Page :
Date : / /

Assignment No. - 02

- **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)

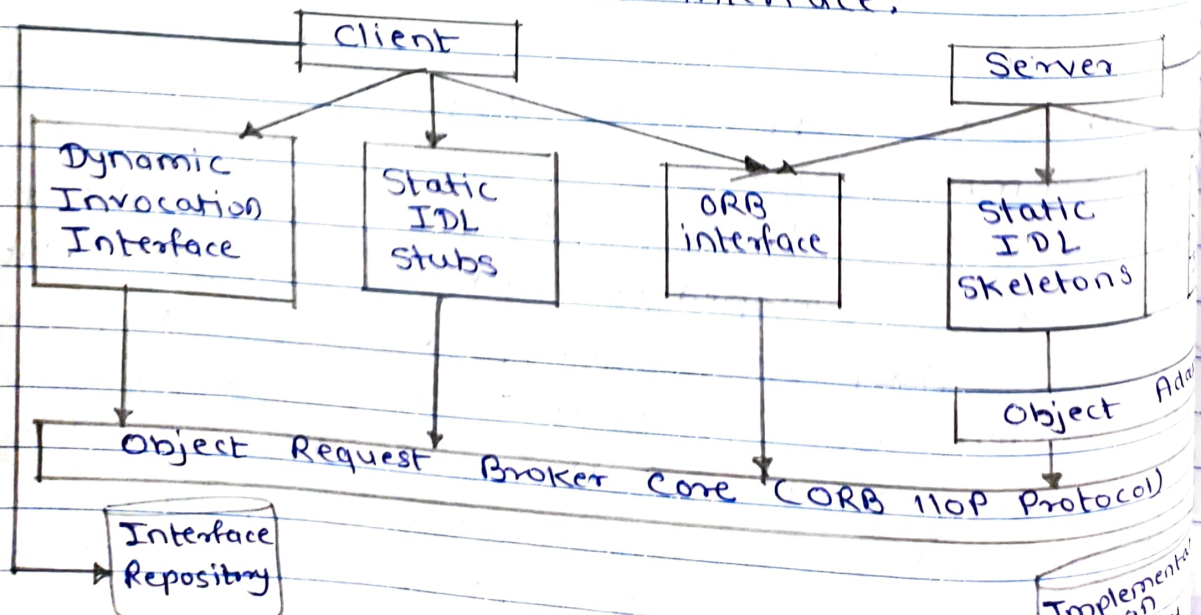
- **Theory:-**
- **Common Object Request Broker Architecture (CORBA)**
  CORBA is an open source, vendor-independent archi. & infrastructure developed by Object Management Group (OMG) to integrate enterprise applications across a distributed network. All kinds of technologies can implement these standards using their own technical implementations.

- **Why CORBA is used?**
  When two applications / systems in a distributed environment interact with each other, there are a few unknown bet? those systems, including the technology they are developed in (such as Java / PHP / .NET), the base OS they are running on (such as Windows/Linux) or system configuration (such as memory allocation). They communicate mostly with the help of each other network address or thro' a naming service.
  An application developed based on CORBA standards

standard Internet Inter-ORB Protocol (IIOP) of vendor that develops it, should be able to integrate & operate with another application based on CORBA Standards thro' the same or

Except legacy applications, most of applica follow common standards when it comes to modeling, for eg, applications similar to "HR & B maintain an object model with details of orga employees benefits details. They are only differ in the way they handle details, based on co & region they are operating for. For each obje Similar to HR & Benefits systems, we can define interface using Interface Definition Language (

The contract betn these applications is in terms of an interface for server objects th can call. IDL interface is a design concept that with multiple programming languages like C, Ruby, Python, & IDLscript. The systems encapsulate implementation along with their respective da handling & processing, & only methods are ava to rest of worlds thro' 'interface.



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

In CORBA, each object instance acquires an object reference for itself with electronic token identifier. The stub & skeleton represent client & server, respectively, to their counterparts. They help establish this communication thro' ORB & pass arguments to right method.

- Java Support for CORBA:

CORBA complements Java™ platform by providing a distributed object framework, & interoperability with other languages. The Java platform complements CORBA by providing portable, highly productive implementation environment.

CORBA standards provide proven, interoperable infrastructure to Java platform. IIOP (Internet Inter-ORB Protocol) manages comm. betn object components that power system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides network transparency. An ORB (Object Request Broker) is part of Java Platform. Java IDL included both a Java-based ORB, which supported IIOP & IDL-to-Java compiler, for generating client-side stub & server-side code skeletons.

CORBA is a language-neutral system in which the argument values or return values are limited to what can be represented in involved implementation languages. With RMI, the interface & implementation language are described in same lang, so you don't have to worry about mapping from one to the other.

Language-level objects (code itself) can be passed from one process to next. Values can be returned by their actual type, not declared type. IIOP stubs & skeletons which allow your objects to be accessible from any CORBA-compliant languages.

