Foo
{
   Bar foo1(); // notice the use of Bar
};
#endif // foo_idl
// file bar.idl
#ifndef bar_idl
#define bar_idl
interface Bar; // declare Bar so foo.idl can refer to it
#include foo.idl
interface Bar
{
   Foo bar1(); // notice the use of Foo
};
#endif // bar_idl
```

The following is a known limitation due to problems inherent in the CORBA mapping for C++:

The C++ bindings map attributes into overloaded C++ accessor functions whose name is the attribute name. As a result, the following IDL does not map to useful C++ bindings (because Y's l method interferes with the inherited mapping for X's attribute). If Y's method took any arguments, there would not be a problem because of C++ overloading. The compiler indicates an error only when C++ overloading does not distinguish inherited accessors from newly introduced methods (or vice versa).

```
interface X
{
   attribute long l;
}
interface Y : X
{
   long l();
};
```

Name scoping and modules in the C++ bindings

Interface Definition List (IDL) scoped names are mapped to C++ scopes as follows:

- IDL modules are mapped to C++ namespaces. IDL definitions occurring within a module are mapped to corresponding C++ definitions within the C++ namespace.
- IDL interfaces are mapped to C++ classes. All IDL constructs defined within an interface are mapped to corresponding C++ definitions within the C++ interface class.
- If an IDL identifier is the same as the name of a C++ keyword, the IDL identifier is mapped to the same identifier prepended with "_cxx_". For example, an IDL identifier named "class" is mapped to "_cxx_class".

Storage management and _var types

The C++ bindings attempt to make the programmer's storage management responsibility as easy as possible. One aspect of this is the "_var" types. For each user-defined structured Interface Definition List (IDL) type T (struct, union, sequences, and arrays) and for interfaces, the bindings generate both a class T and a class T_var. The classes CORBA::String and CORBA::Any also have corresponding CORBA::String_var and CORBA::Any_var classes.

The essential purpose of a _var object is to hold a pointer to dynamically allocated memory. A _var object can be used as if it were a pointer to the IDL type for which it is named. Special constructors, assignment operators, and conversion operators

make this work in a way that is invisible to programmers. The memory pointed to by a _var object always is considered to be owned (managed) by the _var object. When the _var object is deleted, goes out of scope, or is assigned a new value, it deletes (or, in the case of an object reference, releases) the managed memory.

A typical _var object is declared by a programmer as an automatic (stack) variable within a code block and is then used to receive an operation result or is passed to an operation as an out parameter. Later, when the code block is exited, the _var object destructor runs and its managed memory is deleted (or, for object references, released).

When a pointer (rather than another _var object, struct, union, array, or sequence element) is assigned to (or used to construct) a _var object, this pointer must point to dynamically allocated memory. It must point to dynamically allocated memory because the _var object does not make a copy. Instead, it assumes ownership of the pointer and later deletes it (or, for object references, releases it). The single exception is that pointers to const data can be assigned to a _var object. When this occurs, the _var object dynamically allocates new memory and copies the const data into the new memory. A pointer assigned to a _var object must not be ″owned″ by some other data structure and the pointer must not be subsequently used by the application except by the _var object.

The default constructor for a _var type loads the contained pointer with NULL. You must assign a value to a _var object created by a default constructor before invoking methods on it. This is similar to when you assign a value to a pointer variable before invoking methods on it.

The copy constructor and _var assignment operator of a _var type perform a deep copy of the source data. The copy is later deallocated (or released, in the case of object references) when the _var is destroyed or when it is assigned a new value.

Methods are provided to fully support the passing of _var types as parameters. In most cases, these methods are not necessary, but they might be used solely to self-document the code. The in() method returns the appropriate type for in parameters. The inout() method returns the appropriate type for inout parameters. The out() method returns the appropriate type for out parameters and ensures that any pre-existing value is properly released.

Note: Out signatures use the _out class, which also takes care of releasing pre-existing values.

The _retn() method is useful for obtaining a value from a _var type that contains a returned value. The _retn() method obtains the value and releases/clears the value in the _var.

The following is the typical form for a T_var class that is emitted for an IDL-constructed data type named T:

```
class T_var
{
   public:
   T_var ();
   T_var (T*);
   T_var (const T_var&);
   ~T_var ();
   T_var  &operator= (T*);
   T_var  &operator= (const T_var&);
   T * operator-> ();
```

```
   const T * operator-> () const;
   in_paramater_type in() const;
   inout_paramater_type inout();
   out_paramater_type out();
   return_paramater_type _retn();

   ...
};
```

For more information on storage management and argument passing, see "Argument passing considerations for C++ bindings".

Argument passing considerations for C++ bindings

Rules must be observed when passing parameters to a CORBA object implementation. The type used to pass the parameters of a method signature is dependent on the Interface Definition List (IDL) type and the directionality of the parameter (in , inout, out, or return value).

The following rules for passing these parameters are dictated by CORBA Object Management Group (OMG) IDL to C++ mapping and must be followed to:

- Ensure the required access authority
- Prevent memory leaks
- Ensure that the allocation and deallocation of memory is performed consistently

in parameters

The caller (client) must allocate the input parameters. The callee (implementation) is restricted to read access. The caller is responsible for the eventual release of the storage. Primitive types and fixed-length aggregate types can either be heap allocated or stack allocated. By their nature, variable-length aggregates cannot be completely stack allocated.

Normally, a caller uses a _var object to pass non-primitive arguments as input parameters. The _var class contains an in() method that can be used as additional evidence that the argument is an input argument..

```
call opIn(t_var);
call opIn(t_var.in());
```

inout parameters

For inout parameters, the caller provides the initial value and the callee can change that value. For primitive types and fixed-length aggregates, this is a straightforward process. The caller provides the storage and the callee overwrites the storage on return. For variable-length aggregates, the size of the contained data provided as input might differ from the size of the contained data provided at output. Therefore, the callee is required to deallocate any contained input data that is being replaced on output with callee allocated data. For object references, the caller provides an initial value. If the callee reassigns the value, the callee must first release the original input value. The callee assumes or retains ownership of the returned parameters and must eventually deallocate or release them.

Normally, a caller uses a _var object to pass non-primitive arguments as inout parameters. The _var class contains an inout() method that can be used as additional evidence that the argument is an inout argument.

```
call opInOut(t_var);
call opInOut(t_var.inout());
```

out parameters

For primitive types and fixed-length aggregate types, the caller allocates the storage for the out parameter and the callee sets the value. For

variable-length aggregate types, the caller allocates a pointer and passes it by reference and the callee sets the pointer to point to a valid instance of the parameter's type. For object references, the caller allocates storage for the _ptr and the callee sets the _ptr to point to a valid instance. Because a pointer to an array in C++ must be represented as a pointer to the array element type, CORBA defines an array_slice type, where a slice is an array with all the dimensions of the original except the first. The output parameter is typed as a reference to an array_slice pointer. The caller allocates the storage for the pointer and the callee updates the pointer to point to a valid instance of an array_slice. The caller assumes or retains ownership of the output parameter storage and must eventually deallocate it or, in the case of object references, release it.

The callee can assume that the output argument does not have a value. If an output argument contains an initial value, the value is orphaned and never released. To prevent this from happening, all of the out parameters are defined in the method signature with an _out class. The _out class is a special generated class that ensures that the values are properly released before the callee gets control. The callee can treat _out objects as if they were the underlying type.

Normally, a caller uses a _var object to pass non-primitive arguments as out parameters. The _var class contains an out() method that can be used as additional documentation that the argument is an out argument.

