

LIST OF ASSIGNMENTS LP-V

Sr. No.	List of Laboratory Assignments
1	Implement multi-threaded client/server Process communication using RMI.
2	Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).
3	Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.
4	Implement Berkeley algorithm for clock synchronization.
5	Implement token ring based mutual exclusion algorithm.
6	Implement Bully and Ring algorithm for leader election.
7	Create a simple web service and write any distributed application to consume the web service.
8	Mini Project (In group): A Distributed Application for Interactive Multiplayer Games

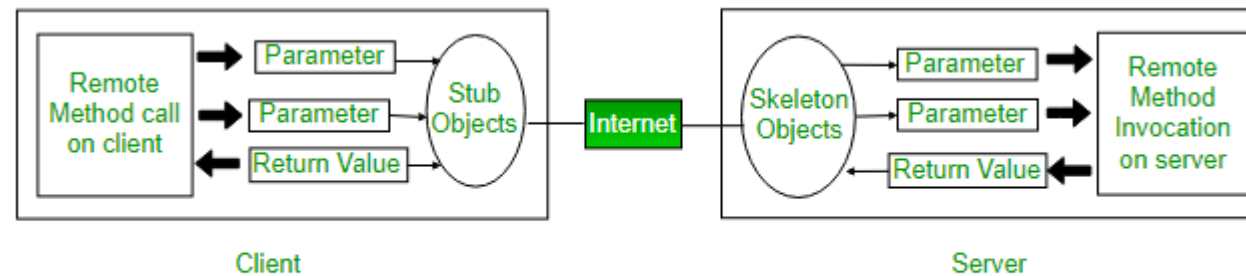
ASSIGNMENT NO. 1)	
Assignment No. 1)	Implement multi-threaded client/server Process communication using RMI.
Objective(s):	By the end of this assignment, the student will be able to implement any distributed applications based on RMI.
Tools	Eclipse, Java 8, rmiregistry

REMOTE METHOD INVOCATION (RMI)

- **Remote Method Invocation (RMI)** is an API which allows an object to invoke a method of an object that exists in another address space, which could be on the same machine or on a remote machine.
- Through RMI, object running in a JVM present on a computer (Client side) can invoke methods on an object present in another JVM (Server side).
- RMI creates a public remote server object that enables client and server side communications through simple method calls on the server object.
- The communication between client and server is handled by using two intermediate objects:
Stub object (on client side) and **Skeleton object** (on server side).
- **Stub Object:**
The stub object on the client machine **builds an information block and sends this information to the server.**
- The block consists of:
An identifier of the remote object to be used
Method name which is to be invoked
Parameters to the remote JVM.

RMI IMPLEMENTATION

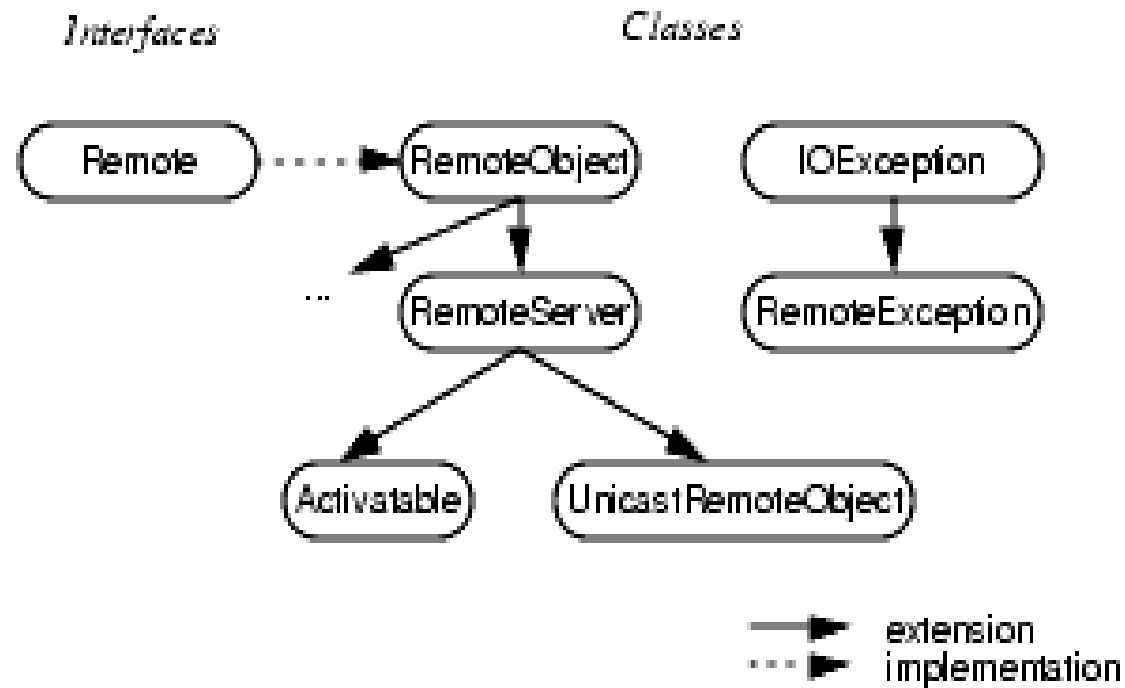
- **Skeleton Object**
- The skeleton object passes the request from the stub object to the remote object. It performs following tasks:
 - It calls the desired method on the real object present on the server.
 - It forwards the parameters received from the stub object to the method