The IDL Programming Model :-

The IDL prog. model, known as Java™ IDL, is a part of both Java CORBA ORB & idlj compiler that maps OMG IDL to Java bindings that use Java CORBA ORB. It consists of API's, which can be explored by selecting org.omg.CORBA from Package section of API index.

To use IDL prog. model, define remote interfaces using OMG Interface Definition Language (IDL) & then compile interfaces using idlj compiler. When you run idlj it generates Java version of interface, as client & server files for stubs & skeletons that enable application to plug into ORB.

Designing the Solution :-

Here the design of how to create a complete application using IDL to define interfaces & Java compiler to generates stub & skeletons.

The server-side implementation generated by idlj compiler is Portable Servant Inheritance Model, also known as POA (Portable Object Adapter) model. This presents a sample application created using default behavior of idlj compiler, which uses a POA server side model.

1) Creating CORBA Objects using Java IDL :-

In order to distribute a Java object over network using CORBA, one has to define its own CORBA-enabled interface & its implementation. This involves doing the follo:-

- Writing an interface in CORBA interface Definition Lang.
- Generating Java base interface, plus a Java stub & skeleton class, using an IDL to Java compiler.
- Writing a server-side implementation of Java interface in Java.

Modules :-

Modules are declared in IDL using module keyword, followed by name for module & openning brace that starts module scope. Everything defined within scope of this module (interfaces, constants, other modules) falls within module & is referenced in other IDL modules using syntax modulename::x.

e.g:- //IDL

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

Interfaces :-

The declaration of an interface includes an interface header & an interface body. The header specifies name of the interface & interfaces it inherits from (if any). Here is an IDL interface header.

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

This header starts declaration of an interface called PrintServer that inherits all the methods & data members from the Server interface.

Data members & methods :-

The interface body declares all data mem- methods of an interface. Data members are d- using attribute keyword. At a mini., declaration a name & a type.

readonly attribute string myString;

The method can be declared by specifying its type, & parameters, at a minimum.

String parseString (in string buffer);

This declares a method called parseString () that single string argument & returns a single val-