```
call opOut(t_var);
call opOut(t_var.out());
```

return values

For primitive types and fixed-length aggregate types, the caller allocates the storage for the return value and the callee returns a value for the type. For variable-length aggregate types, the caller allocates a pointer and the callee returns a pointer to an instance of the type. For object references, the caller allocates storage for the _ptr and the callee returns a _ptr that points to a valid object instance. Because a pointer to an array in C++ must be represented as a pointer to the array element type, the array_slice type is used for returning array values. The caller allocates storage for a pointer to the array_slice and the callee returns a pointer to a valid instance of an array_slice. The caller assumes or retains ownership of the storage associated with returned values and must eventually deallocate it or, in the case of object references, release it.

These rules for passing parameters are captured and enforced by the header files produced when an IDL interface description is compiled. Some rules cannot be enforced by the bindings. For example, parameters that are passed or returned as a pointer type (T*) or reference to pointer(T*&) must not be passed or returned as a null pointer. Memory management responsibilities cannot be enforced by the bindings. Client (caller) and implementation (callee) programmers must understand and implement according to these rules.

For more detailed information on storage management and argument passing, see the following topics:

- "C++ type mapping for argument passing"
- "Storage management responsibilities for arguments" on page 134

C++ type mapping for argument passing:

Argument type mappings are discussed in this topic and summarized in the two tables that follow. For the rules that must be observed when passing parameters (in , inout, out, or return value) to a CORBA object implementation, see "Argument passing considerations for C++ bindings" on page 131.

For primitive types and enumerations, the type mapping is straightforward. For in parameters and return values, the type mapping is the C++ type representation (abbreviated as "T" in the text that follows) of the Interface Definition List (IDL) specified type. For inout and out parameters the type mapping is a reference to the C++ type representation (abbreviated as "T&" in the text that follows).

For object references, the type mapping uses _ptr for in parameters and return values and _ptr& for inout and out parameters. That is, for a declared interface A, an object reference parameter is passed as type A_ptr or A_ptr&. The conversion functions on the _var type permit the client (caller) the option of using _var type rather than the _ptr for object reference parameters. Using the _var type can have an advantage because it relieves the client (caller) of the responsibility to deallocate a returned object reference (out parm or return value) between successive calls. This is because changing the assignment operator of a _ptr to a _var automatically releases the embedded reference.

The type mapping of parameters for aggregate types (also referred to as complex types) are complicated by when and how the parameter memory is allocated and deallocated. Mapping in parameters is straightforward because the parameter storage is caller-allocated and read-only. For an aggregate IDL type t with a C++ type representation of T, the in parameter mapping is const T&. The mapping of out and inout parameters is slightly more complex. To preserve the client capability to stack allocate fixed length types, Object Management Group (OMG) has defined the mappings for fixed-length and variable-length aggregates differently. The inout and out mapping of an aggregate type represented in C++ as T is T& for fixed-length aggregates and as T*& for variable-length aggregates.

Basic argument and result passing

Data Type	In	Inout	Out	Return
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
long long	LongLong	LongLong&	LongLong&	LongLong
unsigned short	UShort	UShort&	Ushort&	Ushort
unsigned long	ULong	ULong&	Ulong&	Ulong
unsigned long long	ULongLong	ULongLong&	ULongLong&	ULongLong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
long double	LongDouble	LongDouble&	LongDouble&	LongDouble
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
wchar	Wchar	Wchar&	Wchar&	Wchar
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr

Data Type	In	Inout	Out	Return
struct, fixed	const struct&	struct&	struct&	struct
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union
union, variable	const union&	union*&	union*&	union*
string	const char*	char*&	char*&	char*
wstring	const Wchar*	Wchar*&	Wchar*&	Wchar*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*
array, variable	const array	array	array slice*&	array slice*
any	const any&	any&	any*&	any*
valuetype	valuetype*	valuetype*&	valuetype*&	valuetype*

For an aggregate type represented by the C++ type T, the T_var type also is defined. The conversion operations on each T_var type allows the client (caller) to use the T_var type directly for any directionality, instead of using the required form of the T type (T, T& or T*&) The emitted bindings define the operation signatures in terms of the parameter passing modes shown in the 134 and result passing table and the T_var types provide the necessary conversion operators to allow them to be passed directly.

T_var argument and result passing

Data Type	In	Inout	Out	Return
object reference_var	const object_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var
valuetype_var	const valuetype_var&	valuetype_var&	valuetype_var&	valuetype_var

Do not pass or return a null pointer for parameters that are passed or returned as a pointer type (T*) or reference to pointer(T*&). However, this cannot be enforced by the bindings.

Storage management responsibilities for arguments:

The storage access and allocation responsibilities for argument passing are summarized in the following two tables. For the detailed rules that must be observed when passing parameters (in , inout, out, or return value) to a CORBA object implementation, see “Argument passing considerations for C++ bindings” on page 131.

As an overall requirement when allocating and deallocating argument storage, the storage allocation rules for the specific type must be followed. Specifically, for strings, sequences, and arrays or for aggregate types composed of these types, use the associated memory allocation and dealloaction functions. For string types, use the string_alloc(), string_dup(), and string_free() methods. For sequence types, use the allocbuf() and freebuf() methods. For arrays, use the T_alloc(), T_dup() and T_free() methods. The memory deallocation responsibilities of the client can be minimized by stack allocation and with the use of the _var types whenever possible. When an argument is passed or returned as a pointer type, do not pass or return a NULL pointer value.

The following table shows the storage access responsibilities for argument passing.

Argument storage responsibilities

Data Type	Inout	Out	Return
short	1	1	1
long	1	1	1
long long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
unsigned long long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
wchar	1	1	1
octet	1	1	1
enum	1	1	1
object reference pointer	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
wstring	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3
valuetype	7	7	7

For definitions of the numerical values in the previous table, refer to the table below.

Argument passing cases

Case	Description
1	The caller allocates all of the necessary storage, except the storage that can be encapsulated and managed within the parameter itself. For inout parameters, the caller allocates the storage but does need not to initialize it. The callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value. If the callee wants to reassign the inout parameter, it first calls CORBA:release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The caller is responsible for the release of all out and return object references. Release of all of the object references embedded in other structures is performed automatically by the structures themselves.
3	The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local and remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage: To modify the values, the caller first must copy the returned instance into a new instance and then modify it.
4	For inout strings, the caller provides storage for both the input string and the char* pointing to it. Because the callee can deallocate the input strings and reassign the char* to point to new storage to hold the output value, the caller should allocate the input string using string_alloc(). The size of the out string is therefore not limited by the size of the in string. The caller is responsible for deleting the storage for the out using string_free(). The callee is not allowed to return a null pointer for an inout, out, or return value.
5	The assignment or modification of an inout sequence or the any type might cause deallocation of owned storage before any reallocation occurs. This depends upon the state of the Boolean release parameter with which the sequence or the any type was constructed.

Case	Description
6	For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer. In both cases, the caller always must release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage. To modify any values, the caller must first copy the returned array instance into a new array instance, and then modify the new instance.
7	The caller allocates storage for the valuetype instance. For inout parameters, the caller provides an initial value. If the callee wants to reassign the inout pointer to a different valuetype instance, it first calls _remove_ref on the original input valuetype. To continue to use a valuetype instance passed in as an inout after the invoked operation returns, the caller first must invoke _add_ref on the valuetype instance. The caller is responsible for invoking _remove_ref on all out and return valuetype instances. The structures themselves reduce the reference counts using_remove_ref for all valuetype instances embedded in other structures.

Implementation registration utility (regimpl)

Before a server can be used by client applications, it must be registered in the Implementation Repository by running the implementation registration utility, regimpl.

The regimpl utility can be entered at a command prompt. For IBM WebSphere Application Server enterprise services, the regimpl utility takes the following usage syntax forms and parameters:

Syntax

To add an implementation

```
regimpl -A -i <em>alias_string</em> [-p <em>svr_string</em>] [-m
{on|off}] [-t <em>string</em>]
```

To update an implementation

```
regimpl -A -i <em>alias_string</em> [-p <em>svr_string</em>] [-m
{on|off}] [-t <em>prot_string</em>]
```

To delete one or more implementations

```
regimpl -D -i <em>alias_string</em> [-i ...]
```

To list all, or selected, implementations

```
regimpl -L [-i <em>alias_string</em> [-i ...]]
```

To list all implementation aliases

```
regimpl -S
```

Parameters

The following table describes the command parameters used in the previous syntax forms.

regimpl command parameters used by WebSphere enterprise services

Parameter	Description
-i alias_string	Implementation alias name (maximum of 16 -i names)
-p svr_string	Server program name (default: null)
-m {on|off}	Enable multi-threaded server (optional)
-t prot_string	Protocol name (default: SOMD_TCPIP)

Examples

```
regimpl -L

CORB1150I: Retrieving all aliases from the database...
============================================================
Information for implementation definition 1:
   ID:                      1bc727f8-7f95-1dad-e000-079809355cc6
   ALIAS:                   WASDAEMON
   PROGRAM:
   PROTOCOLS:               TCPIP
   CONFIG:
   Server specific name  1 : DAEMON_PID
   Server specific value 1 : 00000634
------------------------------------------------------------

============================================================
Information for implementation definition 2:
   ID:                      13463dad-7fe1-1dad-e000-030409355cc6
   ALIAS:                   indiserver
   PROGRAM:                 indiserv.exe
   PROTOCOLS:               TCPIP
   CONFIG:
------------------------------------------------------------

============================================================
Information for implementation definition 3:
   ID:                      042f1128-7fed-1dad-e000-07c809355cc6
   ALIAS:                   wasirsvr
   PROGRAM:                 wasirsvr.exe
   PROTOCOLS:               TCPIP
   CONFIG:
------------------------------------------------------------

CORB1148I: Exiting the regimpl utility
```

CORBA exceptions

The preferred coding practice for handling errors in C++ and Java is using exceptions. The CORBA programming model supports this coding practice by using the standard try and throw logic of exception handling. Handling exceptions

are a critical part of the programming model. The exceptions that are thrown must be understood and handled appropriately by application developers.

No matter how much care an object provider takes in implementing a business object, there are times when things go wrong. In these cases, a business object might need to throw an exception to the client to give the client the opportunity to recover from the error.

The detail of exception handling is given in the following topics:

- CORBA exceptions: Catching
- CORBA exceptions: Throwing
- CORBA system exception minor codes

For further information regarding C++ exceptions and their usage, see a standard C++ book.

CORBA exceptions: Catching

You must handle exceptions in client programs. Remember that any method might throw a standard exception. Therefore, an exception can be thrown by these methods at any time - even if there are no exceptions declared in the `raises` clause of that method. The default behavior for uncaught exceptions is to end that process. If this happens, suspect an uncaught exception first. The exact style of how or what exceptions are caught depends on what the client application does for error recovery. However, the following is a list of some general rules:

- Perform the most specific error recovery that you need. By properly structuring `catch` clauses, specific error recovery can be done.
- Check for most specific exceptions first and most general exceptions last.
- Make use of the information that is available in the exception. All of the CORBA exceptions support the .id() method that returns the exception identifier. System exceptions also provide .minor() and .completed() methods that return the minor code and completion status respectively.

The following is a simple client code example:

```
try
{
   // Some real code goes here
   foo.boo();
}
// Catch any specific User exceptions defined for the method in the
// `raises' clause
catch (IManagedClient::INoObjectWKey  nowk)
{
   // Process the error, more specific recovery could be made here
   // because the specific error is known
}
// Catch and process any other specific User exceptions
...
// Catch any other User exceptions defined for the method in the
// `raises' clause
catch (CORBA::UserException  ue)
{
   // Process any other User exceptions. Use the .id() method to
   // record or display useful information
   cout << "Caught a User Exception: " << ue.id() << endl;
}
// Catch any System exceptions defined for the method in the
// `raises' clause
catch (CORBA::SystemException  se)
{
```