Working of RMI

REMOTE METHOD INVOCATION (RMI)

The interfaces and classes that are responsible for specifying the remote behavior of the RMI system are defined in the java.rmi package hierarchy. The following figure shows the relationship between several of these interfaces and classes:



REMOTE METHOD INVOCATION (RMI)

- `java.rmi.Remote` **Interface:**
- In RMI, a *remote* interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine.
- **The `RemoteObject` Class and its Subclasses:**
- RMI server functions are provided by `java.rmi.server.RemoteObject` and its subclasses, `java.rmi.server.RemoteServer` and `java.rmi.server.UnicastRemoteObject`.
- The class `java.rmi.server.RemoteObject` provides implementations for the `java.lang.Object` methods that are sensible for remote objects.
- The methods needed to create remote objects and make them available to remote clients are provided by the class `UnicastRemoteObject`.
- The `java.rmi.server.UnicastRemoteObject` class defines a singleton (unicast) remote object whose **references are valid only while the server process is alive.**

REMOTE METHOD INVOCATION (RMI)

- **Locating Remote Objects:** A simple name server is provided for storing named references to remote objects.
- A remote object reference can be stored using the URL-based methods of the class `java.rmi.Naming`.
- For a client to invoke a method on a remote object, that client must first obtain a reference to the object.
- The `java.rmi.Naming` class provides Uniform Resource Locator (URL) based methods to look up, bind, rebind, unbind, and list the name-object pairings maintained on a particular host and port.
- **Stub hides the serialization of parameters and the network-level communication** in order to present a simple invocation mechanism to the caller.
- In the remote JVM, each remote object may have a corresponding skeleton.
- **The skeleton** is responsible for dispatching the call to the actual remote object implementation.

RMI IMPLEMENTATION

Steps to implement RMI:

1. Defining a remote interface
 2. Implementing the remote interface
 3. Creating Stub and Skeleton objects from the implementation class using rmic (rmi compiler)
 4. Start the `rmiregistry`
 5. Create and execute the server application program
 6. Create and execute the client application program.
- Remote interfaces: Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

RMI IMPLEMENTATION

- **Step 1: Defining the remote interface:**
- To create an interface which will provide the description of the methods that can be invoked by remote clients.
- This interface should extend the **Remote** interface and the method prototype within the interface should throw the RemoteException.

```
// Creating a Search interface (Search.java)
import java.rmi.*;
public interface Search extends Remote
{
    // Declaring the method prototype
    public String query(String search) throws
RemoteException;
}
```

RMI IMPLEMENTATION

Step 2: Implementing the remote interface

- To implement the remote interface, the class should extend to `UnicastRemoteObject` class of `java.rmi` package.

```
// Java program to implement the Search interface (SearchQuery.java)
import java.rmi.*;
import java.rmi.server.*;
public class SearchQuery extends UnicastRemoteObject
                                implements Search
{
    // Implementation of the query interface
    public String query(String search)
                                throws RemoteException
    {
        String result;
        if (search.equals("Reflection in Java"))
            result = "Found";
        else
            result = "Not Found";
        return result; } }
```

RMI IMPLEMENTATION

- **Step 3: Creating Stub and Skeleton objects from the implementation class using `rmic`**

The `rmic` tool is used to invoke the rmi compiler that creates the **Stub and Skeleton objects**. Its prototype is `rmic classname`. The command need to be executed at the command prompt

```
# rmic SearchQuery
```

- **STEP 4: Start the `rmiregistry`**
- Start the registry service by issuing the command at the command prompt :

```
# start rmiregistry
```

RMI IMPLEMENTATION

- **STEP 5: Create and execute the server application program**
- To create the server application program and execute it on a separate command prompt.
- The server program uses `createRegistry` method of `LocateRegistry` class to create `rmiregistry` within the server JVM with the port number passed as argument.
- The `rebind` method of `Naming` class is used to bind the remote object to the new name.

RMI IMPLEMENTATION

```
//program for server application (SearchServer.java)
import java.rmi.*;
import java.rmi.registry.*;
public class SearchServer
{
    public static void main(String args[])
    {
        try
        {
            // Create an object of the interface
            // implementation class
            Search obj = new SearchQuery();
            // rmiregistry within the server JVM with port number 1900
            LocateRegistry.createRegistry(1900);
            // Binds the remote object by the name LP-V
            Naming.rebind("rmi://localhost:1900"+
                           "/LP-V",obj);
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

RMI IMPLEMENTATION

Step 6: Create and execute the client application program

- The last step is to create the client application program and execute it on a separate command prompt .
- The lookup method of Naming class is used to get the reference of the Stub object.

```
//program for client application  (ClientRequest.java)
import java.rmi.*;
public class ClientRequest
{
    public static void main(String args[])
    {
        String answer,value= "RMI in Java";
        try
        { // lookup method to find reference of remote object
          Search access =
(Search)Naming.lookup("rmi://localhost:1900/cl9");
          answer = access.query(value);
          System.out.println("Article on " + value +
                                " " + answer+" at
cl9");
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

RMI IMPLEMENTATION

Step 7: Compile and execute application programs:

#Javac SearchQuery.java

#rmic SearchQuery

#rmiregistry on console

On console-1:

Compile Server Application:

#javac SearchServer.java

#java SearchServer

On console-2:

Compile ClientRequest Application:

#Javac ClientRequest.java

#java ClientRequest

REFERENCES

- <https://www.geeksforgeeks.org/remote-method-invocation-in-java/>
- <https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html>

ASSIGNMENT NO. 2

Assignment No. 2

Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).

