Getting Started with Java™ IDL

Java ™ IDL is a technology for distributed objects--that is, objects interacting on different platforms across a network. Java IDL enables objects to interact regardless of whether they're written in the Java programming language or another language such as C, C++, COBOL, or others.

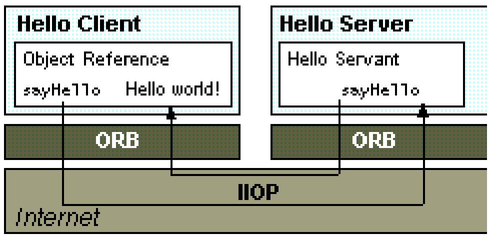
This is possible because Java IDL is based on the Common Object Request Brokerage Architecture (CORBA), an industry-standard distributed object model. A key feature of CORBA is IDL, a language-neutral Interface Definition Language. Each language that supports CORBA has its own IDL mapping--and as its name implies, Java IDL supports the mapping for Java. To learn more about the IDL-to-Java language mapping, see IDL-to-Java Language Mapping.

To support interaction between objects in separate programs, Java IDL provides an Object Request Broker, or ORB. The ORB is a class library that enables low-level communication between Java IDL applications and other CORBA-compliant applications.

This tutorial teaches the basic tasks needed to build a CORBA distributed application using Java IDL. You will build the classic "Hello World" program as a distributed application. The Hello World program has a single operation that returns a string to be printed.

Any relationship between distributed objects has two sides: the client and the server. The server provides a remote interface, and the client calls a remote interface. These relationships are common to most distributed object standards, including Java Remote Method Invocation (RMI, RMI-IIOP) and CORBA. Note that in this context, the terms client and server define object-level rather than application-level interaction--any application could be a server for some objects and a client of others. In fact, a single object could be the client of an interface provided by a remote object and at the same time implement an interface to be called remotely by other objects.

This figure shows how a one-method distributed object is shared between a CORBA client and server to implement the classic "Hello World" application.



A one-method distributed object shared between a CORBA client and server.

On the client side, the application includes a reference for the remote object. The object reference has a stub method, which is a stand-in for the method being called remotely. The stub is actually wired into the ORB, so that calling it invokes the ORB's connection capabilities, which forwards the invocation to the server.

On the server side, the ORB uses skeleton code to translate the remote invocation into a method call on the local object. The skeleton translates the call and any parameters to their implementation-specific format and calls the method being invoked. When the method returns, the skeleton code transforms results or errors, and sends them back to the client via the ORBs.

Between the ORBs, communication proceeds by means of a shared protocol, IIOP--the Internet Inter-ORB Protocol. IIOP, which is based on the standard TCP/IP internet protocol, defines how CORBA-compliant ORBs pass information back and forth. Like CORBA and IDL, the IIOP standard is defined by OMG, the Object Management Group.

The Java IDL Development Process and the Hello World Tutorial

This tutorial teaches the basic tasks in building a CORBA distributed application using Java IDL. You will build the classic "Hello World" program as a distributed application. The "Hello World" program has a single operation that returns a string to be printed.

Despite its simple design, the Hello World program lets you learn and experiment with all the tasks required to develop almost any CORBA program that uses static invocation. The following steps provide a general guide to designing and developing a distributed object application with Java IDL. Links to the relevant steps of the tutorial will guide you through creating this sample application.

1. **Define the remote interface**

   You define the interface for the remote object using the OMG's Interface Definition Langauge (IDL). You use IDL instead of the Java language because the `idlj` compiler automatically maps from IDL, generating all Java language stub and skeleton source files, along with the infrastructure code for connecting to the ORB. Also, by using IDL, you make it possible for developers to implement clients and servers in any other CORBA-compliant language.

   Note that if you're implementing a client for an existing CORBA service, or a server for an existing client, you would get the IDL interfaces from the implementer--such as a service provider or vendor. You would then run the `idlj` compiler over those interfaces and follow these steps.

   Writing the IDL file in this tutorial walks you through defining the remote interface for the simple "Hello World" example.

2. **Compile the remote interface**

   When you run the `idlj` compiler over your interface definition file, it generates the Java version of the interface, as well as the class code files for the stubs and skeletons that enable your applications to hook into the ORB.

Mapping Hello.idl to Java in this tutorial walks you through these steps for the simple "Hello World" example.

3. **Implement the server**

   Once you run the `idlj` compiler, you can use the skeletons it generates to put together your server application. In addition to implementing the methods of the remote interface, your server code includes a mechanism to start the ORB and wait for invocation from a remote client.

   Developing the Hello World Server walks you through writing a simple server for the "Hello World" application.

4. **Implement the client**

   Similarly, you use the stubs generated by the `idlj` compiler as the basis of your client application. The client code builds on the stubs to start its ORB, look up the server using the name service provided with Java IDL, obtain a reference for the remote object, and call its method.

   Developing a Client Application walks you through writing a simple client application.

5. **Start the applications**

   Once you implement a server and a client, you can start the name service, then start the server, then run the client.

   Running the Hello World Application walks you through running the server and client program that together make up the "Hello World" application, and the name service that enables them to find one another.

Using Stringified Object References walks you through making an object reference when there is no naming service.

Running the Hello World Application on Two Machines describes one way of distributing the simple application across two machines - a client and a server.

For More Information

Although concepts are explained as they are introduced in the tutorial, you will find more information in the Concepts section. New terms are linked to their definitions throughout the tutorial.

The Object Management Group no longer maintains this site, but the *CORBA for Beginnners* page contains links to web pages that provide introductory CORBA information.

