Java IDL

9th March 2005

Abstract

Java IDL is a technology that enables distributed 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 aim of this worksheet is to provide a basic introduction to Java IDL programming. This worksheet is a stripped down version of the Sun Java IDL tutorial.

1 Introduction

Java IDL is a technology that enables distributed 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 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.

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.

Figure 1 shows how a one-method distributed object is shared between a CORBA client and server to implement the classic "Hello World" application.

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

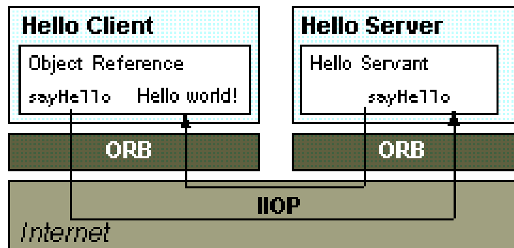

Figure 1: A one-method distributed object shared between a CORBA client and server.

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.

2 Define the remote interface

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.

Below is an example interface that defines two methods including the now familiar `getCurrentTime()`.

```
module MyCORBADemo {
    interface MyIDLInterface {
        string getCurrentTime();
        oneway void shutdown();
    };
};
```

Enter this code into a text editor and save with the name `MyIDLInterface.idl`

2.1 The IDL Interface

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

2.2 Mapping 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 , you must use the `-fall` option when running the idlj compiler.

2.3 Compiling the Interface

To compile the interface enter the command below:

```
idlj -fall MyIDLInterface.idl
```

If you list the contents of the directory, you will see that a directory called MyCORBADemo has been created and that it contains six files. Open `MyIDLInterface.java` in your text editor. `MyIDLInterface.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 should look like this:

```
package MyCORBADemo;
/**
* MyCORBADemo/MyIDLInterface.java .
* Generated by the IDL-to-Java compiler (portable), version "3.2"
* from MyIDLInterface.idl
* 09 March 2005 19:28:24 o'clock GMT
*/
public interface MyIDLInterface extends MyIDLInterfaceOperations,
                    org.omg.CORBA.Object,
                    org.omg.CORBA.portable.IDLEntity
{
} // interface MyIDLInterface
```

As you can see the IDL module statement maps to the package statement in Java and the IDL interface statement to the Java public interface 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.

The operations defined on the IDL interface are put in the operations interface, `MyIDLInterfaceOperations.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 `MyIDLInterface.idl`, this file looks like this:

```
package MyCORBADemo;
/**
* MyCORBADemo/MyIDLInterfaceOperations.java .
* Generated by the IDL-to-Java compiler (portable), version "3.2"
* from MyIDLInterface.idl
* 09 March 2005 19:28:24 o'clock GMT
*/
public interface MyIDLInterfaceOperations
{
    String getCurrentTime ();
    void shutdown ();
} // interface MyIDLInterfaceOperations
```

2.4 `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 `MyIDLInterface.idl`, with the -fall command line option, are:

- `MyIDLInterfacePOA.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 `MyIDLInterfaceOperations` interface. The server class, `MyIDLInterfaceServant`, extends `MyIDLInterfacePOA`.

- `_MyIDLInterfaceStub.java`

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

- `MyIDLInterface.java`

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

- `MyIDLInterfaceHelper.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 Anys. The Holder class delegates to the methods in the Helper class for reading and writing.

- `MyIDLInterfaceHolder.java`

This final class holds a public instance member of type `MyIDLInterface`. 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`.

- MyIDLInterfaceOperations.java

This interface contains the methods getCurrentTime() 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.

3 Developing the Server

The example server consists of two classes, the servant and the server. The servant, MyServerImpl, is the implementation of the `MyIDLInterface` IDL interface; each `MyIDLInterface` instance is implemented by a `MyIDLInterfaceImpl` instance. The servant is a subclass of `MyIDLInterfacePOA`, which is generated by the `idlj` compiler from the example IDL.

The servant contains one method for each IDL operation, in this example, the `getCurrentTime()` 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 `MyIDLInterface` 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 "`MyIDLInterface`"
- Waits for invocations of the new object from the client

3.1 Server Code

The code for the server is shown below. Enter it into a text file and save as MyServer.java.

```
import MyCORBADemo.*;
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.Date;
import java.util.Properties;

class MyServerImpl extends MyIDLInterfacePOA {

    private ORB orb;

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

    // implement getCurrentTime() method
    public String getCurrentTime() {
        Date now = new Date();
        return now.toString();
    }

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

}

public class MyServer {

    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
            MyServerImpl myServerImpl = new MyServerImpl();
            myServerImpl.setORB(orb);
            // get object reference from the servant
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(myServerImpl);
            MyIDLInterface href = MyIDLInterfaceHelper.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 = "MyIDLInterface";
            NameComponent path[] = ncRef.to_name( name );
            ncRef.rebind(path, href);
            System.out.println("MyServer 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("MyServer Exiting ...");
    }

}
```

3.2 Understanding the server code

This section explains the server code and why it is needed for this application.