Objective(s):

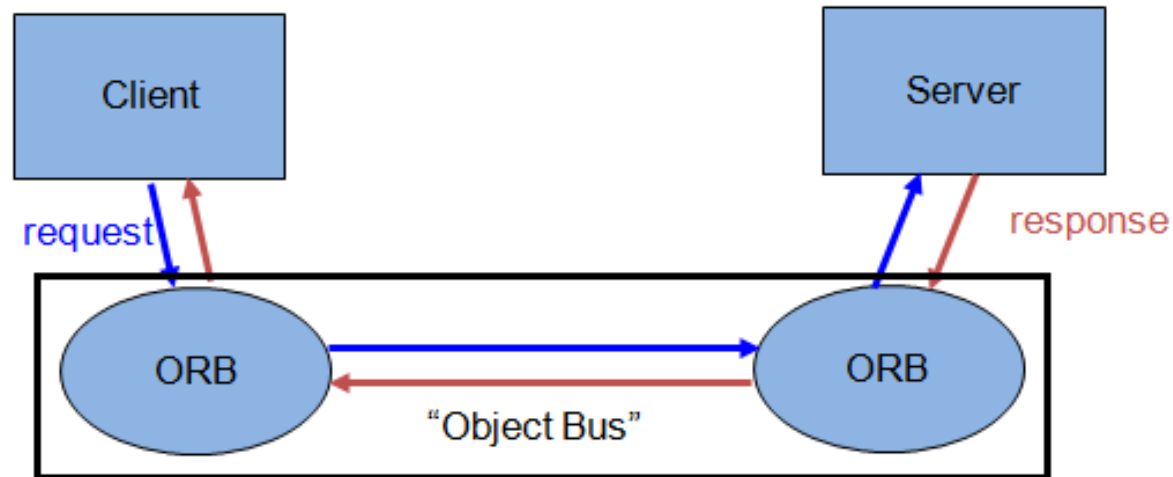
By the end of this assignment, the student will be able to implement any distributed applications based on MPI.

Tools

Java 8 with idlj compiler.

CORBA

- Stands for Common Object Request Broker Architecture.
- It is a specification for creating distributed objects and NOT a programming language.
- It promotes design of applications as **a set of cooperating objects**.
- Clients are isolated from servers by interface.
- CORBA objects run on any platform, can be located anywhere on the network and can be written in any language that has IDL mapping.



CORBA ARCHITECTURE

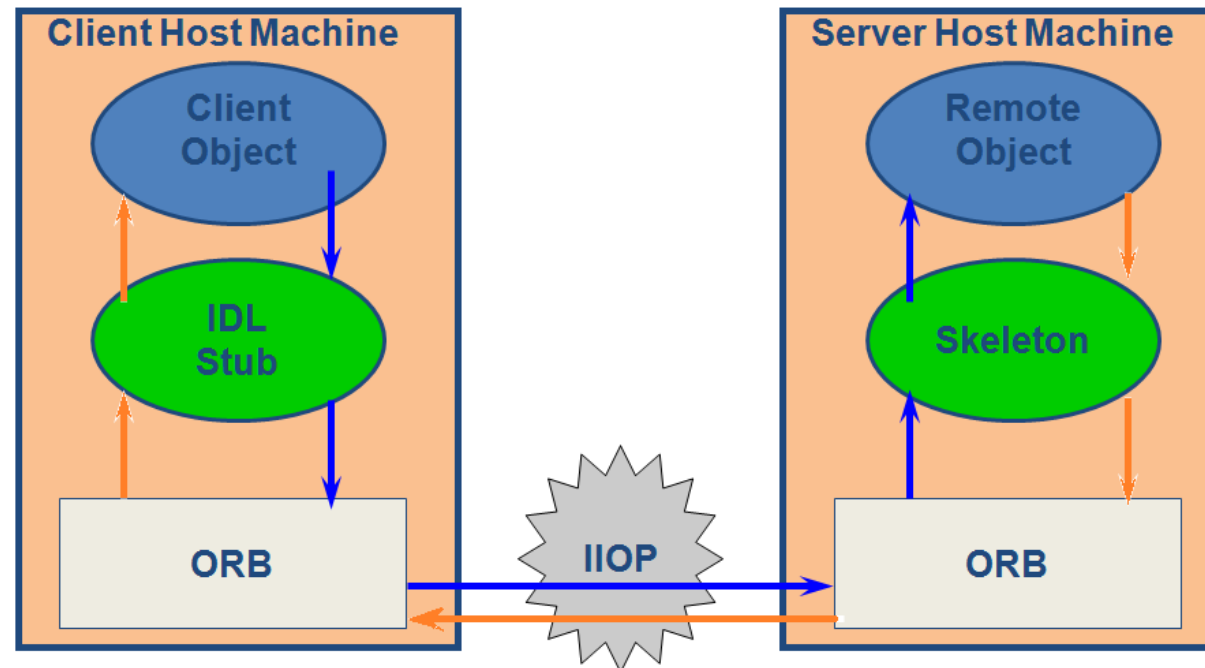
- **Object Request Broker** is an **Object Manager** in CORBA.
- It is present on the client side as well as server side (allows agents to act as both clients and servers of remote objects).
- On **client side** the ORB is **responsible** for
 - accepting requests for a remote object
 - finding implementation of the object
 - accepting client-side reference to the remote object(converted to a language specific form, e.g., a Java stub object)
 - routing client method calls through the object reference to the object implementation.
- On **server side** the ORB
 - lets object servers register new objects
 - receives requests from the client ORB
 - uses object's skeleton interface to invoke object's activation method
 - creates reference for new object and sends it back to client.
- Between the ORBs, **Internet Inter-ORB Protocol** is used for communication.