Getting Started with Java IDL: Writing the Interface Definition

Note: All command and troubleshooting instructions apply to the Java 2 Platform, Standard Edition, version 1.4 and its version of `idlj` only.

Before you start working with Java IDL, you need to install version 1.4 of J2SE. J2SE v.1.4 provides the Application Programming Interface (API) and Object Request Broker (ORB) needed to enable CORBA-based distributed object interaction, as well as the `idlj` compiler. The `idlj` compiler uses the IDL-to-Java language mapping to convert IDL interface definitions to corresponding Java interfaces, classes, and methods, which you can then use to implement your client and server code.

This section teaches you how to write a simple IDL interface definition and how to translate the IDL interface to Java. It also describes the purpose of each file generated by the `idlj` compiler.

These topics are included in this section:

1. Writing `Hello.idl`
2. Understanding the IDL file
3. Mapping `Hello.idl` to Java
4. Understanding the idlj Compiler Output

Writing `Hello.idl`

To create the `Hello.idl` file,

1. Create a new directory, named `Hello`, for this application.
2. Start your favorite text editor and create a file named `Hello.idl` in this directory.
3. In your file, enter the code for the interface definition, **Hello.idl**:

```
module HelloApp
{
  interface Hello
  {
    string sayHello();
    oneway void shutdown();
  };
};
```

4. Save the file.

Understanding the IDL file

OMG IDL is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation's parameters. An OMG IDL interface provides the information needed to develop clients that use the interface's operations.

Clients are written in languages for which mappings from OMG IDL concepts have been defined. The mapping of an OMG IDL concept to a client language construct will depend on the facilities available in the client language. OMG specifies a mapping from IDL to several different programming languages, including C, C++, Smalltalk, COBOL, Ada, Lisp, Python, and Java. When mapped, each statement in OMG IDL is translated to a corresponding statement in the programming language of choice.

For example, you could use the tool `idlj` to map an IDL interface to Java and implement the client class. When you mapped the same IDL to C++ and implemented the server in that language, the Java client (through the Java ORB) and C++ server (through the C++ ORB) interoperate as though they were written in the same language.

The IDL for "Hello World" is extremely simple; its single interface has but two operations. You need perform only three steps:

1. Declare the CORBA IDL module
2. Declare the interface
3. Declare the operations

Declaring the CORBA IDL Module

A CORBA module is a namespace that acts as a container for related interfaces and declarations. It corresponds closely to a Java package. Each module statement in an IDL file is mapped to a Java package statement.

The module statement looks like this:

```
module HelloApp
{
    // Subsequent lines of code here.
};
```

When you compile the IDL, the module statement will generate a package statement in the Java code.

Declaring the Interface

Like Java interfaces, CORBA interfaces declare the API contract an object has with other objects. Each interface statement in the IDL maps to a Java interface statement when mapped.

In your `Hello.idl` file, the interface statement looks like this:

```
module HelloApp
{
  interface Hello  // These lines
  {                // declare the
                   // interface
  };               // statement.
};
```

When you compile the IDL, this statement will generate an interface statement in the Java code.

Declaring the Operations

CORBA operations are the behavior that servers promise to perform on behalf of clients that invoke them. Each operation statement in the IDL generates a corresponding method statement in the generated Java interface.

In your `Hello.idl` file, the operation statement looks like this:

```
module HelloApp
{
  interface Hello
  {
    string sayHello();        // This line is an operation statement.
    oneway void shutdown();   // This line is another
  };
};
```

The interface definition for our little "Hello World" application is now complete.

Mapping `Hello.idl` to Java

The tool `idlj` reads OMG IDL files and creates the required Java files. The `idlj` compiler defaults to generating only the client-side bindings. If you need both client-side bindings and server-side skeletons (as you do for our "Hello World" program), you must use the **`-fall`** option when running the `idlj` compiler. For more information on the IDL-to-Java compiler options, follow the link.

New in J2SE v.1.4: The default server-side mapping generated when either the `-fall` or `-fserver` arguments are used conform to Chapter 11, *Portable Object Adapter* (POA) of the CORBA 2.3.1 Specification (formal/99-10-07). For more information on the POA, link to Portable Object Adapter.

The advantages of using the Portable Object Adaptor (POA) are:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.

1. Make sure that the `j2sdk/bin` directory (or the directory containing `idlj`, `java`, `javac`, and `orbd`) are in your path.
2. Go to a command line prompt.
3. Change to the directory containing your `Hello.idl` file.

4. Enter the compiler command:

```
    idlj -fall Hello.idl
```

If you list the contents of the directory, you will see that a directory called `HelloApp` has been created and that it contains six files. Open `Hello.java` in your text editor. `Hello.java` is the *signature interface* and is used as the signature type in method declarations when interfaces of the specified type are used in other interfaces. It looks like this:

```
//Hello.java
package HelloApp;


/**
* HelloApp/Hello.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Hello.idl
*/

public interface Hello extends HelloOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity
{
} // interface Hello
```

With an interface this simple, it is easy to see how the IDL statements map to the generated Java statements.

IDL Statement	**Java Statement**
`module HelloApp`	`package HelloApp;`
`interface Hello`	`public interface Hello`

The single surprising item is the `extends` statement. All CORBA objects are derived from `org.omg.CORBA.Object` to ensure required CORBA functionality. The required code is generated by `idlj`; you do not need to do any mapping yourself.

