Assignment No. 2

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

Objective: Demonstration of object brokering using CORBA in Java.

Infrastructure:

Software used: Java, JDK 1.8, IDE like Eclipse(optional)

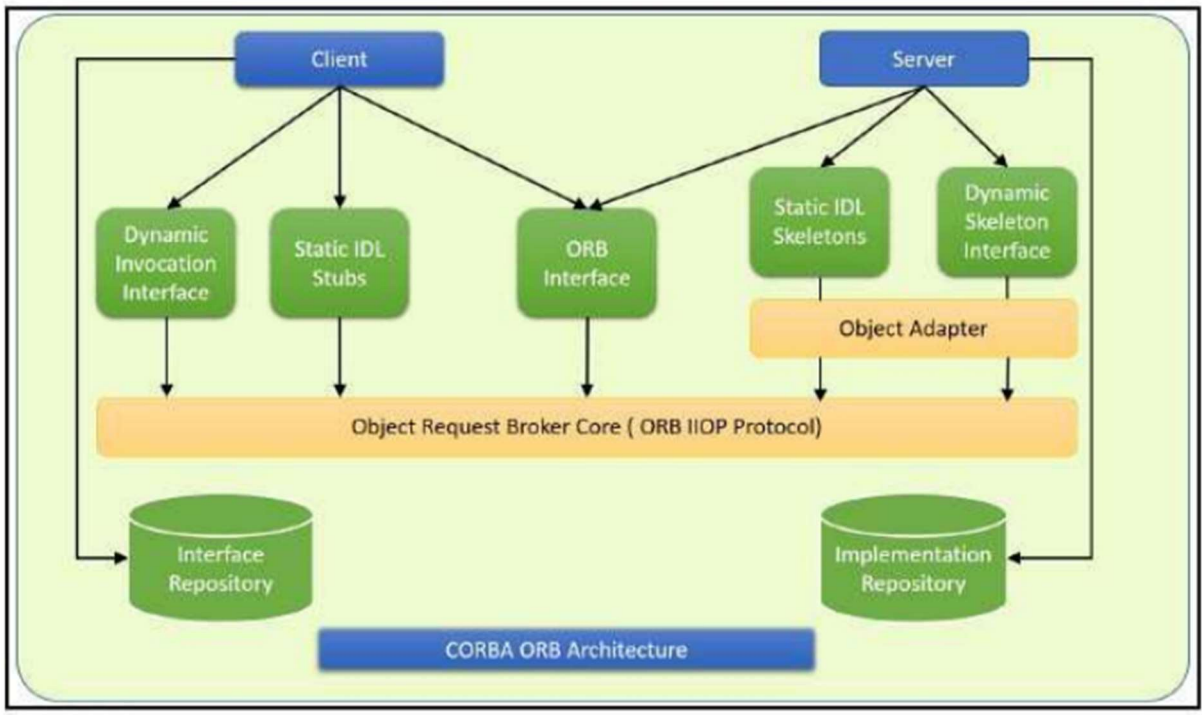
Theory:

Common Object Request Broker Architecture (CORBA):
CORBA is an acronym for Common Object Request Broker Architecture, is an open source, vendor-independent architecture and infrastructure developed by the Object Management Group (OMG) to integrate enterprise applications across a distributed network. CORBA specifications provide guidelines for such integration applications, based on the way they want to interact, irrespective of the technology; hence, all kinds of technologies can implement these standards using their own technical implementations.

Why CORBA is used?
When two applications/systems in a distributed environment interact with each other, there are quite a few unknowns between those applications/systems, including the technology they are developed in (such as Java/ PHP/ .NET), the base operating system they are running on (such as Windows/Linux), or system configuration (such as memory allocation). They communicate mostly with the help of each other's network address or through a naming service. Due to this, these applications end up with quite a few issues in integration, including content (message) mapping mismatches.
An application developed based on CORBA standards with standard Internet Inter-ORB Protocol (IIOP), irrespective of the vendor that develops it, should be able to smoothly integrate and operate with another application developed based on CORBA standards through the same or different vendor.

The contract between these applications is defined in terms of an interface for the server objects that the clients can call. This IDL interface is used by each client to indicate when they should call any particular method to marshal (read and send the arguments).