CORBA

- A CORBA (Common Object Request Broker Architecture) application is developed using IDL (Interface Definition Language).
- IDL is used to define interfaces and the Java IDL compiler generates skeleton code.
- CORBA technology is an integral part of the Java platform. It consists of an Object Request Broker (ORB), APIs for the RMI programming model, and APIs for the IDL programming model.
- The Java CORBA ORB supports both the RMI and IDL programming models.
- We use IDL programming model in this example.

CORBA

- IDL is Interface Definition Language which defines protocol to access objects.
- Stub lives on client and pretends to be remote object.
- Skeleton lives on server , receives requests from stub, talks to true remote object and delivers response to stub.

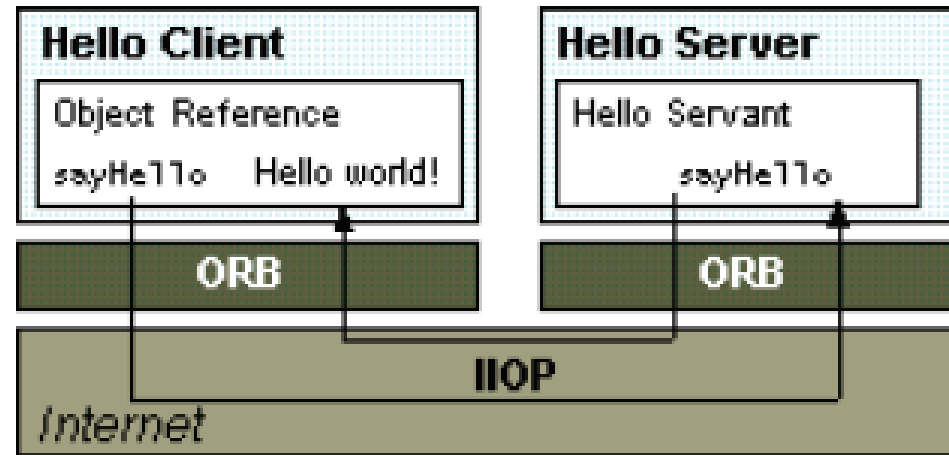


JAVA IDL-USING CORBA FROM JAVA

- Java – IDL is a technology for distributed objects -- that is, objects interacting on different platforms across a network.
- Translates IDL concepts into Java Language Constructs.
- Java IDL supports distributed objects written entirely in the Java programming language.
- Java IDL enables objects to interact regardless of whether they're written in the Java programming language or another language such as C, C++.
- This is possible because Java IDL is based on the Common Object Request Brokerage Architecture (CORBA), an industry-standard distributed object model.
- 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.

JAVA IDL-USING CORBA FROM JAVA

- 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 IIOP.

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL

1. Define the remote interface:

Define the interface for the remote object using Interface Definition Language (IDL).

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.**

1.1 Writing the IDL file:

- J2SDK v1.3.0 above 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 mapping to convert IDL interface definitions to corresponding Java interfaces, classes, and methods, which can then be used to implement the client and server code.

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL

1.1.1 Writing `Hello.idl`

- Create a directory named `Hello` for this application.
- Create a file named `Hello.idl` in this directory.

1.1.2 Understanding the IDL file : Perform 3 steps to write IDL file as follows:

- **Declaring the CORBA IDL module:** When you compile the IDL, the module statement will generate a package statement in the Java code.

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

- **Declaring the interface:** When you compile the IDL, interface statement will generate an interface statement in the Java code.

```
module HelloApp
{
    interface Hello    { // These are the interface //
statement.

                        };

};
```

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL

1.1.2 Understanding the IDL file : Perform 3 steps to write IDL file as follows:

- **Declaring the operations:** Each operation statement in the IDL generates a corresponding method statement in the generated Java interface.
- In the file, enter the code for the interface definition(Hello.idl):

```
module HelloApp
{
interface Hello
{
    string sayHello(); // This line is the operation statement.
};
};
```

.

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL

2. Compile the remote interface: When you run the `idlj` compiler the 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.

2.1. Mapping `Hello.idl` to Java:

- The tool `idlj` reads 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**, use the **`-fall` option** when running the `idlj` compiler.
- Enter compiler command on command line prompt having path to the **`java/bin`** directory:

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

- If you list the contents of the directory, you will see that six files are created.

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL

2.2 Understanding the idlj Compiler Output

- The files generated by the `idlj` compiler for `Hello.idl` are:
 - `HelloPOA.java`** : This abstract **class is the skeleton**, providing basic CORBA functionality for the server. The server class `HelloImpl` extends `HelloPOA`. An *object adapter* is the mechanism that connects a request using an object reference with the proper code to service that request.
 - `_HelloStub.java`** : This class is the **client stub** providing CORBA functionality for the client. It implements the `Hello.java` interface.
 - `Hello.java`** : This interface contains the **Java version of IDL interface**. The `Hello.java` interface extends `org.omg.CORBA.Object`, providing standard CORBA object functionality.