In previous versions of the `idlj` compiler (known as `idltojava`), the operations defined on the IDL interface would exist in this file as well. Starting with J2SDK v1.3.0, in conformance with the CORBA 2.3.1 Specification (formal/99-10-07), the IDL-to-Java mapping puts all of the operations defined on the IDL interface in the *operations interface*, `HelloOperations.java`. The operations interface is used in the server-side mapping and as a mechanism for providing optimized calls for co-located clients and servers. For `Hello.idl`, this file looks like this:

```
//HelloOperations.java
package HelloApp;


/**
* HelloApp/HelloOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Hello.idl
*/

public interface HelloOperations
{
  String sayHello ();
  void Shutdown ();
} // interface HelloOperations
```

Because there are only two operations defined in this interface, it is easy to see how the IDL statements map to the generated Java statements.

IDL Statement	**Java Statement**
`string sayHello();`	`String sayHello();`
`oneway void shutdown();`	`void Shutdown ();`

Understanding the `idlj` Compiler Output

The `idlj` compiler generates a number of files. The actual number of files generated depends on the options selected when the IDL file is compiled. The generated files provide standard functionality, so you can ignore them until it is time to deploy and run your program. Under J2SE v.1.4, the files generated by the `idlj` compiler for `Hello.idl`, with the **`-fall`** command line option, are:

- `HelloPOA.java`

  This abstract class is the stream-based server skeleton, providing basic CORBA functionality for the server. It extends `org.omg.PortableServer.Servant`, and implements the `InvokeHandler` interface and the `HelloOperations` interface. The server class, `HelloServant`, extends `HelloPOA`.

- `_HelloStub.java`

  This class is the client stub, providing CORBA functionality for the client. It extends `org.omg.CORBA.portable.ObjectImpl` and implements the `Hello.java` interface.

- `Hello.java`

  This interface contains the Java version of our IDL interface. The `Hello.java` interface extends `org.omg.CORBA.Object`, providing standard CORBA object functionality. It also extends the `HelloOperations` interface and `org.omg.CORBA.portable.IDLEntity`.

- `HelloHelper.java`

  This class provides auxiliary functionality, notably the `narrow()` method required to cast CORBA object references to their proper types. The Helper class is responsible for reading and writing the data type to CORBA streams, and inserting and extracting the data type from `Any`s. The Holder class delegates to the methods in the Helper class for reading and writing.

- `HelloHolder.java`

  This final class holds a public instance member of type `Hello`. Whenever the IDL type is an `out` or an `inout` parameter, the Holder class is used. It provides operations for `org.omg.CORBA.portable.OutputStream` and `org.omg.CORBA.portable.InputStream` arguments, which CORBA allows, but which do not map easily to Java's semantics. The Holder class delegates to the methods in the Helper class for reading and writing. It implements `org.omg.CORBA.portable.Streamable`.

- `HelloOperations.java`

  This interface contains the methods `sayHello()` and `shutdown()`. The IDL-to-Java mapping puts all of the operations defined on the IDL interface into this file, which is shared by both the stubs and skeletons.

When you write the IDL interface, you do all the programming required to generate all these files for your distributed application. The next steps are to implement the client and server classes. In the steps that follow, you will create the `HelloClient.java` client class and the `HelloServer.java` server class.

Troubleshooting

- Error Message: "idlj" not found

  If you try to run `idlj` on the file `Hello.idl` and the system cannot find `idlj`, it is most likely not in your path. Make certain that the location of `idlj` (the J2SDK v.1.4 `bin` directory) is in your path, and try again.

For More Information

- IDL to Java Language Mapping Overview

  Provides the basics for mapping IDL constructs to the corresponding Java statements.

- Chapter 3 of the OMG CORBA 2.3.1 specification, formal/99-10-07

  Provides the complete specification for OMG Interface Definition Language. At this writing, the specification can be downloaded from http://cgi.omg.org/c gi-bin/doc?formal/99-10-07.

Getting Started with Java IDL: Developing the Hello World Server

The example server consists of two classes, the servant and the server. The **servant**, `HelloImpl`, is the implementation of the `Hello` IDL interface; each `Hello` instance is implemented by a `HelloImpl` instance. The **servant** is a subclass of `HelloPOA`, which is generated by the `idlj` compiler from the example IDL.

The **servant** contains one method for each IDL operation, in this example, the `sayHello()` and `shutdown()` methods. Servant methods are just like ordinary Java methods; the extra code to deal with the ORB, with marshaling arguments and results, and so on, is provided by the skeleton.

The **server** class has the server's `main()` method, which:

- Creates and initializes an ORB instance
- Gets a reference to the root POA and activates the `POAManager`
- Creates a servant instance (the implementation of one CORBA `Hello` object) and tells the ORB about it
- Gets a CORBA object reference for a naming context in which to register the new CORBA object
- Gets the root naming context
- Registers the new object in the naming context under the name "Hello"
- Waits for invocations of the new object from the client

This lesson introduces the basics of writing a CORBA server. For an example of the "Hello World" program with a persistent object server, see *Example 2: Hello World with Persistent State*. For more discussion of CORBA servers, see *Developing Servers*.