```
   // Process any System exceptions. Use the .id(), and .minor()
   // methods to record or display useful information
   cout << "Caught a System Exception: " << ue.id() << ": " <<
        ue.minor() << endl;
}
catch (...)
{
   // Process any other exceptions. This would catch any other C++
   // exceptions and should probably never occur
   cout << "Caught an unknown Exception" << endl;
}
```

Specific standard exceptions cannot be caught individually. If processing individual standard exceptions is required, it can be done within the CORBA::SystemException catch block using the .id() method.

CORBA exceptions: Throwing

A business object might wrap existing logic that might not be written in C++ or might not use the exception paradigm. These business objects must convert the existing exceptions or error return codes to CORBA exceptions that can be returned to the client program.

Any non-CORBA exception thrown by the business object is mapped automatically to CORBA::UNKNOWN by the framework. This does not provide specific information to the client and severely limits the error recovery capability of the client program. These C++ exceptions should be mapped to appropriate CORBA exceptions by the business object.

CORBA system exception minor codes

In the CORBA model for exception handling, a system exception might contain an associated minor code. This topic provides details of these minor codes, grouped by system exception.

Minor codes are used in several ways:

- Returned in the minor code field of exception bodies (when appropriate).
- Placed in the message log as part of the PrimaryMessage.
- Can be written in diagnostic messages on a computer screen.

Each minor code consists of a 5-digit hexadecimal vendor identifier followed by a 3-digit hexadecimal value, which indicates the specific reason for the system exception. A minor code containing a vendor identifier of 0x4F4D0 is an Object Management Group (OMG)-assigned minor code. A minor code containing a vendor identifier of 0x49420 is an IBM-assigned minor code.

Specific minor code values are meaningful only within the context of the particular system exception in which they are contained. Minor code values might be used in more than one system exception type, but the system exception is used to interpret the minor code value. For example, the minor code value 0x4F4D0001 means ″the repository ID is already defined in the Interface Repository″ when associated with a BAD_PARAM exception, However, the same code means ″unable to locate and/or use the appropriate Value Factory″ when associated with a MARSHAL exception.

In this topic, the description of each minor code consists of:

The System Exception (in alphabetical order)
A description of the problem that caused the error.

User Response: Actions are needed to resolve the problem, if appropriate.

Minor Code number (prefixed with vendor ID). In some cases, the minor code number is followed by the following entries:

- **Error Text** A string that identifies the minor code.
- **Details** Details for this specific minor code, in the context of the System Exception.

Minor code definitions

BAD_CONTEXT

Explanation: Cannot find a specified CORBA::Context property.

User Response: Ensure that the specified property name exists in the context object.

0x49420047 SOMDERROR_CtxNoPropFound

Error Text: A string that identifies the minor code.

Details: This error occurs if an invalid property name was passed to CORBA::Context::delete_values.

BAD_INV_ORDER

Explanation: Operations were invoked in an improper order.

User Response: Verify the proper order of the operations and correct the order.

0x4942005D SOMDERROR_BadInvOrder

Details: This is the generic bad invocation order minor code.

0x4F4D0004

Details: The Object Request Broker (ORB) has been shut down, but an ORB operation was requested.

0x4F4D000A

Details: There is an invalid Portable Interceptor call or a valid Portable Interceptor call in an invalid order.

0x4F4D000B

Details: A request was made to add a new service context to the Portable Interceptor. The service context was found to exist already but the requester specified that it not be replaced.

BAD_OPERATION

Explanation: A bad class, method, operation, or object reference was encountered.

User Response: Verify the operation and make sure the correct bindings exist.

0x49420045 SOMDERROR_ClassNotFound

Details: Cannot convert an Interoperable Object Reference (IOR) to an object. The class name was unknown or the proxy factory cannot be created. Verify the class implementation and verify that the bindings exist.

0x4942004DSOMDERROR_WrongRefType

Details: The wrong type of object reference was used. Probably, a client invoked an operation on an object in a server and the object did not

support the invoked method. To support a given operation, a server must be compiled and linked with the server-side C++ bindings for the interface that introduces that IDL operation. This error also occurs when a server application invokes CORBA::BOA::get_id and passes in a proxy object rather than a local object. Verify that a server is compiled and linked with all of the server-side C++ bindings for the interfaces.

BAD_PARAM

Explanation: An application supplied an invalid parameter to an operation.

User Response: Check the error log for a message that indicates which operation was given the invalid parameter. Check the documentation for that operation and verify that the passed parameters are valid.

0x49420048 SOMDERROR_BadParm

Details: This is the generic bad parameter minor code.

0x4F4D0001

Details: Interface Repository. An operation specified an object with an ID that already exists in the Interface Repository (IR) database container.

0x4F4D0002

Details: Interface Repository. Repository ID is defined already in the Interface Repository.

0x4F4D0003

Details: Interface Repository. Name is used already in the context in the Interface Repository.

0x4F4D0004

Details: Interface Repository. The target is not a valid container.

0x4F4D0005

Details: Interface Repository. The name clash is in inherited context.

0x4F4D0017

Details: A Portable Interceptor operation cannot find the specified service context.

0x4F4D0018

Details: A null object was passed into register_initial_reference().

0x4F4D0019

Details: A Portable Interceptor operation cannot find the specified tagged component on the target object.

0x4F4D001A

Details: A Portable Interceptor operation was given a profile ID that is not supported. The supported profile id's are IOP::TAG_INTERNET_IOP and IOP::TAG_MULTIPLE_COMPONENTS.

COMM_FAILURE

Explanation: A communications failure occurred. Possible reasons are:

- A process might have received an unknown or unexpected message type or message content.

- The process might have encountered a low-level communications failure in attempting to send a message or binding to a socket.
- An unexpected broken connection might have occurred.

User Response: See the details of the following minor codes.

0x49420038 SOMDERROR_HostAddress

Details: Cannot map a host name on a different machine to a host address. Ensure that the host to which this process is attempting to communicate is known and can be reached via TCP/IP. Ping the remote host by the host name.

0x4942003ESOMDERROR_CannotConnect

Details: A client process cannot connect to a server process when attempting to invoke a method on a proxy to an object residing in that server process. Ensure that the location service daemon is running. Ensure that the object reference is still valid. Ping the remote machine to see that the two machines are connected.

0x49420042SOMDERROR_CommunicationsError

Details. This is the generic communications error minor code. Ensure that communications resources are functioning properly. For example, when using TCP/IP, ping the remote host. Ensure that the process has not failed due to an application error.

0x49420059SOMDERROR_Server_Connection_Broken

Details. The location service daemon might not restore a connection to a server whose connection was broken.

DATA_CONVERSION

Explanation: Cannot perform codeset translation for character data or wide character data. This results from a failure of the translation utilities. It can occur if the process is using a non-standard codeset that does not map to an OSF codeset. Also, it can occur if there is no common codeset between the client and the server.

User Response: When using the translationEnabled configuration setting, ensure that the NLS-related configuration settings are correctly set. Verify that both the client and the server are using standard codesets and that there is some codeset supported by both the client and the server. Refer to the codeset reference section of this document.

0x49420051 SOMDERROR_DataConversion

Details: This is the generic data conversion minor code.

INITIALIZE

Explanation:A configuration or installation error is causing a problem during initialization.

User Response: See the details of the following minor codes.

0x49420015 SOMDERROR_CouldNotLoadLibrary

Details: Client initialization cannot load a required library. Check the activity log for more information.

0x49420035 SOMDERROR_InvalidProtocolInformation

Details: The configuration of the communications protocol is incorrect. Supported communication protocols are TCP/IP and IPC. Verify that at

least one valid communications protocol image is configured. Also, verify that for each communications protocol configured, the csProfileTag and portNumber are set and that the portNumber is not use by another process on the system. (The portNumber is the port on which the location service daemon or server listens for requests.) The csProfileTag and portNumber settings must be unique.

0x49420036 SOMDERROR_SOMDDAlreadyRunning

Details: The location service daemon cannot begin listening because another process is using the port number. Probably another instance of the process is running already. If no other location service daemon is running, reconfigure the location service daemon to listen on a different port number.

0x49420037 SOMDERROR_InvalidConfigSetting

Details: A configuration setting or environment variable was not properly set. An error log entry indicates which configuration setting or environment variable is not properly set. If the reported variable is WASORBTOP, verify that the product was properly installed (Set WASORBTOP to the directory where the product was installed.)

0x49420046 SOMDERROR_ServerAlreadyExists

Details. A server cannot register with the location service daemon during CORBA::BOA::impl_is_ready. Another server might be registered already with the location service daemon under the server UUID. Only one instance of a particular server can run on a given host. Terminate the duplicate server process. If no duplicate server process is running, restart the location service daemon.

0x4942004E SOMDERROR_SOMDDNotRunning

Details: A server cannot register with the location service daemon (in CORBA::BOA::impl_is_ready) because it cannot contact the daemon. It is possible that the daemon is not running or the daemon is running on a port number that is different from what the server expected. Verify that the location service daemon is running and is listening on the correct port.

INTERNAL

Explanation: An internal error condition was detected.

User Response: See the details of the following minor codes. Report the occurrence to technical support.

0x49420034 SOMDERROR_NotImplemented

Details: The invoked operation is not supported in the product or is not valid on the target object. Check that the operation being invoked and the target object run-time type are compatible. Refer to the documentation for the operation for information about restrictions.

0x4942004B SOMDERROR_Internal

Details: Report the problem to technical support.

0x4942004F SOMDERROR_ServerInterrupt

Details: The server has been shut down by an invocation of the interrupt_server() method.

0x4F4D0001

Details: A Portable Interceptor operation cannot find an object key on the target object.

0x4F4D0002