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL

- iv. **HelloHelper.java** : This class provides **additional functionality** , 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.** The Holder class uses the methods in the Helper class for reading and writing.
- v. **HelloHolder.java**: It provides operations for OutputStream and InputStream. It provides operations for `out` and `inout` arguments, which CORBA allows, but which do not map easily to Java's semantics.
- vi. **HelloOperations.java** : This operations interface contains the single methods `SayHello()`. The IDL-to-Java mapping puts all of the operations defined on the IDL interface into this file.

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL

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, the **server code includes a mechanism to start the ORB and wait for invocation from a remote client.**

3.1 Developing the Hello World Server:

- The example server consists of two classes, the **Servant** and the **Server**. The servant, `HelloServant`, is the implementation of the `Hello` IDL interface; each `Hello` instance is implemented by a `HelloServant` instance. The servant is a subclass of `_HelloImplBase`, which is generated by the `idlj` compiler from the example IDL.
- **The servant contains one method for each IDL operation**, in this example, just the `sayHello()` method. **Servant methods are just like ordinary Java methods; extra code to deal with the ORB, with marshaling arguments and results, and so on, is provided by the server and the stubs.**

BUILDING A CORBA DISTRIBUTED APPLICATION USING JAVA IDL : IMPLEMENT THE SERVER

3.1 Developing the Hello World Server

- The **server** class has the server's `main()` method, which:
- Creates an ORB instance.
- 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.
- Registers the new object in the naming context under the name "Hello".
- Waits for invocations of the new object

IMPLEMENT THE SERVER

The steps in writing the CORBA transient Server:

3.1.1. Creating `HelloServer.java`

3.1.2. Understanding `HelloServer.java`

3.1.3. Compiling the Hello World Server

3.1.1 Creating `HelloServer.java`: Enter the following code for `HelloServer.java` in the text file.

IMPLEMENT THE SERVER

3.1.1 Creating HelloServer.java: Enter the following code for HelloServer.java in the text file.

```
// 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.*;

public class HelloServer
{
    public static void main(String args[])
    {
        try{

            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Create the servant and register it with the ORB
            HelloServant helloRef = new HelloServant();
            orb.connect(helloRef);

            // Get the root naming context
```

IMPLEMENT THE SERVER

3.1.1 Creating HelloServer.java: Enter the following code for HelloServer.java in the text file.

```
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);

// Bind the object reference in naming
// Make sure there are no spaces between ""
NameComponent nc = new NameComponent("Hello", "");
NameComponent path[] = {nc};
ncRef.rebind(path, helloRef);

// Wait for invocations from clients
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
    sync.wait();
}

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

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

IMPLEMENT THE SERVER

3.1.2 Understanding `HelloServer.java` implementation

- **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:**

```
// 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.*;
```

IMPLEMENT THE SERVER

3.1.2 Understanding the Server implementation

- **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.
}
```

Save and close `HelloServer.java`.

IMPLEMENT THE SERVER

- **Handling CORBA System Exceptions:**

- 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 an ORB Object:**

- Just like in the client application, a CORBA server also needs a local ORB object.
- 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);
```

IMPLEMENT THE SERVER

- **Managing the Servant Object**
- A server is a process that instantiates one or more servant objects. The servant implements the interface generated by `idlj` and actually performs the work of the operations on that interface. The `HelloServer` needs a `HelloServant`.

`HelloServant helloRef = new HelloServant();`
- **Instantiating the Servant Object:** We instantiate the servant object inside the try-catch block, just below the call to `init()`, as shown:

`orb.connect(helloRef);`

IMPLEMENT THE SERVER

- **Defining the Servant Class**
- At the end of `HelloServer.java`, outside the `HelloServer` class, we define the class for the servant object.

```
class HelloServant extends _HelloImplBase
{
    // The sayHello() method goes here.
}
```

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

```
public String sayHello()
{
    // The method implementation goes here.
    return "\nHello world!!\n";
}
```

IMPLEMENT THE SERVER

- **Working with CORBA Object Service (COS) Naming:**
- The `HelloServer` works with the naming service to make the servant object's operations available to clients. The server needs an object reference to the name service, so that it can register itself and ensure that invocations on the `Hello` interface are routed to its servant object.
- **Obtaining the Initial Naming Context:**
- In the try-catch block, below instantiation of 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.

IMPLEMENT THE SERVER

- **Narrowing the Object Reference:** As with all CORBA object references, `objRef` is a generic CORBA object. To use it as a `NamingContext` object, it must be narrowed to its proper type. The call to `narrow()` is just below the previous statement:

```
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

- Here the `idlj`-generated **helper class** is used.
- **Registering the Servant with the Name Server:**
- Just below the call to `narrow()`, we create a new `NameComponent` member:

```
NameComponent nc = new NameComponent("Hello", "");
```
- This statement sets the `id` field of `nc` to `"Hello"` and the `kind` component to the empty string. There are no spaces between the `""`.
- The `kind` attribute adds descriptive power to names in a syntax-independent way. e.g. `c_source`, `object_code`, `executable`, `postscript`, or `""`.
- Finally, **we pass path and the servant object to the naming service, binding the servant object to the "Hello" id:**

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

IMPLEMENT THE SERVER

- **Waiting for Invocation**
- The following code, which is at the end of the try-catch block, shows how to accomplish this.

```
java.lang.Object sync = new java.lang.Object();  
synchronized(sync) {  
    sync.wait();  
}
```
- This form of `Object.wait()` requires `HelloServer` to remain alive 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.