The steps in this lesson cover:

1. Creating HelloServer.java
2. Understanding HelloServer.java
3. Compiling the Hello World Server

Creating HelloServer.java

To create `HelloServer.java`,

1. Start your text editor and create a file named `HelloServer.java` in your main project directory, `Hello`.

2. Enter the following code for `HelloServer.java` in the text file. The following section, Understanding HelloServer.java, explains each line of code in some detail.

```
// HelloServer.java
// Copyright and License
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

class HelloImpl extends HelloPOA {
  private ORB orb;

  public void setORB(ORB orb_val) {
    orb = orb_val;
  }

  // implement sayHello() method
  public String sayHello() {
    return "\nHello world !!\n";
  }

  // implement shutdown() method
  public void shutdown() {
    orb.shutdown(false);
  }
}


public class HelloServer {

  public static void main(String args[]) {
    try{
      // create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // get reference to rootpoa & activate the POAManager
      POA                          rootpoa                          =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();

      // create servant and register it with the ORB
      HelloImpl helloImpl = new HelloImpl();
      helloImpl.setORB(orb);

      // get object reference from the servant
      org.omg.CORBA.Object                   ref                    =
rootpoa.servant_to_reference(helloImpl);
      Hello href = HelloHelper.narrow(ref);

      // get the root naming context
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
      // Use NamingContextExt which is part of the Interoperable
      // Naming Service (INS) specification.
      NamingContextExt                    ncRef                    =
NamingContextExtHelper.narrow(objRef);

      // bind the Object Reference in Naming
      String name = "Hello";
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, href);

      System.out.println("HelloServer ready and waiting ...");

      // wait for invocations from clients
      orb.run();
    }

      catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }

      System.out.println("HelloServer Exiting ...");

  }
}
```

3. Save and close `HelloServer.java`.

Understanding HelloServer.java

This section explains each line of `HelloServer.java`, describing what the code does, as well as why it is needed for this application.

Performing Basic Setup

The structure of a CORBA server program is the same as most Java applications: You import required library packages, declare the server class, define a `main()` method, and handle exceptions.

Importing Required Packages

First, we import the packages required for the server class:

```
// The package containing our stubs
import HelloApp.*;

// HelloServer will use the naming service
import org.omg.CosNaming.*;

//  The  package  containing  special  exceptions  thrown  by  the  name
service
import org.omg.CosNaming.NamingContextPackage.*;

// All CORBA applications need these classes
import org.omg.CORBA.*;
```

```
// Classes needed for the Portable Server Inheritance Model
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

// Properties to initiate the ORB
import java.util.Properties;
```

Defining the Servant Class

In this example, we are defining the class for the servant object within `HelloServer.java`, but outside the `HelloServer` class.

```
class HelloImpl extends HelloPOA
{
  // The sayHello() and shutdown() methods go here.
}
```

The servant is a subclass of `HelloPOA` so that it inherits the general CORBA functionality generated for it by the compiler.

First, we create a private variable, `orb` that is used in the `setORB(ORB)` method. The `setORB` method is a private method defined by the application developer so that they can set the ORB value with the servant. This ORB value is used to invoke `shutdown()` on that specific ORB in response to the `shutdown()` method invocation from the client.

```
  private ORB orb;

  public void setORB(ORB orb_val) {
    orb = orb_val;
  }
```

Next, we declare and implement the required `sayHello()` method:

```
  public String sayHello()
  {
    return "\nHello world!!\n";
  }
```

And last of all, we implement the `shutdown()` method in a similar way. The `shutdown()` method calls the `org.omg.CORBA.ORB.shutdown(boolean)` method for the ORB. The `shutdown(false)` operation indicate that the ORB should shut down immediately, without waiting for processing to complete.

```
  public void shutdown() {
    orb.shutdown(false);
  }
```

Declaring the Server Class

The next step is to declare the server class:

```
public class HelloServer
{
  // The main() method goes here.
}
```

Defining the main() Method

Every Java application needs a `main` method. It is declared within the scope of the `HelloServer` class:

```
  public static void main(String args[])
  {
    // The try-catch block goes here.
  }
```

Handling CORBA System Exceptions

Because all CORBA programs can throw CORBA system exceptions at runtime, all of the `main()` functionality is placed within a try-catch block. CORBA programs throw runtime exceptions whenever trouble occurs during any of the processes (marshaling, unmarshaling, upcall) involved in invocation. The exception handler simply prints the exception and its stack trace to standard output so you can see what kind of thing has gone wrong.

The try-catch block is set up inside `main()`, as shown:

```
    try{

      // The rest of the HelloServer code goes here.

    } catch(Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }
```

Creating and Initializing an ORB Object

A CORBA server needs a local ORB object, as does the CORBA client. Every server instantiates an ORB and registers its **servant objects** so that the ORB can find the server when it receives an invocation for it.

The ORB variable is declared and initialized inside the try-catch block.

```
      ORB orb = ORB.init(args, null);
```

The call to the ORB's `init()` method passes in the server's command line arguments, allowing you to set certain properties at runtime.

Get a Reference to the Root POA and Activate the `POAManager`

The ORB obtains the initial object references to services such as the Name Service using the method `resolve_initial_references`.

The reference to the root POA is retrieved and the `POAManager` is activated from within the try-catch block.

```
      POA                           rootpoa                         =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
```

The `activate()` operation changes the state of the POA manager to active, causing associated POAs to start processing requests. The POA manager encapsulates the processing state of the POAs with which it is associated. Each `POA` object has an associated `POAManager` object. A POA manager may be associated with one or more POA objects.

Managing the Servant Object

A **server** is a process that instantiates one or more servant objects. The **servant** inherits from the interface generated by `idlj` and actually performs the work of the operations on that interface. Our `HelloServer` needs a `HelloImpl`.

Instantiating the Servant Object

We instantiate the servant object inside the try-catch block, just after activating the POA manager, as shown:

```
      HelloImpl helloImpl = new HelloImpl();
```

The section of code describing the servant class was explained previously.

In the next line of code, `setORB(orb)` is defined on the servant so that `ORB.shutdown()` can be called as part of the shutdown operation. This step is required because of the `shutdown()` method defined in `Hello.idl`.

```
      helloImpl.setORB(orb);
```

There are other options for implementing the shutdown operation. In this example, the `shutdown()` method called on the `Object` takes care of shutting down an ORB. In another implementation, the shutdown method implementation could have simply set a flag, which the server could have checked and called `shutdown()`.