The IDL interface is a design concept that works with multiple programming languages including C, C++, Java, Ruby, Python, and IDLscript. This is close to writing a program to an interface. The interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.



The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram.

The client and server use stubs and skeletons as proxies, respectively. The IDL interface follows a strict definition, and even though the client and server are implemented in different technologies, they should integrate smoothly with the interface definition strictly implemented.

In CORBA, each object instance acquires an object reference for itself with the electronic token identifier. Client invocations are going to use these object references that have the ability to figure out which ORB instance they are supposed to interact with. The stub and skeleton represent the client and server, respectively, to their counterparts. They help establish this communication through ORB and pass the arguments to the right method and its instance during the invocation.

Java Support for CORBA

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency; Java provides the implementation transparency. An Object Request Broker (ORB) is part of the Java Platform.

When using the IDL programming model, the interface is everything! It defines the points of entry that can be called from a remote process, such as the types of arguments the called procedure will accept, or the value/output parameter of information returned. Using IDL, the programmer can make the entry points and data types that pass between communicating processes act like a standard language.

CORBA is a language-neutral system in which the argument values or return values are limited to what can be represented in the involved implementation languages. In CORBA, object orientation is limited only to objects that can be passed by reference (the object code itself cannot be passed from machine-to-machine) or are predefined in the overall framework. Passed and returned types must be those declared in the interface.

The IDL Programming Model:

The IDL programming model, known as Java™ IDL, consists of both the Java CORBA ORB and the idlj compiler that maps the IDL to Java bindings that use the Java CORBA ORB, as well as a set of APIs, which can be explored by selecting the org.omg prefix from the Package section of the API index.

Java IDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Runtime components include a Java ORB for distributed computing using IIOP communication.

To use the IDL programming model, define remote interfaces using OMG Interface Definition Language (IDL), then compile the interfaces using idlj compiler. When you run this, it generates the Java version of the interface, as well as the class code files for the stubs and skeletons that enable applications to hook into the ORB.

Portable Object Adapter (POA): An object adapter is the mechanism that connects a request using an object reference with the proper code to service that request. The Portable Object Adapter, or POA, is a particular type of object adapter that is defined by the CORBA specification. The POA is designed to meet the following goals:

• Allow programmers to construct object implementations that are portable between different ORB products.

• Provide support for objects with persistent identities.

Designing the solution:

1. Creating CORBA Objects using Java IDL:

1.1. First one has to define CORBA-enabled interface and it implementation, which involves:

- Writing an interface in the CORBA Interface Definition Language
- Generating a Java base interface, plus a Java stub and skeleton class, using an IDL to-Java compiler
- Writing a server-side implementation of the Java interface in Java

Interfaces in IDL are declared much like interfaces in Java.

1.2. Modules

Modules are declared in IDL using the module keyword. Everything defined within the scope of this module (interfaces, constants, other modules) falls within the module and is referenced in other

IDL modules using the syntax modulename::x. e.g.

```
// IDL
module jen {
module corba {
interface NeatExample ...
};
};
```

1.3. Interfaces

It involves declaration of interface which includes an interface header and an interface body. For example:

```
interface PrintServer : Server { ...
```

1.4 Data members and methods

The interface body declares all the data members (or attributes) and methods of an interface. Data members are declared using the attribute keyword. At a minimum, the declaration includes a name and a type.

```
readonly attribute string myString;
```

The method can be declared by specifying its name, return type, and parameters, at a minimum.

```
string parseString(in string buffer);
```

This declares a method called parseString() that accepts a single string argument and returns a string value.

1.5 A complete IDL example

Now let's tie all these basic elements together. Here's a complete IDL example that declares a module within another module, which itself contains several interfaces:

```
module OS {
module services {
interface Server {
readonly attribute string serverName;
boolean init(in string sName);
};
interface Printable {
boolean print(in string header);
};
interface PrintServer : Server {
boolean printThis(in Printable p);
};
};
};
```

2. Turning IDL Into Java

Now for implementing the remote interfaces in Java we use an IDL-to-Java compiler. Every standard IDL-to-Java compiler generates the following 3 Java classes from an

IDL interface:

- A Java interface with the same name as the IDL interface. This can act as the basis for a Java implementation of the interface (but you have to write it, since IDL doesn't provide any details about method implementations).
- A helper class whose name is the name of the IDL interface with "Helper" appended to it
- A holder class whose name is the name of the IDL interface with "Holder" appended to it

The idltoj tool generate 2 other classes:

- A client stub class, called _interface-nameStub, that acts as a client-side implementation of the interface and knows how to convert method requests into ORB requests that are forwarded to the actual remote object. The stub class for an interface named Server is called _ServerStub.
- A server skeleton class, called _interface-nameImplBase, that is a base class for a serverside implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named Server is called _ServerImplBase.