IMPLEMENT THE SERVER

3.1.3 Compiling the Hello World Server:

- To compile `HelloServer.java`:
- Run the Java compiler on `HelloServer.java`:
`javac HelloServer.java HelloApp/*.java`
- The files `HelloServer.class` and `HelloServant.class` are generated in the `Hello` directory.

IMPLEMENT THE CLIENT

4.1 Developing a Client Application:

- Use the stubs generated by the `idlj` compiler as the basis of 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.**

4.1.1 Creating `HelloClient.java`:

- To create `HelloClient.java`,
 1. Start text editor and create a file named `HelloClient.java` in main project directory, Hello.
 2. Enter the code for `HelloClient.java` in the text file.

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

```
public class HelloClient
{
    public static void main(String args[])
    {
```

IMPLEMENT THE CLIENT

4.1.1 Creating HelloClient.java:

```
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");
    NamingContext ncRef = NamingContextHelper.narrow(objRef);

    // Resolve the object reference in naming
    // make sure there are no spaces between ""
    NameComponent nc = new NameComponent("Hello", "");
    NameComponent path[] = {nc};
    Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

    // Call the Hello server object and print results
    String Hello = helloRef.sayHello();
    System.out.println(Hello);

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

Save and close HelloClient.java.

IMPLEMENT THE CLIENT

4.1.2. Understanding the Client implementation

- **Performing Basic Setup:** The structure of a CORBA client program is the same as most Java applications: You import required library packages, declare the application class, define a `main()` method, and handle exceptions.
- **Importing required packages:**

```
import HelloApp.*;           // The package containing our stubs.  
import org.omg.CosNaming.*;  // HelloClient will use the naming service.  
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.  
}
```

IMPLEMENT THE CLIENT

4.1.2 Understanding the Client implementation

Defining the `main()` Method

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

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

- **Handling CORBA System Exceptions:** 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);
}
```

IMPLEMENT THE CLIENT

- **4.1.2 Understanding the Client implementation**
- **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.
- Just like in the client application, a CORBA server also needs a local ORB object.
- The ORB variable is declared and initialized inside the try-catch block.

```
ORB orb = ORB.init(args, null);
```
- **Finding the Hello Server:** As the application has an ORB, it can ask the ORB to locate the actual service it needs, i.e. Hello server. There are a number of ways for a CORBA client to get an initial object reference; client application will **use the COS Naming Service** provided with Java IDL.

IMPLEMENT THE CLIENT

- **4.1.2 Understanding the Client implementation**
- **Obtaining the Initial Naming Context:** The first step in using the naming service is to get the object reference to the name server which is accomplished by the call `orb.resolve_initial_references()`. `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.
- **Narrowing the Object Reference:** As with all CORBA object references, `objRef` is a generic CORBA object. To use it as a `NamingContext` object, you must narrow it to its proper type.
- `NamingContext ncRef = NamingContextHelper.narrow(objRef);`
- Here we see the use of an `idlj`-generated helper class, `HelloHelper`.

IMPLEMENT THE CLIENT

- **4.1.2 Understanding the Client implementation**
- **Finding a Service in Naming:** CORBA name servers handle complex names by way of `NameComponent` objects.
- An array of `NameComponent` objects can hold a fully specified path to an object on any computer file or disk system.

```
NameComponent nc = new NameComponent("Hello", "");  
NameComponent path[] = {nc};  
Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));
```

- We pass `path` to the naming service's `resolve()` method to get an object reference to the `Hello` server and narrow it to a `Hello` object.
- The `resolve()` method returns a generic CORBA object, `HelloHelper` immediately narrow it to a `Hello` object.

IMPLEMENT THE CLIENT

- **4.1.2 Understanding the Client implementation**
- **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 call to the server method are completely transparent to the client programmer and is done by generated code,
- Invocation is done as follows in CORBA programming:
`String Hello = helloRef.sayHello();`
- Finally, print the results of the invocation to standard output:
`System.out.println(Hello);`

IMPLEMENT THE CLIENT

- **4.1.2 Understanding the Client implementation**
- **Compiling HelloClient.java:**
- Run the Java compiler on `HelloClient.java`:
- `javac HelloClient.java HelloApp/*.java`
- The `HelloClient.class` is generated to the `Hello` directory.

RUNNING THE HELLO WORLD APPLICATION

- To run this client-server application on the machine:
 1. Start the Java IDL Name Server. To do this from a UNIX command shell, enter:
`tnameserv -ORBInitialPort 1050&`
If the `nameserverport` is not specified, port 900 will be chosen by default.
 2. From a second prompt or shell, start the Hello server(Unix):
`java HelloServer -ORBInitialPort 1050&`
 3. From a third prompt or shell, run the Hello application client:
`java HelloClient -ORBInitialPort 1050`
 4. The client prints the string from the server to the command line:
`Hello world!!`

Implementation Details:String Reverse

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

- Creates an ORB instance

```
// initialize the ORB
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
```

- Gets a reference to the root POA and activates the POAManager

```
// initialize the BOA/POA
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
rootPOA.the_POAManager().activate();
```

- Creates a servant instance (the implementation of one CORBA Reverse object) and tells the ORB about it
- Gets a CORBA object reference for a naming context in which to register the new CORBA object

```
// creating the object
ReverseImpl rvr = new ReverseImpl();

// get the object reference from the servant class
org.omg.CORBA.Object ref = rootPOA.servant_to_reference(rvr);
```

- Registers the new object in the naming context under the name “Reverse”
- Waits for invocations of the new object.