The next set of code is used to get the object reference associated with the servant. The `narrow()` method is required to cast CORBA object references to their proper types.

```
      org.omg.CORBA.Object                    ref                   =
rootpoa.servant_to_reference(helloImpl);
      Hello href = HelloHelper.narrow(ref);
```

Working with COS Naming

The `HelloServer` works with the Common Object Services (COS) Naming Service to make the servant object's operations available to clients. The server needs an object reference to the naming service so that it can publish the references to the objects implementing various interfaces. These object references are used by the clients for invoking methods. Another way a servant can make the objects available to clients for invocations is by **stringifying** the object references to a file.

The two options for Naming Services shipped with J2SE v.1.4 are:

- `orbd`, which includes both a Transient Naming Service and a Persistent Naming Service, in addition to a Server Manager.
- `tnameserv` - a Transient Naming Service.

This example uses `orbd`.

Obtaining the Initial Naming Context

In the try-catch block, below getting the object reference for the servant, we call `orb.resolve_initial_references()` to get an object reference to the name server:

```
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
```

The string "NameService" is defined for all CORBA ORBs. When you pass in that string, the ORB returns a naming context object that is an object reference for the name service. The string "NameService" indicates:

- The naming service will be persistent when using ORBD's naming service, as we do in this example.
- The naming service will be transient when using `tnameserv`.

The proprietary string "TNameService" indicates that the naming service will be transient when using ORBD's naming service.

Narrowing the Object Reference

As with all CORBA object references, `objRef` is a generic CORBA object. To use it as a `NamingContextExt object`, you must narrow it to its proper type. The call to `narrow()` is just below the previous statement:

```
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
```

Here you see the use of an `idlj`-generated helper class, similar in function to `HelloHelper`. The `ncRef` object is now an `org.omg.CosNaming.NamingContextExt` and you can use it to access the naming service and register the server, as shown in the next topic.

The `NamingContextExt object` is new to J2SE v.1.4, and is part of the Interoperable Naming Service specification.

Registering the Servant with the Name Server

Just below the call to `narrow()`, we create a new `NameComponent` array. Because the path to `Hello` has a single element, we create the single-element array that `NamingContext.resolve` requires for its work:

```
        String name = "Hello";
        NameComponent path[] = ncRef.to_name( name );
```

Finally, we pass `path` and the servant object to the naming service, binding the servant object to the "Hello" id:

```
        ncRef.rebind(path, href);
```

Now, when the client calls `resolve("Hello")` on the initial naming context, the naming service returns an object reference to the `Hello` servant.

Waiting for Invocation

The previous sections describe the code that makes the server ready; the next section explains the code that enables it to simply wait around for a client to request its service. The following code, which is at the end of (but within) the try-catch block, shows how to accomplish this.

```
        orb.run();
```

When called by the main thread, `ORB.run()` enables the ORB to perform work using the main thread, waiting until an invocation comes from the ORB. Because of its placement in `main()`, after an invocation completes and `sayHello()` returns, the server will wait again. This is the reason that the `HelloClient` explicitly shuts down the ORB after completing its task.

Compiling the Hello World Server

Now we will compile the `HelloServer.java` so that we can correct any errors before continuing with this tutorial.

Windows users note that you should substitute backslashes (\) for the slashes (/) in all paths in this document.

To compile `HelloServer.java`,

1. Change to the `Hello` directory.
2. Run the Java compiler on `HelloServer.java`:

```
javac HelloServer.java HelloApp/*.java
```

3. Correct any errors in your file and recompile if necessary.
4. The files `HelloServer.class` and `HelloImpl.class` are generated in the `Hello` directory.

Running the Hello World Server

The document Running the Hello World Application discusses running `HelloServer` and the rest of the application.

Understanding The Server-Side Implementation Models

CORBA supports at least two different server-side mappings for implementing an IDL interface:

- **The Inheritance Model**

  Using the Inheritance Model, you implement the IDL interface using an implementation class that also extends the compiler-generated skeleton.

  Inheritance models include:

  - The OMG-standard, *POA*. Given an interface `My` defined in `My.idl`, the file `MyPOA.java` is generated by the `idlj` compiler. You must provide the implementation for `My` and it must inherit from `MyPOA`, a stream-based skeleton that extends `org.omg.PortableServer.Servant`, which serves as the base class for all POA servant implementations.

    New in J2SE v.1.4: The default server-side mapping generated when either the `-fall` or `-fserver` arguments are used conform to Chapter 11, *Portable Object Adapter* (POA) of the CORBA 2.3.1 Specification (formal/99-10-07). For more information on the POA, link to *Portable Object Adapter*.

    The advantages of using the Portable Object Adaptor (POA) are:

    - Allow programmers to construct object implementations that are portable between different ORB products.
    - Provide support for objects with persistent identities.
    - Provide support for transparent activation of objects.
    - Allow a single servant to support multiple object identities simultaneously.
  - *ImplBase*. Given an interface `My` defined in `My.idl`, the file `_MyImplBase.java` is generated. You must provide the implementation for `My` and it must inherit from `_MyImplBase`.

    NOTE: ImplBase is deprecated in favor of the POA model, but is provided to allow compatibility with servers written in J2SE 1.3 and prior. We do not recommend creating new servers using this nonstandard model.

- The Delegation Model

  Using the Delegation Model, you implement the IDL interface using two classes:

  - An IDL-generated Tie class that inherits from the compiler-generated skeleton, but delegates all calls to an implementation class.
  - A class that implements the IDL-generated operations interface (such as `HelloOperations`), which defines the IDL function.

