Java IDL: The "Hello World" Example

POA model, **transient** server

This document is a high-level overview of how to create a complete CORBA (Common Object Request Broker Architecture) application using IDL (Interface Definiton Language) to define interfaces and the Java IDL compiler to generate stubs and skeletons. For more information on the development process, and a more detailed tutorial on creating a CORBA application using IDL, see Getting Started with Java IDL: The Hello World Tutorial. You can also create CORBA application by defining the interfaces in the Java programming language. For more information and a tutorial on this development process, see the Java RMI-IIOP documentation.

In this release of Java SE, the server-side implementation generated by the `idlj` compiler is the *Portable Servant Inheritance Model*, also known as the POA model. The POA, or Portable Object Adapter, is discussed in more detail in Portable Object Adapter. This document presents a sample application created using the default behavior of the `idlj` compiler, which uses a POA server-side model.

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.
  - *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

The Tie Model is a delegation model. Use the `idlj` compiler to first generate server-side bindings. Then, run the `idlj` compiler a second time with the with the `-fallTie` option to generate Tie model server-side bindings. For the interface `Hello`, `HelloPOATie.java` is one of the generated files. The constructor to `HelloPOATie` takes a `delegate` or a `delegate` and a `poa`. You must provide the implementation for `delegate` and/or the `poa`, but the delegate does not have to inherit from any other class, only the interface `HelloOperations`. For more information, refer to the *IDL to Java Language Mapping Specification*.

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 versions of Java IDL prior to J2SE 1.4. Given an interface `Hello` defined in `Hello.idl`, the file `_HelloImplBase.java` is generated. You must provide the implementation for `Hello` and it must inherit from `_HelloImplBase`.

  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.

This document contains:

- The example code for this application
- The IDL for a simple "Hello World" program
- A server that creates an object and publishes it with the naming service using the default server-side implementation (POA)
- An application client that knows the object's name, retrieves a reference for it from the naming service, and invokes the object
- Instructions for compiling and running the example

To create this example, create a directory named `hello/` where you develop sample applications and create the files in this directory, or download the example code and unzip it into your sample applications directory.

Defining the Interface (`Hello.idl`)

The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). IDL has a syntax similar to C++ and can be used to define modules, interfaces, data structures, and more. The IDL can be mapped to a variety of programming languages. The IDL mapping for Java is summarized in IDL to Java Language Mapping Summary.

The following code is written in the OMG IDL, and describes a CORBA object whose `sayHello()` operation returns a string and whose `shutdown()` method shuts down the ORB. To learn more about OMG IDL Syntax and Semantics, read Chapter 3 of the CORBA 2.3.1 Specification.

Hello.idl

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

NOTE: When writing code in OMG IDL, do not use an interface name as the name of a module. Doing so runs the risk of getting inconsistent results when compiling with tools from different vendors, thereby jeopardizing the code's portability. For example, code containing the same names could be compiled with the IDL to Java compiler from Sun Microsystems and get one result. The same code compiled with another vendor's IDL to Java compiler could produce a different result.

To complete the application, you simply provide the server (`HelloServer.java`) and client (`HelloClient.java`) implementations.

Implementing the Server (`HelloServer.java`)

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 `HelloServer` 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 example provides an example of a transient object 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*.

For more discussion of the code, see the detailed tutorial topic Getting Started with Java IDL: Developing a Hello World Server.

HelloServer.java

```
// 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
        // NameService invokes the name service
        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 ...");

    }
  }
```

Implementing the Client Application (`HelloClient.java`)

The example application client that follows:

- Creates and initializes an ORB
- Obtains a reference to the root naming context
- Looks up "Hello" in the naming context and receives a reference to that CORBA object
- Invokes the object's `sayHello()` and `shutdown()` operations and prints the result

For more discussion of the details of the code, see Getting Started with Java IDL: Developing a Client Application.

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);
          }
    }

}
```

Building and Running Hello World

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. Static invocation, which uses a client stub for the invocation and a server skeleton for the service being invoked, is used when the interface of the object is known at compile time. If the interface is not known at compile time, dynamic invocation must be used.

This example requires a naming service, which is a CORBA service that allows CORBA objects to be named by means of binding a name to an object reference. The name binding may be stored in the naming service, and a client may supply the name to obtain the desired object reference. The two options for Naming Services shipped with this release of Java SE include `orbd` (Solaris, Linux, or Mac OS X or Windows), a daemon process containing a Bootstrap Service, a Transient Naming Service, a Persistent Naming Service, and a Server Manager, and `tnameserv` (Solaris, Linux, or Mac OS X or Windows), a transient naming service that is provided for backward compatibility. 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. Change to the directory that contains the file `Hello.idl`.
2. Run the IDL-to-Java compiler, `idlj`, on the IDL file to create stubs and skeletons. This step assumes that you have included the path to the java/bin directory in your path.

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

   You must use the `-fall` option with the `idlj` compiler to generate both client and server-side bindings. This command line will generate the default server-side bindings, which assumes the POA Inheritance server-side model. For more information on the `idlj` options, see the man page for `idlj` (Solaris, Linux, or Mac OS X or Windows).

   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. 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 `HelloImpl` 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.

3. Compile the `.java` files, including the stubs and skeletons (which are in the directory `HelloApp`). This step assumes the java/bin directory is included in your path.

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

4. Start `orbd`.

   To start `orbd` from a Solaris, Linux, or Mac OS X command shell, enter:

   ```
   orbd -ORBInitialPort 1050&
   ```

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

   ```
   start orbd -ORBInitialPort 1050
   ```

   Note that `1050` is the port on which you want the name server to run. The `-ORBInitialPort` argument 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.

   For an example of how to run this program on two machines, see Running the Hello World Program on 2 machines.

5. Start the Hello server:

   To start the Hello server from a Solaris, Linux, or Mac OS X command shell, enter:

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

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

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

   You will see `HelloServer ready and waiting...` when the server is started.

   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`.

6. Run the client application:

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

   When the client is running, you will see a response such as the following on your terminal: `Obtained a handle on server object: IOR: (binary code) Hello World! HelloServer exiting...`

   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`.

When you have finished this tutorial, be sure to shut down or kill the name server (`orbd`). 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 shell on Solaris, Linux, or Mac OS X, find the process, and kill it. The

server will continue to wait for invocations until it is explicitly stopped.

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