Implementation Details:String Reverse

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

- Gets the root naming context.

```
System.out.println("Step1");
Reverse h_ref = ReverseModule.ReverseHelper.narrow(ref);
System.out.println("Step2");

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

- Registers the new object in the naming context under the name “Reverse”

```
System.out.println("Step3");
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
System.out.println("Step4");

String name = "Reverse";
NameComponent path[] = ncRef.to_name(name);
ncRef.rebind(path,h_ref);
```

- Waits for invocations of the new object

```
System.out.println("Reverse Server reading and waiting....");
orb.run();
```

- Save and Close ReverseServer.java.

Implementation Details:String Reverse

- Create ReverseImpl.java containing the logic to reverse a string in the reverse_string() method and return it.

```
import ReverseModule.ReversePOA;
import java.lang.String;
class ReverseImpl extends ReversePOA
{
    ReverseImpl()
    {
        super();
        System.out.println("Reverse Object Created");
    }

    public String reverse_string(String name)
    {
        StringBuffer str=new StringBuffer(name);
        str.reverse();
        return (("Server Send "+str));
    }
}
```


Implementation Details:String Reverse

- Create ReverseClient.java containing code to initialize ORB and get the string to be reversed from the user.
- The code looks up “Reverse” in the naming context and receives a reference to that CORBA object.

```
// initialize the ORB
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

String name = "Reverse";
ReverseImpl = ReverseHelper.narrow(ncRef.resolve_str(name));

System.out.println("Enter String=");
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str= br.readLine();

String tempStr= ReverseImpl.reverse_string(str);

System.out.println(tempStr);
```

Implementation Details:String Reverse

- Create ReverseServer.java
- The ReverseServer class does the following:
- Creates and initializes an ORB instance

```
// initialize the ORB
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
```

- Gets a reference to the root POA and activates the POAManager

```
// initialize the BOA/POA
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
rootPOA.the_POAManager().activate();
```

- Creates a servant instance (the implementation of one CORBA Reverse object) and tells the ORB about it.
- Gets a CORBA object reference for a naming context in which to register the new CORBA object

```
// creating the object
ReverseImpl rvr = new ReverseImpl();

// get the object reference from the servant class
org.omg.CORBA.Object ref = rootPOA.servant_to_reference(rvr);
```

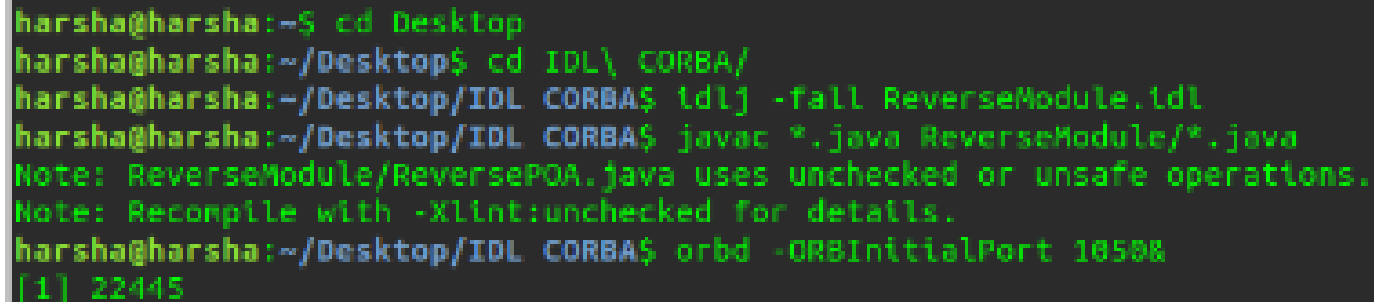
Implementation Details:String Reverse

- Compile the **.java files**, including the stubs and skeletons (which are in the directory newly created directory). This step assumes the java/bin directory is included in your path.

```
javac *.java ReverseModule/*.java
```

- Orbd – Object Request Broker Daemon
- **orbd** is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment.
- Start orbd. To start orbd from a UNIX command shell, enter :