  The Delegation model is also known as the *Tie* model, or the Tie Delegation model. It inherits from either the POA or ImplBase compiler-generated skeleton, so the models will be described as POA/Tie or ImplBase/Tie models in this document.

This tutorial presents the POA Inheritance model for server-side implementation. For tutorials using the other server-side implementations, see the following documents:

- *Java IDL: The "Hello World" Example with the POA/Tie Server-Side Model*

  You might want to use the Tie model instead of the typical Inheritance model if your implementation must inherit from some other implementation. Java allows any number of interface inheritance, but there is only one slot for class inheritance. If you use the inheritance model, that slot is used up . By using the Tie Model, that slot is freed up for your own use. The drawback is that it introduces a level of indirection: one extra method call occurs when invoking a method.

- *Java IDL: The "Hello World" Example with the ImplBase Server-Side Model*

  The ImplBase server-side model is an Inheritance Model, as is the POA model. Use the `idlj` compiler with the `-oldImplBase` flag to generate server-side bindings that are compatible with older version of Java IDL (prior to J2SE 1.4).

  Note that using the `-oldImplBase` flag is non-standard: these APIs are being deprecated. You would use this flag ONLY for compatibility with existing servers written in J2SE 1.3 or earlier. In that case, you would need to modify an existing MAKEFILE to add the `-oldImplBase` flag to the `idlj` compiler, otherwise POA-based server-side mappings will be generated.

For More Information

Exceptions: System Exceptions
: Explains how CORBA system exceptions work and provides details on the minor codes of Java IDL's system exceptions

Developing Servers
: Covers topics of interest to CORBA server programmers

Java IDL Naming Service

Covers the COS Naming Service in greater detail

Getting Started with Java IDL: Developing a Client Application

This topic introduces the basics of writing a CORBA client application. Included in this lesson are:

1. Creating HelloClient.java
2. Understanding HelloClient.java
3. Compiling HelloClient.java

Creating HelloClient.java

To create `HelloClient.java`,

1. Start your text editor and create a file named `HelloClient.java` in your main project directory, `Hello`.
2. Enter the following code for `HelloClient.java` in the text file. The following section, Understanding HelloClient.java, explains each line of code in some detail.

HelloClient.java

```
// Copyright and License

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloClient
{
  static Hello helloImpl;

  public static void main(String args[])
    {
      try{
        // create and initialize the ORB
  ORB orb = ORB.init(args, null);

        // get the root naming context
        org.omg.CORBA.Object objRef =
      orb.resolve_initial_references("NameService");
        //  Use  NamingContextExt  instead  of  NamingContext.  This
is
        // part of the Interoperable naming Service.
        NamingContextExt                  ncRef                   =
NamingContextExtHelper.narrow(objRef);

        // resolve the Object Reference in Naming
        String name = "Hello";
        helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));
        System.out.println("Obtained a handle on server object: 
" + helloImpl);
        System.out.println(helloImpl.sayHello());
        helloImpl.shutdown();

  } catch (Exception e) {
          System.out.println("ERROR : " + e) ;
    e.printStackTrace(System.out);
    }
     }

}
```

3. Save and close `HelloClient.java`.

Understanding HelloClient.java

This section explains each line of `HelloClient.java`, describing what the code does, as well as why it is needed for this application.

Performing Basic Setup

The basic shell of a CORBA client is the same as many Java applications: You import required library packages, declare the application class, define a `main` method, and handle exceptions.

Importing Required Packages

First, we import the packages required for the client class:

```
import HelloApp.*; // the package containing our stubs
import org.omg.CosNaming.*; // HelloClient will use the Naming Service
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*; // All CORBA applications need these classes
```

Declaring the Client Class

The next step is to declare the client class:

```
public class HelloClient
{
  // The main() method goes here.
}
```

Defining a `main()` Method

Every Java application needs a `main()` method. It is declared within the scope of the `HelloClient` class, as follows:

```
  public static void main(String args[])
  {
    // The try-catch block goes here.
  }
```

Handling CORBA System Exceptions

Because all CORBA programs can throw CORBA system exceptions at runtime, all of the `main()` functionality is placed within a try-catch block. CORBA programs throw system exceptions whenever trouble occurs during any of the processes (marshaling, unmarshaling, upcall) involved in invocation.

Our exception handler simply prints the name of the exception and its stack trace to standard output so you can see what kind of thing has gone wrong.

The try-catch block is set up inside `main()`,

```
    try{

      // Add the rest of the HelloClient code here.

    } catch(Exception e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
      }
```

Creating an ORB Object

A CORBA client needs a local ORB object to perform all of its marshaling and IIOP work. Every client instantiates an `org.omg.CORBA.ORB` object and initializes it by passing to the object certain information about itself.

The ORB variable is declared and initialized inside the try-catch block.

```
        ORB orb = ORB.init(args, null);
```

The call to the ORB's `init()` method passes in your application's command line arguments, allowing you to set certain properties at runtime.

Finding the Hello Server

Now that the application has an ORB, it can ask the ORB to locate the actual service it needs, in this case the Hello server. There are a number of ways for a CORBA client to get an initial object reference; our client application will use the COS Naming Service specified by OMG and provided with Java IDL. See Using Stringified Object References for information on how to get an initial object reference when there is no naming service available.

The two options for Naming Services shipped with J2SE v.1.4 are `orbd`, which is a daemon process containing a Bootstrap Service, a Transient Naming Service, a Persistent Naming Service, and a Server Manager, and `tnameserv`, a transient naming service. This example uses `orbd`.