So, in addition to generating a Java mapping of the IDL interface and some helper classes for the Java interface, the idltoj compiler also creates subclasses that act as an interface between a CORBA client and the ORB and between the server-side implementation and the ORB.

This creates the five Java classes: a Java version of the interface, a helper class, a holder class, a client stub, and a server skeleton.

3. Writing the Implementation

The IDL interface is written and generated the Java interface and support classes for it, including the client stub and the server skeleton.

Implementing the solution:

Now create a directory named hello/ where you develop sample applications and create the files in this directory.

1. 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). To complete the application,

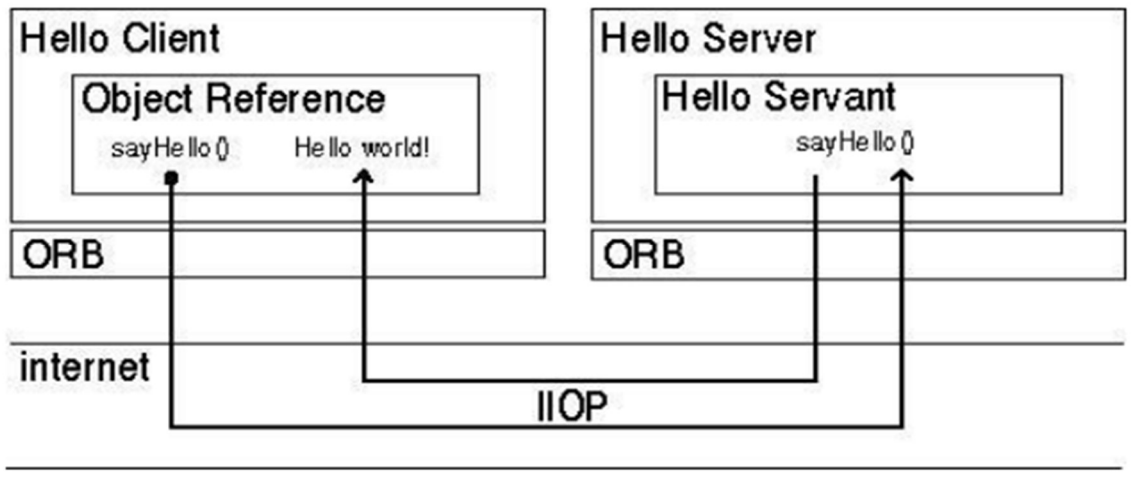
you simply provide the server (HelloServer.java) and client (HelloClient.java) implementations.

2. Implementing the Server (HelloServer.java)

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.



3. 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.

Building and executing the solution:

Now we use 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 with Java include orbd, a daemon process containing a Bootstrap Service, a Transient Naming Service,

To run this client-server application on the 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 serverside bindings.

The files generated by the idlj compiler for Hello.idl, with the -fall command line option, are:

- HelloPOA.java
- _HelloStub.java
- Hello.java
- HelloHelper.java
- HelloHolder.java
- HelloOperations.java

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

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

4. Start orbd.

To start orbd from a UNIX command shell,

enter: orbd -ORBInitialPort 1050&

5. Start the HelloServer:

To start the HelloServer from a UNIX command shell, enter:

java HelloServer -ORBInitialPort 1050 -

ORBInitialHost localhost&

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

6. Run the client application:

java HelloClient -ORBInitialPort 1050 -

ORBInitialHost localhost

Conclusion:

CORBA provides the network transparency; Java provides the implementation transparency. CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment. The combination of Java and CORBA allows you to build more scalable and more capable applications than can be built using the JDK alone.