```
orbd -ORBInitialPort 1050&
```



```
harsha@harsha:~$ cd Desktop
harsha@harsha:~/Desktop$ cd IDL\ CORBA\
harsha@harsha:~/Desktop/IDL CORBA$ idlj -fall ReverseModule.idl
harsha@harsha:~/Desktop/IDL CORBA$ javac *.java ReverseModule/*.java
Note: ReverseModule/ReversePOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
harsha@harsha:~/Desktop/IDL CORBA$ orbd -ORBInitialPort 1050&
[1] 22445
```

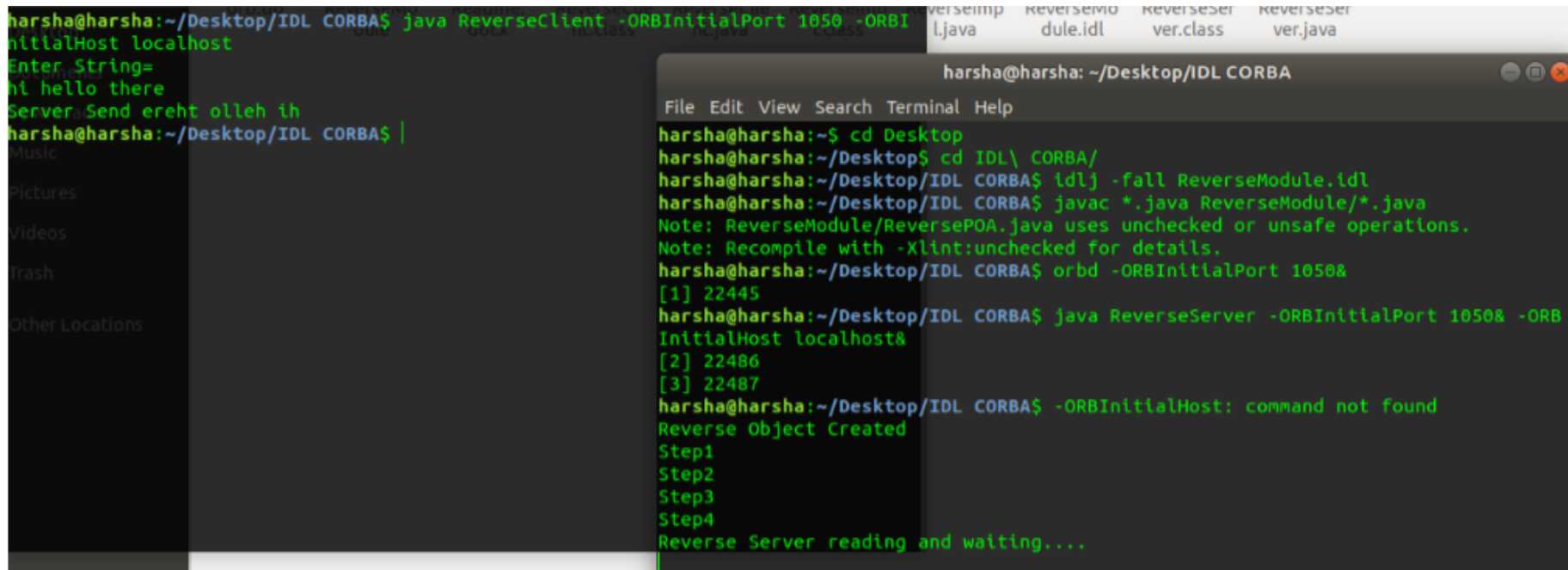
Implementation Details:String Reverse

- Start the server. To start the server from a UNIX command shell, enter :
- `java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&`

```
harsha@harsha:~$ cd Desktop
harsha@harsha:~/Desktop$ cd IDL\ CORBA/
harsha@harsha:~/Desktop/IDL CORBA$ idlj -fall ReverseModule.idl
harsha@harsha:~/Desktop/IDL CORBA$ javac *.java ReverseModule/*.java
Note: ReverseModule/ReversePOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
harsha@harsha:~/Desktop/IDL CORBA$ orbd -ORBInitialPort 1050&
[1] 22445
harsha@harsha:~/Desktop/IDL CORBA$ java ReverseServer -ORBInitialPort 1050& -ORB
InitialHost localhost&
[2] 22486
[3] 22487
harsha@harsha:~/Desktop/IDL CORBA$ -ORBInitialHost: command not found
Reverse Object Created
Step1
Step2
Step3
Step4
Reverse Server reading and waiting....
```

Expected Output

- Start the client. To start the server from a UNIX command shell, enter :
- `java ReverseClient -ORBInitialPort 1050& -ORBInitialHost localhost`



The screenshot shows a terminal window with the following commands and output:

```
harsha@harsha:~/Desktop/IDL CORBA$ java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost
Enter String=
hi hello there
Server: Send ereht olleh ih
harsha@harsha:~/Desktop/IDL CORBA$
```

Below this, a separate terminal window titled "harsha@harsha: ~/Desktop/IDL CORBA" shows the compilation and execution of the server:

```
harsha@harsha:~$ cd Desktop
harsha@harsha:~/Desktop$ cd IDL\ CORBA/
harsha@harsha:~/Desktop/IDL CORBA$ idlj -fall ReverseModule.idl
harsha@harsha:~/Desktop/IDL CORBA$ javac *.java ReverseModule/*.java
Note: ReverseModule/ReversePOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
harsha@harsha:~/Desktop/IDL CORBA$ orbd -ORBInitialPort 1050&
[1] 22445
harsha@harsha:~/Desktop/IDL CORBA$ java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&
[2] 22486
[3] 22487
harsha@harsha:~/Desktop/IDL CORBA$ -ORBInitialHost: command not found
Reverse Object Created
Step1
Step2
Step3
Step4
Reverse Server reading and waiting....
```

References

- **Theory for CORBA -**
<https://docs.oracle.com/javase/7/docs/technotes/guides/idl/corba.html>
- **A 'Hello World' Program using CORBA -**
<https://docs.oracle.com/javase/6/docs/technotes/guides/idl/jidlExample.html>
- Official CORBA website - http://www.corba.org/omg_idl.htm.
- **Java IDL Tutorial and Guide**
https://paginas.fe.up.pt/~nflores/dokuwiki/lib/exe/fetch.php?media=teaching:0809:java_20tutorial_20and_20guide.pdf

THANK YOU