Obtaining the Initial Naming Context

The first step in using the naming service is to get the initial naming context. In the try-catch block, below your ORB initialization, you call `orb.resolve_initial_references()` to get an object reference to the name server:

```
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
```

The string "NameService" is defined for all CORBA ORBs. When you pass in that string, the ORB returns the initial naming context, an object reference to the name service. The string "NameService" indicates:

- The persistent naming service will be used when using ORBD as the naming service.
- The transient naming service will be used when using `tnameserv` as the naming service.

The string "TNameService" indicates that the transient naming service will be used when ORBD is the naming service. In this example, we are using the persistent naming service that is a part of `orbd`.

Narrowing the Object Reference

As with all CORBA object references, `objRef` is a generic CORBA object. To use it as a `NamingContextExt` object, you must narrow it to its proper type.

```
        NamingContextExt                    ncRef                    =
NamingContextExtHelper.narrow(objRef);
```

Here we see the use of an `idlj`-generated helper class, similar in function to `HelloHelper`. The `ncRef` object is now an `org.omg.CosNaming.NamingContextExt` and you can use it to access the naming service and find other services. You will do that in the next step.

The `NamingContextExt` object is new to J2SE v.1.4, and is part of the Interoperable Naming Service.

Resolve the Object Reference in Naming

To publish a reference in the Naming Service to the `Hello` object implementing the `Hello` interface, you first need an identifying string for the `Hello` object.

```
      String name = "Hello";
```

Finally, we pass `name` to the naming service's `resolve_str()` method to get an object reference to the Hello server and narrow it to a `Hello` object:

```
        helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));
        System.out.println("Obtained a handle on server object: " +
helloImpl);
```

Here you see the `HelloHelper` helper class at work. The `resolve_str()` method returns a generic CORBA object as you saw above when locating the name service itself. Therefore, you immediately narrow it to a `Hello` object, which is the object reference you need to perform the rest of your work. Then, you send a message to the screen confirming that the object reference has been obtained.

Invoking the `sayHello()` Operation

CORBA invocations look like a method call on a local object. The complications of marshaling parameters to the wire, routing them to the server-side ORB, unmarshaling, and placing the upcall to the server method are completely transparent to the client programmer. Because so much is done for you by generated code, invocation is really the easiest part of CORBA programming.

Finally, we print the results of the invocation to standard output and explicitly shutdown the ORB:

```
        System.out.println(helloImpl.sayHello());
        helloImpl.shutdown();
```

Compiling HelloClient.java

Now we will compile `HelloClient.java` so that we can correct any errors before continuing with this tutorial.

Windows users note that you should substitute backslashes (\) for the slashes (/) in all paths in this document.

To compile `HelloClient.java`,

1. Change to the `Hello` directory.
2. Run the Java compiler on `HelloClient.java`:

```
javac HelloClient.java HelloApp/*.java
```

3. Correct any errors in your file and recompile if necessary.
4. The `HelloClient.class` is generated to the `Hello` directory.

Running the Client Application

Running the Hello World application is covered in Running the Hello World Application.

For More Information

Developing Clients
Covers topics of interest to CORBA client programmers
Exceptions: System Exceptions

Explains how CORBA system exceptions work and provides details on the minor codes of Java IDL's system exceptions

Initialization: System Properties

Explains what properties can be passed to the ORB at initialization

Naming Service

Covers the COS Naming Service in greater detail

Getting Started with Java IDL
Running the Hello World Application

This topic walks you through running the server and client program that together make up the "Hello World" application.

Running the Hello World Application

Despite its simple design, the Hello World program lets you learn and experiment with all the tasks required to develop almost any CORBA program that uses static invocation.

This example requires a naming service to make the servant object's operations available to clients. The server needs an object reference to the naming service so that it can publish the references to the objects implementing various interfaces. These object references are used by the clients for invoking methods. The two options for Naming Services shipped with J2SE v.1.4 are `tnameserv`, a transient naming service, and `orbd`, which is a daemon process containing a Bootstrap Service, a Transient Naming Service, a Persistent Naming Service, and a Server Manager. This example uses `orbd`.

When running this example, remember that, when using Solaris software, you must become root to start a process on a port under 1024. For this reason, we recommend that you use a port number greater than or equal to 1024. The `-ORBInitialPort` option is used to override the default port number in this example. The following instructions assume you can use port 1050 for the Java IDL Object Request Broker Daemon, `orbd`. You can substitute a different port if necessary. When running these examples on a Windows machine, subtitute a backslash (\) in path names.

To run this client-server application on your development machine:

1. Start `orbd`.

   To start `orbd` from a UNIX command shell, enter:

   ```
     orbd -ORBInitialPort 1050 -ORBInitialHost localhost&
   ```

   From an MS-DOS system prompt (Windows), enter:

   ```
     start orbd -ORBInitialPort 1050 -ORBInitialHost localhost
   ```

   Note that `1050` is the port on which you want the name server to run. `-ORBInitialPort` is a required command-line argument. Note that when using Solaris software, you must become root to start a process on a port under 1024. For this reason, we recommend that you use a port number greater than or equal to 1024.

Note that -ORBInitialHost is also a required command-line argument. For this example, since both client and server on running on the development machine, we have set the host to localhost. When developing on more than one machine, you will replace this with the name of the host. For an example of how to run this program on two machines, see The Hello World Example on Two Machines.

2. Start the Hello server.

   To start the Hello server from a UNIX command shell, enter:

   ```
   java HelloServer -ORBInitialPort 1050 -ORBInitialHost localhost&
   ```

   From an MS-DOS system prompt (Windows), enter:

   ```
   start  java  HelloServer  -ORBInitialPort  1050  -ORBInitialHost
   localhost
   ```

   For this example, you can omit -ORBInitialHost localhost since the name server is running on the same host as the Hello server. If the name server is running on a different host, use -ORBInitialHost *nameserverhost* to specify the host on which the IDL name server is running.