3.2.1 Importing Required Package

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

```
// The package containing our stubs
import MyCORBADemo.*;
// MyServer 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;
```

3.2.2 Defining the Servant Class

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

```
class MyServerImpl extends MyIDLInterfacePOA
{
// The getCurrentTime() and shutdown() methods go here.
}
```

The servant is a subclass of `MyIDLInterfacePOA` 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 getCurrentTime() method:

```
public String getCurrentTime() {
    Date now = new Date();
    return now.toString();
}
```

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);
}
```

3.2.3 Declaring the Server Class

The next step is to declare the server class:

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

3.2.4 Defining the main() Method

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

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

3.2.5 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 MyServer code goes here.
} catch(Exception e) {
    System.err.println("ERROR: " + e);
    e.printStackTrace(System.out);
}
```

3.2.6 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.

3.2.7 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.

3.2.8 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 MyServer needs a MyServerImpl.

3.2.9 Instantiating the Servant Object

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

```
MyServerImpl myImpl = new MyServerImpl();
```

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 `MyIDLInterface.idl`.

```
myServerImpl.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(myServerImpl);
MyIDLInterface href = MyIDLInterfaceHelper.narrow(ref);
```

3.2.10 Working with COS Naming

The MyServer 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.

3.2.11 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.

3.2.12 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 `MyIDLInterfaceHelper`. The ncRef object is now an `org.omg.CosNaming.NamingContextExt` and you can use it to access the naming service and register the server.

3.2.13 Registering the Servant with the Name Server

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

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

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

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

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

3.2.14 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 `getCurrentTime()` returns, the server will wait again. This is the reason that the MyClient explicitly shuts down the ORB after completing its task.

3.3 Compiling the Server

Now we will compile the MyServer.java using the following command:

javac MyServer.java MyCORBADemo/*.java

4 Developing the Client

Below is the client code, enter it into a text file and save as `MyClient.java`.

```
import MyCORBADemo.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class MyClient {

    static MyIDLInterface myImpl;
```

```
    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 = "MyIDLInterface";
            myImpl = MyIDLInterfaceHelper.narrow(ncRef.resolve_str(name));
            System.out.println("Obtained a handle on server object:  "
+ myImpl);
            System.out.println(myImpl.getCurrentTime());
            myImpl.shutdown();
        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }

}
```

4.1 Understanding MyClient.java

This section explains what the code does, as well as why it is needed for this application.

4.1.1 Importing Required Packages

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

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

4.1.2 Declaring the Client Class

The next step is to declare the client class:

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

4.1.3 Defining a main() Method

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

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

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

4.1.5 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.

4.1.6 Finding MyServer

Now that the application has an ORB, it can ask the ORB to locate the actual service it needs, in this case the `MyIDLInterface` 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.

4.1.7 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.

4.1.8 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 `MyIDLInterfaceHelper`. The ncRef object is now an org.omg.CosNaming.NamingContextExt and you can use it to access the naming service and find other services.

4.1.9 Resolve the Object Reference in Naming

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

```
String name = "MyIDLInterface";
```

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

```
myImpl = MyIDLInterfaceHelper.narrow(ncRef.resolve-str(name));
```

```
System.out.println("Obtained a handle on server object:  " + myImpl);
```

Here you see the `MyIDLInterfaceHelper` 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 `MyIDLInterface` 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.

4.1.10 Invoking the `getCurrentTime()` 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(myImpl.getCurrentTime());
myImpl.shutdown();
```

4.2 Compiling MyClient.java

We compile `MyClient.java` with the following command:

```
javac MyClient.java MyCORBADemo/*.java
```

5 Running the Application

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.

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.

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 &
```

Note that 1050 is the port on which you want the name server to run. -ORBInitialPort is a required command-line argument.

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.

2. Start the MyServer server.

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

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

For this example, you can omit -ORBInitialHost localhost since the name server is running on the same host as the MyServer 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 MyClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

For this example, you can omit -ORBInitialHost localhost since the name server is running on the same host as the MyClient 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:

```
Wed Mar 09 21:02:59 GMT 2005
```

6 Further Work

- Write your own CORBA application that passes a custom object (created by you) as a method parameter.
- Look through all the generated code and try to get a feel for what is going on in the background!

References

[1] Java IDL Tutorial - http://java.sun.com/docs/books/tutorial/idl/