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*](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlExample2.html). For more discussion of CORBA servers, see [*Developing Servers*](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlServers.html).

The steps in this lesson cover:

1. [Creating HelloServer.java](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/tutorial/GSserver.html#create)
2. [Understanding HelloServer.java](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/tutorial/GSserver.html#understand)
3. [Compiling the Hello World Server](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/tutorial/GSserver.html#compile)

**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](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/tutorial/GSserver.html#understand), explains each line of code in some detail.
3. // **HelloServer.java**
4. import HelloApp.\*;
5. import org.omg.CosNaming.\*;
6. import org.omg.CosNaming.NamingContextPackage.\*;
7. import org.omg.CORBA.\*;
8. import org.omg.PortableServer.\*;
9. import org.omg.PortableServer.POA;
10. import java.util.Properties;
11. class HelloImpl extends HelloPOA {
12. private ORB orb;
13. public void setORB(ORB orb\_val) {
14. orb = orb\_val;
15. }
17. // implement sayHello() method
18. public String sayHello() {
19. return "\nHello world !!\n";
20. }
22. // implement shutdown() method
23. public void shutdown() {
24. orb.shutdown(false);
25. }
26. }
27. public class HelloServer {
28. public static void main(String args[]) {
29. try{
30. // create and initialize the ORB
31. ORB orb = ORB.init(args, null);
32. // get reference to rootpoa and activate the POAManager
33. POA rootpoa = POAHelper.narrow(orb.resolve\_initial\_references("RootPOA"));
34. rootpoa.the\_POAManager().activate();
35. // create servant and register it with the ORB
36. HelloImpl helloImpl = new HelloImpl();
37. helloImpl.setORB(orb);
38. // get object reference from the servant
39. org.omg.CORBA.Object ref = rootpoa.servant\_to\_reference(helloImpl);
40. Hello href = HelloHelper.narrow(ref);
42. // get the root naming context
43. org.omg.CORBA.Object objRef =
44. orb.resolve\_initial\_references("NameService");
45. // Use NamingContextExt which is part of the Interoperable
46. // Naming Service (INS) specification.
47. NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
48. // bind the Object Reference in Naming
49. String name = "Hello";
50. NameComponent path[] = ncRef.to\_name( name );
51. ncRef.rebind(path, href);
52. System.out.println("HelloServer ready and waiting ...");
53. // wait for invocations from clients
54. orb.run();
55. }
57. catch (Exception e) {
58. System.err.println("ERROR: " + e);
59. e.printStackTrace(System.out);
60. }
62. System.out.println("HelloServer Exiting ...");
64. }
65. }
67. 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](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlGlossary.html#servant__object) 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](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlGlossary.html#server) is a process that instantiates one or more servant objects. The [servant](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlGlossary.html#servant__object) 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](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlGlossary.html#object__reference) 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 are:

* orbd ([Solaris, Linux, or Mac OS X](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/orbd.html) or [Windows](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/orbd.html)), which includes both a Transient Naming Service and a Persistent Naming Service, in addition to a Server Manager.
* tnameserv ([Solaris, Linux, or Mac OS X](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/tnameserv.html) or [Windows](https://docs.oracle.com/javase/8/docs/technotes/tools/windows/tnameserv.html)) - 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 part of the [Interoperable Naming Service](https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlNaming.html#INS) 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:
3. javac HelloServer.java HelloApp/\*.java
4. Correct any errors in your file and recompile if necessary.
5. The files HelloServer.class and HelloImpl.class are generated in the Hello directory.