   Specify the name server (orbd) port as done in the previous step, for example, -ORBInitialPort 1050.

3. Run the client application:

   ```
   java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost
   ```

   For this example, you can omit -ORBInitialHost localhost since the name server is running on the same host as the Hello client. If the name server is running on a different host, use -ORBInitialHost *nameserverhost* to specify the host on which the IDL name server is running.

   Specify the name server (orbd) port as done in the previous step, for example, -ORBInitialPort 1050.

4. The client prints the string from the server to the command line:

   ```
   Hello world!!
   ```

The name server, like many CORBA servers, runs until you explicitly stop it. To avoid having many servers running, kill the name server process after the client application returns successfully. To do this from a DOS prompt, select the window that is running the server and enter Ctrl+C to shut it down. To do this from a Unix shell, find the process, and kill it.

Getting Started with Java IDL: Using Stringified Object References

To invoke an operation on a CORBA object, a client application needs a reference to the object. You can get such references in a number of ways, such as calling `ORB.resolve_initial_references()` or using another CORBA object (like the name service). In previous sections of this tutorial, you used both of these methods to get an initial object reference.

Often, however, there is no naming service available in the distributed environment. In that situation, CORBA clients use a ***stringified* object reference** to find their first object. **A stringified object reference is an object reference that has been converted to a string so that it may be stored on disk in a text file (or stored in some other manner)**. Such strings should be treated as opaque because they are ORB-implementation independent. Standard `object_to_string` and `string_to_object` methods on `org.omg.CORBA.Object` make stringified references available to all CORBA Objects.

Example:

```
// Ejemplo conversion a String de una referencia
// Lado Servidor
// ...

    ImplContador unObjetoContador = new ImplContador();
    org.omg.CORBA.Object refObjCORBA =
       rootPOA.servant_to_reference(unObjetoContador);
    Contador refContador = ContadorHelper.narrow(refObjCORBA);
// ... y exporta su referencia en un fichero:
try {
    String ref = orb.object_to_string(refContador);
    String refFile = "Contador.ref";
    java.io.PrintWriter out = new java.io.PrintWriter(
             new java.io.FileOutputStream(refFile));
             out.println(ref);
             out.close();
} catch(java.io.IOException ex) {
    ex.printStackTrace();
    System.exit(1);
}
```

```
// Ejemplo conversion a String de una referencia
// Lado Cliente
// ...
org.omg.CORBA.Object refObjCORBA = null;
try {
    String refFile = "Contador.ref";
    java.io.BufferedReader in =
    new java.io.BufferedReader(new java.io.FileReader(refFile));
    String ref = in.readLine();
    refObjCORBA = orb.string_to_object(ref);
}
catch(java.io.IOException ex) {
    ex.printStackTrace();
    System.exit(1);
}
Contador refContador = ContadorHelper.narrow(refObjCORBA);
```

Java IDL: The "Hello World" Example on Two Machines

To enable the Hello World Tutorial to run on two machines, follow the steps as directed in the tutorial, with the following changes. This tutorial was written for the Java (tm) 2 Platform, Standard Edition (J2Se(tm)), version 1.4. In this example, the client, stubs, and skeletons are located on the client machine, and the server and name server are located on the server machine. This scenario can be changed to meet your needs and is provided simply as an introduction to one way to distribute an application across two machines.

1. Create (as shown in the tutorial) and compile the `Hello.idl` file on the client machine:

```
  idlj -fall Hello.idl
```

2. Create `HelloClient.java` on the **client** machine. Compile the `*.java` files, including the stubs and skeletons (which are in the directory `HelloApp`):

```
   javac *.java HelloApp/*.java
```

3. Create `HelloServer.java` on the **server** machine. Compile the `.java` files:

```
   javac *.java
```

4. Start the Java Object Request Broker Daemon, `orbd`, which includes a Naming Service, on the server machine. To do this on Unix:

```
   orbd -ORBInitialPort 1050 -ORBInitialHost servermachinename&
```

   To do this on Windows:

```
   start      orbd      -ORBInitialPort      1050      -ORBInitialHost
servermachinename
```

   Both `-ORBInitialPort` and `-ORBInitialHost` are required arguments on the `orbd` command line. This example starts the name server on port `1050`, because on Solaris you must become root to start a process on a port under 1024. If you want to use a different *nameserverport*, replace 1050 with the correct port number throughout this example.

   When using `orbd`, the ORBD must be run on the same machine as the machine on which the servers will be activated. Another Naming Service, `tnameserv`, could be used if you prefer to run the Naming Service on a machine other than the machine on which the servers will be activated.

5. On the server machine, start the Hello server, as follows:

```
    java HelloServer -ORBInitialPort 1050
```

If you used a different *nameserverport*, replace 1050 with the correct port number. You do not need to specify the `-ORBInitialHost` argument because the Hello server will be running on the same host as the name server in this example. If the Name Server were running on a different machine, you would specify which machine using the `-ORBInitialHost nameserverhost` argument.

6. On the client machine, run the Hello application client. From a DOS prompt or shell, type:

```
    java HelloClient -ORBInitialHost nameserverhost -ORBInitialPort
1050
```

Note that *nameserverhost* is the host on which the IDL name server is running. In this case, it is the server machine.

If you used a different *nameserverport*, replace 1050 with the correct port number.

7. Kill or stop `orbd` when finished. The name server will continue to wait for invocations until it is explicitly stopped.