Details: A Portable Interceptor operation cannot find an object key on the target most derived object.

INTF_REPOS

Explanation: An Interface Repository operation has encountered a problem or error.

User Response: See the details of the following minor codes.

0x49420052 SOMDERROR_IRIncoherent

Details: An Interface Repository object references another named Interface Repository object that no longer resides in the IR database. Delete and rebuild the WASORBIR database to correct the problem.

0x49420053 SOMDERROR_IRInternal

Details: An internal programming or database error has occurred. Delete and rebuild the WASORBIR database to correct the problem.

0x49420054 SOMDERROR_IRDuplicateEntry

Details. There was an attempt to create an Interface Repository object when one already exists in the Interface Repository with either the same CORBA::RepositoryId or the same name within that container. Change the ID (CORBA::RepositoryId) parameter that is passed to the 'create_xxxx' operation or change the ID (CORBA::RepositoryId) value of the object already in the IR that is causing the duplicate entry error using the ID write operation. Or, change the name of one of the two conflicting objects within that container.

0x49420055 SOMDERROR_IREntryNotFound

Details: This error might occur when the client is attempting to use Dynamic Invocation Interface (DII) with interfaces that are not accessible in the Interface Repository. Check that the client has access to the Interface Repository and that the interfaces being used are defined in the Interface Repository.

0x49420056 SOMDERROR_IRCannotConnect

Details: Cannot find or access the Interface Repository database. This can occur during a call to resolve_initial_references (with an input string of InterfaceRepository). Verify that the Interface Repository database exists and is properly configured. Also, verify that the directory or file permissions associated with the Interface Repository database allow access by the user receiving the exception. If the database is remote, check that the ID and password are properly configured for that database using configuration properties com.ibm.CORBA.irUserid and com.ibm.CORBA.irPassword.

0x49420057 SOMDERROR_IRNameReUse

Details. Another thread or process is updating the needed portion of the Interface Repository database. Retry the Interface Repository operation that generated the exception at a later time.

INV_OBJREF

Explanation: An invalid object reference was used. For example, if a client

uses a reference to an object that no longer exists or cannot be located in the specified server, this error is sent from the server to the client. This error can occur in a client process if an invalid string is passed to CORBA::ORB::string_to_object. It occurs in a server if CORBA::BOA::create is called with input ReferenceData that does not map to any known exportable object residing in that server. The error occurs if CORBA::BOA::get_id is invoked on a nil object reference or on an object reference that has no associated ReferenceData in that server. Also, this error occurs if a server attempts to export an object reference that has no associated ReferenceData in that server or if a client attempts to pass a local object as a parameter on a remote method invocation.

User Response: In a client process, verify that the object that the object reference refers to still exists. Verify that strings passed to CORBA::ORB::string_to_object have not been corrupted or truncated. There is no maximum length for an object reference string. Some lengths, however, are larger than others. Verify that servers do not attempt to export objects that are not handled by the application adaptor of the server.

0x49420040 SOMDERROR_BadObjref

Details: This is the generic invalid object reference minor code.

MARSHAL

Explanation: An error has occurred when trying to marshall or demarshall method parameters or return results as part of a remote invocation. This can occur when demarshalling an inout sequence if the length of the incoming sequence is greater than the original sequence maximum. It alsocan occur if methods are not invoked on the ServerRequest object in the correct order when you use the Dynamic Skeleton Interface (DSI).

User Response: Verify that inout sequences do not exceed the sequence maximum. If using the DSI, verify that operations are invoked in the correct order on the ServerRequest object.

0x4942003C SOMDERROR_MarshalingError

Details: This is the generic marshal error minor code.

0x4F4D0001 SOMDERROR_MarshalingError

Details: Unable to locate or use the appropriate Value Factory.

NO_MEMORY

Explanation: A memory allocation failed

User Response: Verify that the process does not have a memory leak. Increase system resources.

0x4942000A SOMDERROR_NoMemory

Details: This is the generic no memory minor code.

NO_RESOURCES

Explanation: There is a system resources problem. A needed resource is unavailable.

User Response: See the details of the following minor codes.

0x4942000D SOMDERROR_CouldNotStartThread

Details: Cannot start a thread. Check the log for more information. Increase system resources.

0x49420014 SOMDERROR_CouldNotStartProcess

Details. The location service daemon cannot start a server process. Check the log for more information.

0x4F4D0001

Details: A Portable Interceptor operation was invoked that is not supported.

NO_RESPONSE

Explanation: Some process has timed out.

User Response: See the details of the following minor codes.

0x4942003B SOMDERROR_NoMessages

Details: No request messages were pending in a server process when the server invoked CORBA::BOA::execute_next_request or CORBA::BOA::execute_request_loop with the CORBA::BOA::SOMD_NO_WAIT flag. Wait for a request to become available or use the CORBA::BOA::SOMD_WAIT flag to call CORBA::BOA::execute_next_request or CORBA::BOA::execute_request_loop.

0x4942003D SOMDERROR_CommTimeOut

Details. A process has timed out while waiting for a response from another process. Typically, a client receives this error when the server has terminated or is hanging due to an application error. Verify that the other process is still active. To increase the timeout period, change the requestTimeout property in the configuration.

Note: Setting the requestTimeout property to zero results in an infinite timeout.

0x4942005A : SOMDERROR_Server_Registration_Timeout

Details: A server has been started by the location service daemon, but that server has not registered within the timeout period.

OBJECT_NOT_EXIST

Explanation: A locate request failed to find the requested object or the object's server is not running.

User Response: Verify that the correct object reference is being used and the correct server is running.

0x4942005C SOMDERROR_ObjectNotExist

Details: This is the generic object not exist minor code.

PERSIST_STORE

Explanation: There is a Problem with the Implementation Repository.

User Response: See the details of the following minor codes.

0x49420043 SOMDERROR_ImplRepIO

Details: Cannot access the Implementation Repository database. Verify that the Implementation Repository was correctly created and configured. Each host machine must have its own Implementation Repository.

0x49420044 SOMDERROR_EntryNotFound

Details: Cannot find an entry in the Implementation Repository when

attempting to delete, update, or locate it. Verify that the specified server alias or UUID matches a server that was previously registered in the Implementation Repository.

0x4942004A SOMDERROR_DuplicateEntry

Details: The application attempted to add a duplicate entry to the Implementation Repository or attempted to update the server alias of an existing entry using a name that is not unique. The server alias does not need to be unique throughout the network, but it must be unique in each Implementation Repository. Verify that the server UUID and server alias of the ImplementationDef to be added or updated in the Implementation Repository are unique.

SQL_INFORMATION

Explanation:A SQL error occurred.

User Response: Report the problem to technical support.

0x49420058 SOMDERROR_IRSQLInformation

Details: An Interface Repository operation has caught a SQL error.

UNKNOWN

Explanation: An unknown error occurred.

User Response: Report the problem to technical support.

0x49420041 SOMDERROR_Unknown

Details: An unexpected error occurred during an operation.

0x4F4D0001

Details: An unknown user exception was detected in the Portable Interceptor.

Interface Definition Language (IDL)

The interface to a class of objects contains the information that a caller must know to use an object. Specifically, it contains the names of its attributes and the signatures of its methods. In the CORBA programming model, the Object Management Group (OMG) Interface Definition Language (IDL) is the formal language used to define object interfaces independent of the programming language used to implement those methods.

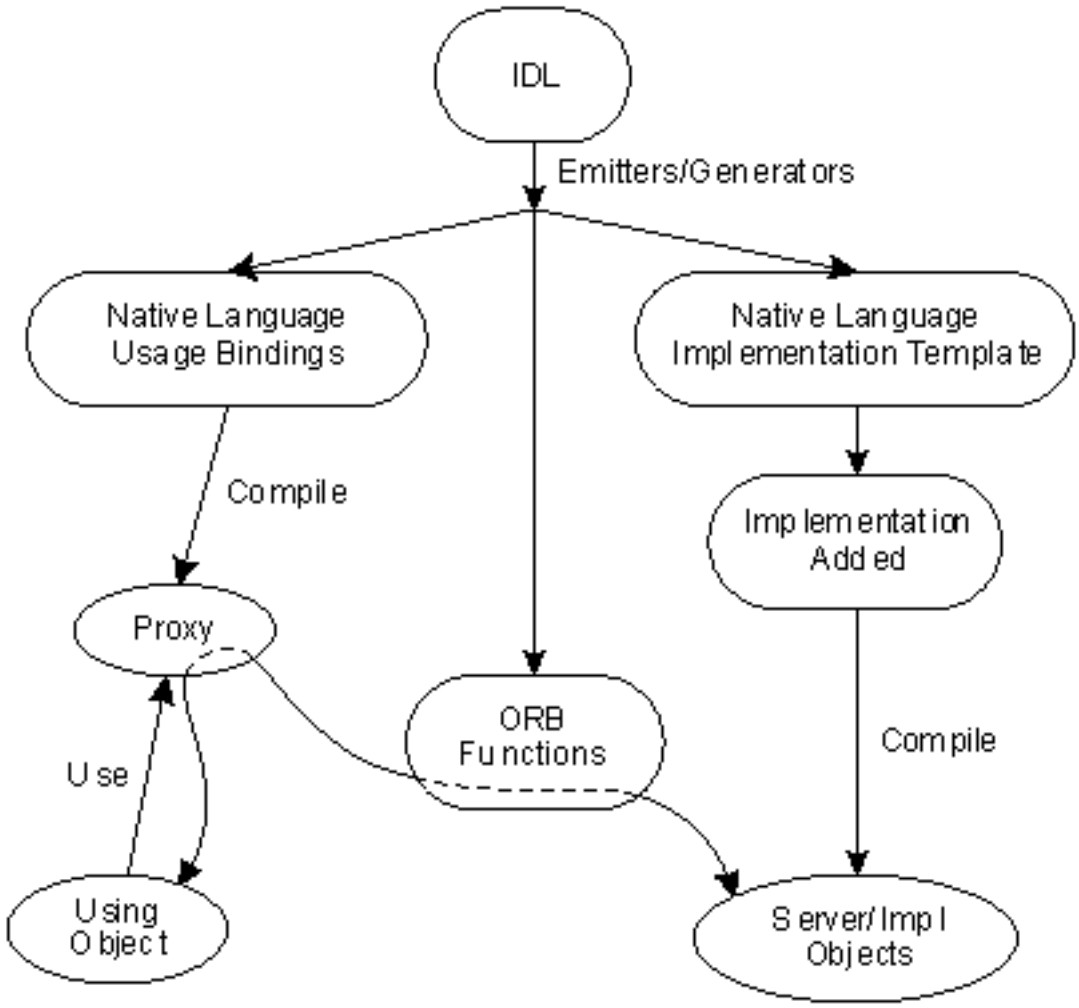


Figure 5. IDL, usage and implementation

This is an overview of the relationship between IDL and application development languages. Object providers use IDL to define the interfaces to their objects. The IDL can be directly defined by the object provider or can be produced transparently to the user in application development tools. Code emitters and generators produce the following elements:

- A usage binding that provides a native, client language rendering of the IDL (for example, as a C++ class or Java interface). The usage binding also is used to generate a client stub object that, through delegation, maps the interface onto the server object providing the implementation.
- An implementation template that provides a native, server language class template into which method behavior can be inserted (for example, by editing the file and adding source code). The implementation of a class of objects (that is, the procedures that implement operations and the variables used to store an object's state) is written in the implementor's preferred programming language (for example, C++ or Java).
- Implementation objects such as skeletons and stubs also can be emitted and compiled if the client and server are in different processes or in different languages. These implementation objects provide the functions needed to make interlanguage calls and remote method execution.

The IDL compiler takes an IDL file as input and produces the usage binding files that make it convenient to implement and use objects that support the defined interface within a particular programming language.

For an enterprise bean, you can create the IDL files from the bean's interface and home classes.

IDL name scoping

An Interface Definition List (IDL) file, together with the contents of any files referenced by #include statements, forms a naming scope. Definitions that do not appear inside a scope are part of the global scope. The following kinds of definitions introduce nested scopes: module, interface, valuetype, struct, union, operation, and exception. An identifier can be defined one time in a particular scope. Identifiers can be redefined in nested scopes, except that an identifier cannot be the same as the identifier defining the scope. For example, an interface X cannot contain an identifier named X. It is possible to define multiple modules with the same identifier. The first declaration of the module defines the module. Subsequent declarations of the module with the same identifier reopens the module and hence its scope, which allows additional definitions to be added to it.

A particular definition may be referenced using an unqualified name (simply the name of the identifier) or a qualified name. An unqualified name reference is resolved by successively searching the enclosing scopes. If an unqualified name is referenced in a scope, that identifier cannot be redefined within the same scope.

A qualified name reference is a series of identifiers with intervening double colons (::). The last identifier is the name of the definition and the leftmost identifiers indicate the scope of the definition. Qualified names have the following form:

```
scope-name::identifier
```

For example, the method name mymethod defined within the interface Test of module M1 has the fully qualified name:

```
M1::Test::mymethod
```

A qualified name is resolved by first resolving the *scope-name* to a particular scope, S and then locating the definition of the *identifier* within that scope. Scopes that enclose the scope S are not searched.

Qualified names also can have the following form:

```
::identifier
```

These names are resolved by locating the definition of *identifier* within the outermost name scope.

Every name defined in an IDL specification is given a global name and is constructed as follows:

- Before the IDL compiler scans the IDL file, the name of the current root and the name of the current scope are empty. When each module is encountered, the string of two colons (::) and the module name are appended to the name of the current root. At the end of the module, they are removed.
- When each interface, valuetype, struct, union, or exception definition is encountered, the string of two colons (::) and the associated name are appended to the name of the current scope. At the end of the definition, they are removed. While parameters of an operation declaration are processed, a new unnamed scope is entered so that parameter names can duplicate other identifiers.

- The global name of an IDL definition is then the concatenation of the current root, the current scope, two colons (::), and the local name for the definition.

The names of types, constants, and exceptions defined by base interfaces and valuetypes are accessible in a derived interface. References to these names must be unambiguous. Ambiguities can be resolved by using a scoped name (prefacing the name with the name of the interface that defines it and the two colons (::), as in *base-interface::identifier*). Scope names also can be used to refer to a constant, type, or an exception name defined by a base interface or valuetype but redefined by a derived interface or valuetype.

IDL interface declarations

The Interface Definition Language (IDL) specification for a class of objects must contain a declaration of the interface these objects support.

IDL interfaces and types should be enclosed inside a module scope. IDL declared outside of a module scope takes up namespace in the global IDL namespace and risks having name collisions with names declared by other IDL developers. For more information, see "IDL name scoping" on page 150.

When objects are implemented using classes, the interface name also is used as a class name. In addition to the interface name and its base interface names, an interface indicates new methods (operations) and any constants, type definitions, and exception structures that the interface exports.

An interface declaration has the following syntax:

```
interface  interface-name [: base-interface1, base-interface2, ...]
{
   constant declarations
   type declarations
   exception declarations
   attribute declarations
   operation declarations
};
```

All of the declaration elements are optional and their order usually is not significant. However you must bear in mind the following considerations:

- Interface names must be declared before they are referenced.
- Types, constants, exceptions, and interface declarations must be defined before they are referenced (as in C or C++).
- Using one declaration can mandate another and determine the order in which they are declared. For example, if an operation raises an exception, the exception must be declared and must come before the operation in the list.

The base-interface names specify the interfaces from which *interface-name* is derived. Parent-interface names are required only for the immediate base interfaces. Each base interface must have its own IDL specification that must be referenced by a #include in the IDL file. A base interface cannot be named more than one time in the interface statement header.

The following topics describe the declaration elements that can be specified within the body of an interface declaration:

- "IDL constant declarations" on page 152
- "IDL type declarations" on page 152
- "IDL exception declarations" on page 155

- "IDL attribute declarations" on page 156
- "IDL operation declarations" on page 157

IDL constant declarations: Constants are declared in Interface Declaration Language (IDL) just as in C++ except that the type of the constant must be a valid IDL type. IDL constant declarations take the following form:

const *const-type identifier=constant-expression*;

The *const-type* must be a valid IDL integer, char, wchar, boolean, floating point, string, wstring, octet, or enum type. The *identifier* is the name of the constant being defined. The *constant-expression* is a constant expression as in C or C++ and can include the following that are usual in the C and C++ programming languages:

- unary and binary operators (|, ^, &, >>, <<, +, -, *, /, %, ~~)
- parentheses for controlling operator precedence
- literal values (integer, string, character, and floating point)
- boolean literal values TRUE and FALSE.

IDL type declarations: IDL specifications can include the following type declarations as in C++ with the restrictions and extensions described in these topics:

- "IDL integer types"
- "IDL floating point types" on page 153
- "IDL character types" on page 153
- "IDL boolean type" on page 153
- "IDL octet type" on page 153
- "IDL any type" on page 153
- "IDL constructed types" on page 153
- "IDL template types" on page 154
- "IDL arrays" on page 155
- "IDL object types" on page 155

The form of a type declaration within the body of an interface declaration is described in IDL interface declarations.

IDL integer types: The Interface Definition Language (IDL) supports the following integer types only, which represent the corresponding value ranges:

Table 1. Supported IDL integer types and their value ranges

Integral type	Value range
short	-2^{15} through $(2^{15})-1$
long	-2^{31} through $(2^{31})-1$
long long	-2^{63} through (263)-1
unsigned short	0 through $(2^{16})-1$
unsigned long	0 through $(2^{32})-1$
unsigned long long	0 through $(2^{64})-1$

IDL floating point types: The Interface Definition Language (IDL) supports the float and double floating-point types. The float type represents the IEEE single-precision floating-point numbers. The double floating-point type represents the IEEE double-precision floating-point numbers.

Because returning floats and doubles by value might not be compatible across all Microsoft Windows compilers, client programs should return floats and doubles by reference.

IDL character types: IDL supports a char type, which represents an 8-bit quantity. The ISO Latin-1 (8859.1) character set defines the meaning and representation of graphic characters. The meaning and representation of null and formatting characters is the numerical value of the character as defined in the ASCII (ISO 646) standard. Unlike C and C++, type char cannot be qualified as signed or unsigned. (The octet type can be used in place of unsigned char.)

IDL boolean type: The Interface Definition Language (IDL) supports a boolean type for data items that can take only the values zero (FALSE) and one (TRUE).

IDL octet type: The Interface Definition Language (IDL) supports an octet type, which is an 8-bit quantity guaranteed not to undergo conversion when transmitted between a client and server process. The octet type can be used in place of the unsigned char type. For more information on the unsigned type, see "IDL character types".

IDL any type: The Interface Definition Language (IDL) supports an any type, which permits the specification of values of any IDL type. Conceptually, an any consists of a value and a TypeCode that represents the type of the value. The TypeCode class provides functions for obtaining information about an IDL type.

IDL constructed types: In addition to the basic types, the Interface Definition Language (IDL) also supports three constructed types:

- Structure (struct)
- Union (union)
- Enumeration (enum)

The *structure* and *enumeration* types are specified in the IDL just as they are in C and C++. However, they have the following restrictions:

- Recursive type specifications are allowed only through the use of the sequence template type.
- Structures and enumerations in IDL must be tagged. For example, struct { int a; ... } is an inappropiate type specification because the tag is missing. The tag introduces a new type name.
- Structure and enumeration type definitions need not be part of a typedef statement. Furthermore, if they are part of a typedef statement, the tag of the struct must differ from the type name being defined by the typedef. For example, the following are valid IDL struct and enum definitions:

```
struct myStruct {
               long x;
               double y;
            };
            /* defines type name myStruct */
            enum colors { red, white, blue };
            /* defines type name colors */
```

The following IDL definitions are not valid:

```
typedef struct myStruct {
               /*  NOT VALID  */
               long x;
               /*  Tag myStruct is the same */
               double y;
               /*  as the type name below;  */
            } myStruct;
               /*  myStruct has been redefined */
            typedef enum colors { red, white, blue } colors;
               /* NOT VALID */
```

The IDL *union* type is a cross between the C union and switch statements. This type is specified in IDL just as it is in C and C++, with the restriction that discriminated unions in IDL must be tagged. The syntax of a union type declaration is as follows:

```
union identifier switch
(switch-type) { case+ }
```

- The *identifier* following the union keyword defines a new legal type. (Union types also can be named using a typedef declaration.)
- The *switch-type* specifies an integral, character, boolean, or enumeration type, or the name of a previously defined integral, boolean, character or enumeration type.
- Each *case* of the union is specified with the following syntax:

  ```
  case-label+ type-spec declarator;
  ```

  where:
  - Each *case-label* has the following form:

    ```
    case const-expr:
    ```

    default: The *const-expr* is a constant expression that must match or be automatically castable to the *switch-type*. A default case can appear no more than once.
  - *type-spec* is any valid type specification.
  - *declarator* is an identifier or an array declarator (such as, foo[3][5]).

Note: A deviation from CORBA specifications exists; there is no support of longlong discriminators in unions.

IDL template types: The Interface Definition Language (IDL) defines two template types not found in C and C++: sequences and strings. A sequence is a one-dimensional array with two characteristics: an optional maximum size (specified at compile time) and a length (determined at run time). Sequences permit passing unbounded arrays between objects. Sequences are specified as follows:

```
sequence simple-type [, positive-integer-const]
```

where *simple-type* specifies any valid IDL typeand the optional *positive-integer-const* is a constant expression that specifies the maximum size of the sequence (as a positive integer).

A string is similar to a sequence of type char. It can contain all possible 8-bit quantities except NULL. Strings are specified as follows:

```
string [ positive-integer-const ]
```

where the *optional positive-integer-const* is a constant expression that specifies the maximum size of the string (as a positive integer, which does not include the extra byte to hold a NULL as required in C or C++).

Since CORBA gives no specific rules on how to process blanks contained within strings, IBM WebSphere Application Server treats `"ABC"` and `"ABC  "` as referring to different managed objects. If you do not want blanks to be treated as significant pre-process your code to either remove trailing blanks or add trailing blanks to some fixed string length.

IDL arrays: Multidimensional, fixed-size arrays can be declared in the Interface Definition Language (IDL) as follows:

```
identifier { [ positive-integer-const ] }+
```

where the *positive-integer-const* is a constant expression that specifies the array size, in each dimension, as a positive integer. The array size is fixed at compile time.

IDL object types: The name of the interface to a class of objects can be used as a type name. For example, if an Interface Definition Language (IDL) specification includes an interface declaration for a class (of objects) C1, then C1 can be used as a type name within that IDL specification. When used as a type, an interface name indicates a reference to an object that supports that interface. An interface name can be used as the type of an operation argument, as an operation return type, or as a member of a constructed type (a struct, union, or enum). In all cases, the use of an interface name indicates a reference to (instead of an instance) an object that supports that interface.

IDL exception declarations: Interface Definition Language (IDL) specifications can include exception declarations, which define data structures to be returned when an exception occurs during the execution of an operation. A name is associated with each type of exception. Optionally, a struct-like data structure for holding error information also can be associated with an exception. Exceptions are declared as follows:

```
exception identifier
  {
     member*
  };
```

The *identifier* is the name of the exception and each *member* has the following form:

```
type-spec declarators ;
```

The *type-spec* is a valid IDL type specification and *declarators* is a list of identifiers or array declarators, delimited by commas. The members of an exception structure must contain information to help the caller understand the nature of the error. The exception declaration can be treated like a struct definition. Whatever you can access in an IDL struct, you can access in an IDL exception. Unlike a struct, an exception can be empty which means that the exception is just identified by its name.

If an exception is returned as the outcome of an operation, the exception *identifier* indicates which exception occurred. The values of the members of the exception provide additional information specific to the exception. The article, "IDL operation declarations" on page 157 describes how to indicate that a particular operation can raise a particular exception.

The following is an example showing the declaration of a BAD_FLAG exception:

```
exception BAD_FLAG
{
   long ErrCode; char Reason[80];
};
```

In addition to user-defined exceptions, there are several predefined exceptions for system run-time errors. The standard exceptions as prescribed by CORBA are subclasses of CORBA::SystemException. These exceptions correspond to standard run-time errors that can occur during the execution of any operation (regardless of the list of exceptions listed in the operation's IDL specification).

Each of the standard exceptions has the same structure: an error code (to designate the subcategory of the exception) and a completion status code. For example, the NO_MEMORY standard exception has the following definition:

```
enum completion_status
{
   COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};
exception NO_MEMORY
{
   unsigned long minor;
     completion_status completed;
};
```

The "completion_status" value indicates whether the operation was never initiated (COMPLETED_NO), if the operation completed its execution prior to the exception (COMPLETED_YES) , or if the operation's completion status is indeterminate (COMPLETED_MAYBE).

IDL attribute declarations: Declaring an attribute as part of an interface is equivalent to declaring one or two accessor operations: one to retrieve the value of the attribute (a get or read operation) and (unless the attribute specifies read-only) one to set the value of the attribute (a set or write operation).

Attributes are declared as follows:

```
[ readonly ] attribute type-spec declarators;
```

where:

- *type-spec* specifies any valid Interface Definition Language (IDL) type (except a sequence).
- *declarators* is a list of identifiers, delimited by commas. An array declarator cannot be used directly when declaring an attribute. The type of an attribute can be a user-defined type that is also an array. Although the type of an attribute cannot be a sequence, it can be a user-defined type that is a sequence. The optional readonly keyword specifies that the value of the attribute can be accessed but not modified. (In other words, a readonly attribute has no set operation.) The following is an example of attribute declarations, which are specified within the body of an interface statement:

  ```
  interface Goodbye: Hello
  {
     void  sayBye();
     attribute short xpos;
     attribute char c1, c2;
     readonly attribute float xyz;
  };
  ```

Attributes are inherited from base interfaces. An inherited attribute name cannot be redefined to be a different type.

IDL operation declarations: Operation declarations define the interface of each operation introduced by the interface. An Interface Definition Language (IDL) operation is typically implemented by a method in the implementation programming language. Hence, the terms operation and method are often used interchangeably. An operation declaration is similar to a C++ virtual function definition:

```
[ oneway ] type-spec identifier ( parameter-list ) [ raises-expr [ context-expr ] ;
```

where:

- *identifier* is the name of the operation.
- *type-spec* is any valid IDL type, except a sequence or the keyword void that indicates that the operation returns no value. (Although the return type cannot be a sequence, it can be a user-defined type that is a sequence.) Unlike C and C++ procedures, operations that do not return a result must specify void as their return type.

The remaining syntax of an operation declaration is elaborated in the following topics:

- "IDL operation declarations: "oneway" keyword"
- "IDL operation declarations: parameter list"
- "IDL operation declarations: "raises" expression" on page 158
- "IDL operation declarations: "context" expression" on page 158

IDL operation declarations: "oneway" keyword: For an overview of IDL operation declarations, see "IDL operation declarations".

The optional oneway keyword specifies that when a caller invokes the operation, no reply is expected or received. The invocation semantics of a oneway operation are best-effort, which does not guarantee delivery of the call. Best-effort implies that the operation is invoked "at most once". A oneway operation must not have any output parameters and must have a return type of void. A oneway operation also must not include a raises expression.

If the oneway keyword is not specified, then the operation has *at-most-once* invocation semantics if an exception is raised and it has *exactly-once* semantics if the operation succeeds. This means that an operation that raises an exception is implemented zero or one times and an operation that succeeds is implemented exactly once.

IDL operation declarations: parameter list: For an overview of the Interface Declaration Language (IDL) operation declarations, see "IDL operation declarations".

The parameter-list contains zero or more parameter declarations for the operation and is delimited by commas. (The target object for the operation is not explicitly specified as an operation parameter in IDL.) If there are no explicit parameters, the syntax "( )" must be used, rather than "(void)". A parameter declaration has the following syntax:

```
{ in | out | inout } type-spec declarator
```

where *type-spec* is any valid IDL type (except a sequence) and declarator is an identifier or an array declarator. Although the type of a parameter cannot be a sequence, it can be a user-defined type that is a sequence.

The required in, out, or inout directional attribute indicates whether the parameter is to be passed from caller to callee (in), from callee to caller (out), or in both directions (inout). The following are examples of valid operation declarations:

```
short meth1(in char c, out float f);
   oneway void meth2(in char c);
   float meth3();
```

An operation's implementation should not modify an *in* parameter. If a change must be made by the implementation, the implementation should copy the parameter and modify the copy only.

If an operation raises an exception, the values of the return result and the values of the out and inout parameters (if any) are undefined.

IDL operation declarations: "raises" expression: For an overview of the Interface Definition Language (IDL) operation declarations, see "IDL operation declarations" on page 157.

The optional raises expression in an IDL operation declaration indicates which exceptions the operation can raise. A raises expression is specified as follows:

```
raises ( identifier1, identifier2, ... )
```

where each identifier is the name of a previously defined exception. In addition to the exceptions listed in the raises expression, an operation also can signal any of the standard exceptions. Standard exceptions, however, should not appear in a raises expression. If no raises expression is given, then an operation can raise the standard exceptions only. "IDL exception declarations" on page 155 contains further information on defining exceptions and the list of standard exceptions.

IDL operation declarations: "context" expression: For an overview of the Interface Declarations Language (IDL) operation declarations, see "IDL operation declarations" on page 157.

The optional context expression (context-expr) in an operation declaration indicates which elements of the caller's context the operation's implementation can consult. A context expression is specified as follows:

```
context ( identifier1, identifier2, ... )
```

where each *identifier* is a string literal made up of alphanumeric characters, periods, underscores and asterisks. The first character must be alphabetic and an asterisk can only appear as the last character, where it serves as a wildcard matching any characters. If convenient, identifiers can consist of period-separated , valid identifier names, but that form is optional.

The context is a special object that is specified by the CORBA standard. It contains a property list. The property list contains a set of property name and string value pairs that the caller can use to store information about its environment, which operations can find useful. It is used similar to environment variables. The property list is passed as an additional parameter to operations that are defined as context-sensitive in the IDL.

The context expression of an operation declaration in IDL specifies which property names the operation uses. If these properties are present in the context object supplied by the caller, they are passed to the object implementation, which can access them through the interface of the context object.

The argument that is passed to the operation having a context expression is a Context object, not the names of the properties. The caller must create a context object and use its interface to set the context properties. The caller then passes the context object in the operation invocation. The CORBA standard allows properties, in addition to those in the context expression, to be passed in the context object.

Multiple IDL interfaces

A single Interface Definition Language (IDL) file can define multiple interfaces. When a file defines two or more interfaces that reference one another, forward declarations are used to declare the name of an interface before it is defined. This is done as follows:

```
interface interfaceName ;
```

The actual definition of the interface for *interfaceName* must appear later in the same IDL file.

If multiple interfaces are defined in the same IDL file, they can be grouped into modules by using the following syntax:

```
module moduleName { definition+ };
```

where each definition is a type declaration, constant declaration, exception declaration, interface statement, or nested module statement. Modules are used to scope identifiers.

Alternatively, multiple interfaces can be defined in a single IDL file without using a module to group the interfaces. Whether or not a module is used for grouping multiple interfaces, the language bindings produced from the IDL file include support for all of the defined interfaces.

IDL include directives

If your interface declaration refers to a parent interface or uses some other referenced types, the Interface Declaration Language (IDL) file must contain #include statements that tell the IDL compiler where to find the referenced interface definitions (the IDL files).

If your interface declaration refers to IDL types (defined by the CORBA specification) that are not IDL reserved words, then the IDL file should contain an #include statement for the orb.idl file.

As in C and C++, if an #include statement specifies a file name that is enclosed in angle brackets ([ ]), the search for the file begins in system-specific locations. If the file name is enclosed in double quotation marks (""), the search for the file begins in the current working directory, before searching the system-specific locations.

For information on other preprocessor directives that can be used in IDL, see "IDL pragma directives".

IDL pragma directives

For information on preprocessor directives that can be used in Interface Definition Language (IDL), see "IDL include directives".

localonly pragma

This pragma supports the generation of bindings for objects that are known to be local (not distributed). This pragma can occur at any point in the IDL file following the definition or forward declaration of the designated interface.

The syntax is:

```
#pragma meta interface-name localonly
```

The IDL interface identified by *interface-name* is treated by generated bindings as strictly local to the caller's process. No calls to the CORBA Object Request Broker (ORB) occur when invoking the operations defined in this interface. *interface-name* can be a simple name of an interface in the current scope or a fully- or partially qualified interface name. The interface must be previously defined or forward declared when the pragma statement is encountered.

localonly abstract pragma

This pragma is like the localonly pragma, but it signifies an abstract function that cannot be instantiated. These types of interfaces are used to just define interfaces.

The syntax is:

```
#pragma meta interface-name localonly abstract
```

cpponly pragma

This pragma suppresses the generation of IOM interlanguage bindings.

The syntax is:

```
#pragma meta interface_name cpponly
```

In the default case, without this pragma, two sets of bindings are produced:

- The standard CORBA C++ bindings suitable for use with the ORB component.
- IOM bindings suitable for interlanguage interaction.

Without this pragma, only the standard CORBA C++ bindings are produced.

init pragma

This pragma specifies a function used to initialize newly created objects.

The syntax is:

```
#pragma meta method-name init
```

This pragma allows the IDL to specify the name of a function used to initialize the newly created method. When this pragma is not used, the emitters produce a _create() function that does not take parameters and does no initialization after the new object is created.

For example, if the IDL contains:

```
interface A
{
   void initFunction(int);
};
#pragma meta A::initFunction init
```

the C++ class A that implements interface A has a _create() function that takes an int parameter (because initFunction takes an int). Also, the code inside _create(int) creates a new instance of class A, calls initFunction(int) on the newly created object, and passes along its int parameter.

ID pragma

This CORBA-defined pragma overrides the default RepositoryID for an IDL entity.

The syntax is:

```
#pragma ID scoped-name literal-string
```

This sets the RepositoryID of *scoped-name* to *literal-string* instead of the default Repository ID.

Prefix pragma

This CORBA-defined pragma sets the RepositoryID prefix.

The syntax is:

```
#pragma prefix string
```

This sets the current prefix used in generating OMG IDL format RepositoryIDs. The specified prefix applies to RepositoryIDs generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered.

version pragma

This CORBA-defined pragma sets the RepositoryID version number.

The syntax is:

```
#pragma version scoped-name major.minor
```

This uses the *major.minor* as the version number for RepositoryID of the *scoped-name*.

idlc command (IDL compiler)

The Interface Definition Language Compiler (idlc) command creates usage and implementation bindings for interfaces described in IDL files.

Use this command to compile one or more files containing CORBA 2.3-compliant IDL statements and (optionally) to emit generated language bindings appropriate to each named input file.

Syntax

```
idlc [options] filename-list
```

where *options* are described in this article and *filename-list* is a list of one or more IDL files.

Each file name specified in *filename-list* can be specified with or without a file name extension. If no file name extension is supplied, it is assumed to be ″.idl″.

When all of the specified input files are compiled, the idlc command returns a value of zero if no errors are detected. Otherwise, a non-zero value is returned.

Options

Options for the idlc command are preceeded with a dash (-) character and can be specified individually or run together. For example, `-p -v -V` or `-pvV` are acceptable.

Some options accept an argument. Where several options have the same argument, these options also can be specified individually or run together. For example, `-p -m tie` or `-pm tie` are acceptable.

The space between the option and its argument is optional. For example, either `-mtie` or `-m tie` are acceptable.

All options are case-sensitive, even on platforms where file names are not case-sensitive.

The following table describes each available option:

idlc command options

Option	Description
-d *directory-name*	Specifies the directory in which to place emitted output files and directories. If none is specified, the default is the current directory.
-V	Shows the version number of the idlc command.
-v	Specifies verbose mode. This shows all of the internal commands (and their arguments) issued by the idlc command.
-? (or -h)	Writes a brief description of the idlc command syntax to standard output.
-D *define-expression*	Predefines a preprocessor variable for the IDL compiler.
-I *include-directory*	Adds a directory to the list of directories used by the IDL compiler to find #include files. In addition to the -I option, the IDLC_INCLUDE environment variable can be used to specify a list with *include-directory* names separated by the PATH separator character.
-i *file-name*	Specifies the name of a file to be compiled that does not have the .idl extension. Do not add an implicit .idl suffix to the *file-name*.
-p	Used as a shorthand for -D__PRIVATE__.

Option	Description
-e (or -s) *emit-list*	Specifies a list of emitters to run. Emitters generate output files that contain language-specific usage and implementation bindings appropriate to each named input file. The rules used to generate the names of these output files are described in the article entitled, ″The idlc command: Emitted C++ filenames″. Each emitter in the list is separated from the others by a colon (:) or semicolon (;) character. Valid emitter names are: **hh** Produces C++ usage bindings. If no modifiers are present, bindings with support for remote cross-language operation are produced. The cpponly and localonly modifiers cause specialized bindings to be produced (see -m*name[=value]*). **sc** Produces a C++ skeleton (server-side bindings) for the basic object adapter of the Object Request Broker (ORB). If no modifiers are present, bindings with support for remote cross-language operation are produced. The cpponly and localonly modifiers cause specialized bindings to be produced (see -m*name[=value]*). **uc** Produces local implementations needed by the C++ usage bindings. If no modifiers are present, bindings with support for remote cross-language operation are produced. The cpponly and localonly modifiers cause specialized bindings to be produced (see -m*name[=value]*). **ih** Produces a C++ implementation header. **ic** Produces a template file for the C++ object implementation code.

Option	Description
-m *name[=value]*	Specifies an output modifier. A modifier can be given as a name or a name=value expression. The emitters are sensitive to the following modifiers: **LINKAGE=***value* Used to insert customized C++ linkage modifiers into the generated bindings. **notcconsts** Eliminates the generation of C++ TypeCode constants and overloaded any operators. **tie** Generates ″tie-style″ bindings that assume delegation rather than inheritance. **cpponly** Suppresses the production of cross-language bindings and produces standard CORBA C++ bindings suitable for use with a standalone ORB. cpponly affects the bindings produced by the hh, sc, and uc emitters. **localonly** Generates bindings that only can be used to access a local object for all of the most-derived interfaces in the IDL file. **IRforce** Causes the interface repository (IR) emitter to destroy objects already present in the IR with the same name as in the IDL being produced. **dllname=***value* Puts Microsoft Windows NT import, export, or both specifications into classes contained in the DLL named by *value*. **preInclude=*file-name*** Adds the line: `#include file-name` to the .hh file, just before the line that includes corba.h. **postInclude=***file-name* Adds the line: `#include file-name` just before the end of the .hh file.

Option	Description
-J	Passes options through to the Java interpreter used internally. For example: `-J"-mx32m"` sets the heap size for the interpreter to 32M.

Environment Variables

IDLC OPTIONS

Any of the idlc command options can also be specified in the environment by adding the option to the IDLC_OPTIONS environment variable. Options specified in the IDLC_OPTIONS environment variable are treated as if they were entered on the command line before any of the actual command line options. For example, the IDLC_OPTIONS might be:

```
IDLC_OPTIONS="-mcpponly -mdllname=mydll"
```

and the command line might be:

```
idlc -ehh idlfile
```

The result is the same as if the IDLC_OPTIONS variable is not set and the command line is:

```
idlc -mcpponly -mdllname=mydll -ehh idlfile
```

IDLC_EMIT

Emitters can also be specified in an *emit-list* held in the IDLC_EMIT environment variable. When you run the idlc command, it looks for emitters specified by the -e or -s options, and also looks in the IDLC_EMIT environment variable. If it cannot find an *emit-list* in either source, then only the syntax of the named files is checked and any errors are reported. When a compilation error (but not a warning) is detected for a particular input file, the emit phase for that file is skipped.

idlc command: Emitted C++ filenames: The names of the generated output files are derived from the file name of the corresponding Interface Definition Language (IDL) file. For a file named *filestem*.idl, the following list of output files can be emitted when the idlc command is run. The list contains the emitter and its corresponding output file name.

hh *filestem*.hh

sc *filestem*_S.cpp

uc *filestem*_C.cpp

ih *filestem*.ih

ic *filestem*_I.cpp

IDL-to-Java compiler

The IDL-to-Java compiler generates Java bindings for a given Interface Definition Language (IDL) file.

The command to invoke the IDL-to-Java code compiler has the general form:

```
idlj [options] source_IDL
```

where *source_IDL* is the name of a file that contains IDL definitions and [options] is any combination of the options in the following list. Options can appear in any order, but must precede the IDL file specification.

IDL-to-Java command options

Option	Description
-d *symbol*	Defines a symbol before compilation. This is equivalent to putting the line #define *symbol* in an IDL file. It is useful when you want to define a symbol for compilation that is not defined within the IDL file, for example to include debugging code in the bindings.
-emitAll	Emits all types, including those found in #include files. By default, only those types found in *idl file* are emitted. For more information, see "Emitting bindings for include files" on page 168.
-f*side*	Defines what bindings to emit. *side* is one of `client`, `server`, `all`, `serverTie`, and `allTie`. This assumes -fclient if the flag is not specified. For more information, see "Emitting client and server bindings"
-i *include_ path*	By default, the current directory is scanned for included files. This option adds another directory. For more information, see "Specifying alternative locations for include files" on page 167.
-keep	Preserves preexisiting bindings. The default is to generate all of the files without considering if they already exist. If the Java binding files do already exist, this option stops the compiler from overwriting them. This is useful if you have customized those files, which you should not do unless you are very comfortable with their contents.
-pkgPrefix *type package*	Wherever *type* is encountered, ensures it resides within *package* in all generated files. **Note:** *type* must be a fully qualified, Java-style name. For more information, see "Inserting package prefixes".
-td *target_directory*	By default, the compiler outputs bindings to the directory from which it was invoked (the current directory). To direct the output to another directory, specify the target directory immediately following the -td flag. The target directory can be absolute or relative.
-v, -verbose	Generates status messages so that you can track the progress of compilation. If this option is not selected, messages are not outputed unless there are errors.

Option	Description
-version	Displays the build version of the IDL-to-Java compiler. Any additional options appearing on the command line are ignored. **Note:** Version information also appears within the bindings generated by the compiler.

Emitting client and server bindings: To generate the Java bindings for an Interface Definition Language (IDL) file named My.idl, set the current working directory to the one that contains My.idl and issue the following command:

```
idlj My.idl
```

This command generates client-side bindings only and is equivalent to:

```
idlj -fclient My.idl
```

Client-side bindings include all of the generated files except the skeleton. If you want to generate server-side bindings for My.idl, issue the following command:

```
idlj -fserver My.idl
```

This command generates all of the client-side bindings plus an inheritance-model skeleton (ImplBase). Currently, server-side bindings include all generated files, even the stub. Thus, the previous command is currently equivalent to each of the following:

```
idlj -fclient -fserver My.idl
idlj -fall My.idl
```

The compiler generates inheritance-model skeletons by default. Given an interface My defined in My.idl, the compiler generates Skeleton _MyImplBase.java. You provide the implementation for My, which must extend _MyImplBase.

Specifying alternative locations for include files: If My.idl included another Interface Definition List (IDL) file, MyOther.idl, the compiler assumes that MyOther.idl resides in the local directory. If it resides in directory */includes*, for example, invoke the compiler with the following command:

```
idlj -i /includes My.idl
```

If My.idl also included Another.idl that resided in /moreIncludes, then you would invoke the compiler as:

```
idlj -i /includes -i /moreIncludes My.idl
```

You can begin to see that if you have a number of places where included files can come from, the command becomes long and unmanageable. As a result, there is another means of indicating to the compiler where to search for included files. This technique is very similar to the idea of an environment variable. You must create a file called idl.config in a directory that is listed in your CLASSPATH. Inside of idl.config you must provide a line in the following form:

```
includes=/includes;/moreIncludes
```

The compiler takes the first version of the file it locates and reads in its includes list. In this example, the separator character between the two directories is a semicolon (;). It is a semicolon (;) on the Microsoft Windows NT platform and a colon (:) on the AIX® platform.

Note: Some platforms will fail when issuing a long command line. If the command line to invoke the compiler becomes too long, use the idl.config file.

Emitting bindings for include files: By default, only those interfaces, structs, and so on, that are defined in the Interface Definition Language (IDL) file on the command line have the Java bindings generated for them. The types defined in included files are not generated. For example, assume the following two IDL files:

```
My.idl

#include MyOther.idl
interface My
{
};

MyOther.idl

interface MyOther
{
};
```

The following command only generates bindings for types within My:

```
idlj My.idl
```

To generate bindings for all of the types in My.idl and all of the types in files that My.idl includes (in this example, MyOther.idl), use the following command:

```
idlj -emitAll My.idl
```

There is a caveat to the default rule. #include statements, which appear at the global scope, are treated as described. These #include statements can be thought of as import statements. #include statements that appear within some enclosing scope are treated as true #include statements. This means that the code within the included file is treated as if it appeared in the original file and, therefore, Java bindings are emitted for it. Here is an example:

```
My.idl

#include MyOther.idl
interface My
{
  #include Embedded.idl
};

MyOther.idl

interface MyOther
{
};

Embedded.idl

enum E {one, two, three};
```

Running the following command:

```
idlj My.idl
```

will generate the following list of Java files:

```
./MyHolder.java
./MyHelper.java
./_MyStub.java
./MyPackage
```

```
./MyPackage/EHolder.java
./MyPackage/EHelper.java
./MyPackage/E.java
./My.java
```

Note: MyOther.java is not generated because it is defined in an import-like #include. But E.java is generated because it is defined in a true #include. Also, because Embedded.idl was included within the scope of the interface My, it appears within the scope of My (that is, in MyPackage).

If the -emitAll flag is used in the previous example, all types in all of the included files are emitted.

Inserting package prefixes: This option has the following form:

`-pkgPrefix` *`type package`*

It ensures that wherever *type* is encountered, *type* resides within *package* in all of the generated files.

For example, let us suppose that a company called ABC has constructed the following Interface Definition Language (IDL) file:

`Widgets.idl`

```
module Widgets
{
  interface W1 {...};
  interface W2 {...};
};
```

Running this file through the IDL-to-Java compiler places the Java bindings for W1 and W2 within the package Widgets. But what if there is an industry convention that states that a company's packages must reside within a package named com.*company name*? Then, the Widgets package does not conform. To follow the convention, it must be com.abc.Widgets. To place this package prefix onto the Widgets module, implement the following:

```
idlj -pkgPrefix Widgets com.abc Widgets.idl
```

Be aware that if you have an IDL file that includes Widgets.idl, the -pkgPrefix flag must appear on that command as well. If it does not, then your IDL file will look for a Widgets package rather than a com.abc.Widgets package.

If you have a number of these packages that require prefixes, it might be easier to place them into the idl.config file as described in "Specifying alternative locations for include files" on page 167. Each package prefix line should be of the form:

`PkgPrefix.`*`type=prefix`*

The line for the previous example would be:

```
PkgPrefix.Widgets=com.abc
```

Emitting makefiles and specifying the path separator character: When the Java bindings are compiled using a makefile, it can become tedious to build the makefile by hand. There are two arguments to the IDL-to-Java compiler that can help to build the makefile.

```
idlj -m My.idl
```

Besides the usual bindings, this command generates bfile My.u that contains the following lines:

```
MyHelper.java: My.idl
My.java: My.idl
MyHolder.java: My.idl
MyPackage/E.java: Embedded.idl
MyPackage/EHelper.java: Embedded.idl
MyPackage/EHolder.java: Embedded.idl
_MyStub.java: My.idl

MyHelper.java \
My.java \
MyHolder.java \
MyPackage/E.java \
MyPackage/EHelper.java \
MyPackage/EHolder.java \
_MyStub.java
```

If you build a makefile that runs on multiple platforms, the slash (/) character is not necessarily the file separator character. The build environment might have a special variable for the file separator character. If this variable were $(Sep), then the compiler can place this in place of the slash in My.u with the following command:

```
idlj -m -sep \$\(Sep\) My.idl
```

Now My.u contains the following:

```
MyHelper.java: My.idl
My.java: My.idl
MyHolder.java: My.idl
MyPackage$(Sep)E.java: Embedded.idl
MyPackage$(Sep)EHelper.java: Embedded.idl
MyPackage$(Sep)EHolder.java: Embedded.idl
_MyStub.java: My.idl

MyHelper.java \
My.java \
MyHolder.java \
MyPackage$(Sep)E.java \
MyPackage$(Sep)EHelper.java \
MyPackage$(Sep)EHolder.java \
_MyStub.java
```

Conventions used in documenting IDL syntax

The following conventions are used in these topics to describe the syntax of Interface Definition Language (IDL) as specified by the CORBA standard:

bold	Indicates literals (such as keywords)
variable	Indicates user-supplied elements
{ }	Groups related items together as a single item
[ ]	Encloses an optional item
*	Indicates zero or more repetitions of the preceding item
+	Indicates one or more repetitions of the preceding item
|	Separates alternatives
_	Within a set of alternatives, an underscore indicates the default, if defined

IDL lexical rules

Interface Definition Language (IDL) generally follows the same lexical rules as C and C++. Exceptions to C++ lexical rules include:

- IDL uses the ISO Latin-1 (8859.1) character set.
- White space is ignored except as token delimiters.
- C and C++ comment styles are supported.

- IDL supports standard C or C++ preprocessing, including macro substitution, conditional compilation, and source file inclusion.
- Identifiers (user-defined names for operations, attributes, instance variables, and so on) are composed of alphanumeric and underscore characters (with the first character alphabetic) and can be of arbitrary length, up to an operating-system limit of about 250 characters.
- Identifiers must be spelled consistently with respect to case throughout a specification.
- Identifiers that differ only in case yield a compilation error.
- Within a particular name scope, there is a single namespace for all identifiers, regardless of their type. For example, using the same identifier for a constant and an interface name within the same name scope yields a compilation error.
- Integer, floating point, character, and string literals are defined as in C and C++.

IDL reserved words

The terms listed below are reserved words and cannot be used otherwise. Reserved words must be spelled using upper- and lower-case characters exactly as shown in the table. For example, ″void″ is correct, but ″Void″ yields a compilation error.

Reserved words for IDL

abstract	double	local	raises	typedef
any	enum	long	readonly	unsigned
attribute	exception	module	sequence	union
boolean	factory	native	short	ValueBase
case	FALSE	Object	string	valuetype
char	fixed	octet	struct	void
const	float	oneway	supports	wchar
context	in	out	switch	wstring
custom	inout	private	TRUE	
default	interface	public	truncatable	

Syntax for comments in IDL code

The Interface Definition Language (IDL) supports both C and C++ comment styles. Two slashes (//) start a line comment, which finishes at the end of the current line. A slash and an asterisk (/*) start a block comment that finishes with an asterisk and a slash (*/). Block comments do not nest. The two comment styles can be used interchangeably.

Because comments appearing in an IDL specification can be transferred to the files that the IDL Compiler generates and because these files are often used as input to a programming language compiler, avoid using characters that are not generally allowed in comments of most programming languages. For example, the C language does not allow an asterisk and a slash (*/) to occur within a comment. Thus, avoid it even when you are using C++ style comments in the IDL file.

IDL also supports throw-away comments. They can appear anywhere in an IDL specification. Throw-away comments start with the string of two slashes and a number sign (//#) and end at the end of the line. Use throw-away comments to comment out portions of an IDL specification.

CORBA: Resources for learning

Use the following links to find relevant supplemental information about CORBA. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks™ that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:

- "Planning, business scenarios, and IT architecture"
- "Programming model and decisions"
- "Administration"

Planning, business scenarios, and IT architecture

- Introduction to CORBA
  http://developer.java.sun.com/developer/onlineTraining/corba/
- The Object Management Group (OMG) Website
  http://www.corba.org/
- Welcome to OMG's CORBA for Beginners Page!
  http://cgi.omg.org/corba/beginners.html

Programming model and decisions

- A Brief Tutorial on CORBA
  http://www.cs.indiana.edu/~kksiazek/tuto.html

Administration

- Listing of all IBM WebSphere Application Server Redbooks
  http://publib-b.boulder.ibm.com/Redbooks.nsf/Portals/WebSphere
- WebSphere Application Server Enterprise Edition 4.0: A Programmer's Guide
  http://www.redbooks.ibm.com/redbooks/SG246504.html

Notices

This information was developed for products and services offered in the U.S.A. IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department LZKS
11400 Burnet Road
Austin, TX 78758
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written.

These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these

programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX
DB2
IBM
OS/390
Redbooks
WebSphere

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.