---

A complete IDL example :-

Here's complete IDL e.g that declares a modul another module, which itself contains several i-

```
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:-

Once remote interfaces in IDL are described, you need to generate Java classes that act as starting point for implementing those remote interfaces in Java using an IDL-to-Java compiler.

IDL interface:-

- A Java interface with same name as IDL interface. This can act as basis for Java implementation of interface
- A helper class whose name is the name of IDL interface with "Helper" appended to it.
- A holder class whose name is the name of 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 interface & knows how to convert method requests into ORB requests that are forwarded to actual remote object. The stub class for 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.

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

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

3 Writing the implementation:-

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

Implementing the Soln :-

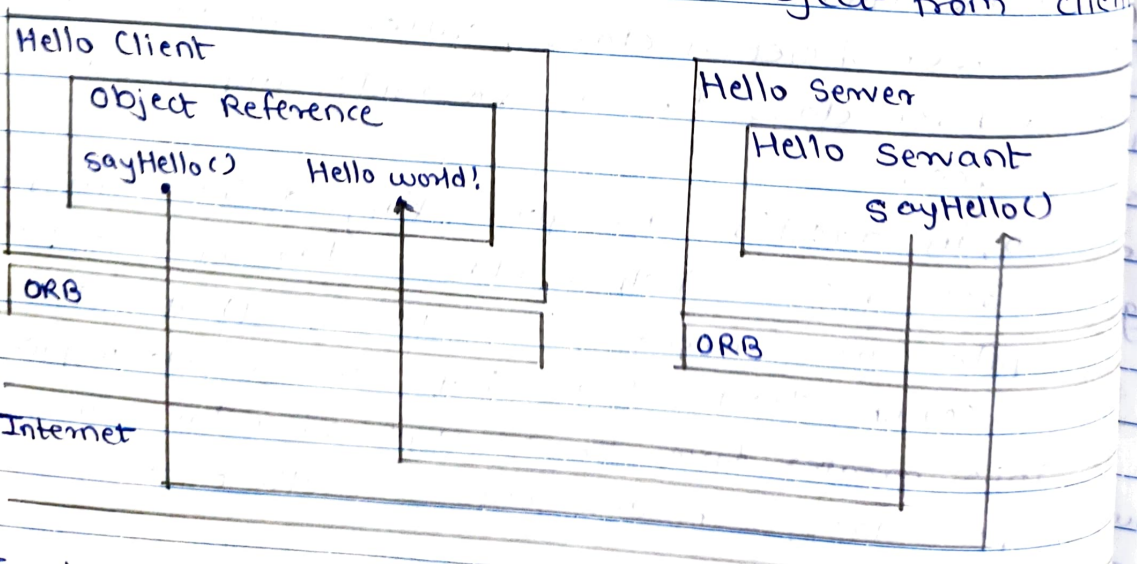
Here, we are demonstrating "Hello World" Exam
create this, create directory named hello/ where
sample applications & create files in this direc

1) Defining Interface (Hello.idl)

The first step to creating a CORBA applica
specify all your objects & their interfaces usin
Interface Definition lang. (IDL). To complete appli
Server (HelloServer.java) & client (HelloClient.java) imple

2) Implementing Server (HelloServer.java)

The HelloServer class has Server's main() met
- Creates & initializes an ORB instance
- Gets a reference to root POA & activates POAMana
- Creates a servant instance & tells the ORB about
- Gets a CORBA object reference for a naming
which to register new CORBA object.
- Gets root naming context.
- Registers new object in naming context under nam
- Waits for invocations of new object from client



Implementing Client (HelloClient.java)

The e.g application client

- obtains a reference to root naming context.
- Looks up "Hello" in naming context & receives a reference to that CORBA object.
- Invokes object's sayHello() & shutdown() operations & prints results

- **Building & executing the soln:-**

The Hello World program lets you learn & experiment with all tasks required to develop any CORBA program that uses static invocation & is used when interface of object is known at compile time.

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 object reference. The Two options for Naming Services with Java include orbd, daemon process containing a Bootstrap Service, a Transient Naming Service,

To run this client-server application on development machine:-

1) Change to directory that contains file Hello.idl.
2) Run IDL-to-Java compiler, idlj, on IDL file to create stubs & skeletons.

This step assumes that you have included path to java/bin directory in your path.

idlj -fall Hello.idl

You must use -fall option with idlj compiler to generate both client & server side bindings. This command line will generate defaut server-side bindings, which assumes POA Inheritance server-side model.

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

- **HelloPOA.java:-**

This abstract class is stream-based server providing basic CORBA functionality for server org.omg.PortableServer.Servant, & implements In interface & HelloOperations interface. The Server HelloImpl extends HelloPOA.

- **HelloStub.java :-**

This class is the client stub, providing CO onality for client. It extends org.omg.CORBA. ObjectImpl & implements the Hello.java interface

- **Hello.java :-**

This interface contains Java version of IDL written. Hello.java interface extends org.omg.CORB providing standard CORBA object functionality. extends HelloOperations interface & org.omg.CORBA. IDLEntity.

- **HelloHelper.java :-**

This class provides auxiliary functionality, na narrow() method required to cast CORBA object to their proper types. The Helper class is res reading & writing data type to CORBA streams & extracting data type from Anys. The Holder cla to methods in the Helper class for reading & w

- **HelloHolder.java :-**

This final class holds a public instance mem type Hello. Whenever IDL type is an out or an parameter, the Holder class used ons for

• HelloOperations.java :-

This interface contains methods sayHello() & shutdown() The IDL-to-Java mapping puts all of operations defined on IDL interface into this file, which is shared by both stubs and skeletons.

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

javac *.java HelloApp/*.java

4) Start orbd

To start orbd from a UNIX command shell, enter: orbd -ORBInitialPort 1050 &

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

5) Start HelloServer :-

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

java HelloServer -ORBInitialPort 1050 -
ORBInitialHost localhost &

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

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...

Note :- After completion kill the name server (orbd)

Conclusion:-

CORBA provides the network transparency provides the implementation transparency. C complements the Java™ platform by providing distributed object framework, services to sup framework, and interoperability with other The Java platform complements CORBA by p portable, highly productive implementation en The combination of Java & CORBA allows yo build more scalable & more capable applic than can be built using the JDK alone.

NAME: PALLAVI K. CHOPADE
ROLL NO.: 14
PRN NO.: 72036169K
BE (IT)
LP - V

#CalcApp.idl

```
module CalcApp
{
interface Calc
        {
exception DivisionByZero {};
float sum(in float a, in float b);
float div(in float a, in float b) raises (DivisionByZero);
float mul(in float a, in float b);
float sub(in float a, in float b);
};
};
```

#CalcServer.JAVA

```
import CalcApp.*;
 import CalcApp.CalcPackage.DivisionByZero;
         import org.omg.CosNaming.*;
 import org.omg.CosNaming.NamingContextPackage.*;
 import org.omg.CORBA.*;
 import org.omg.PortableServer.*;
 import java.util.Properties;

         class CalcImpl extends CalcPOA {
             @Override
public float sum(float a, float b) {
         return a + b;
         }

         @Override public float div(float a, float b) throws DivisionByZero {
         if (b == 0) {
             throw new CalcApp.CalcPackage.DivisionByZero();
             } else {
             return a / b;
             }
         }

         @Override
public float mul(float a, float b) {
         return a * b;
         }

        @Override
public float sub(float a, float b) {
         return a - b;
         }
private ORB orb;

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

         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
            CalcImpl helloImpl = new CalcImpl();
            helloImpl.setORB(orb);

            // get object reference from the servant
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
            Calc href = CalcHelper.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 = "Calc";
           NameComponent path[] = ncRef.to_name(name);
           ncRef.rebind(path, href);
           System.out.println("Ready..");
           // wait for invocations from clients
           orb.run();
           } catch (Exception e) {
           System.err.println("ERROR: " + e);
           e.printStackTrace(System.out);
       }
       System.out.println("Exiting ...");
        }
        }
```

#CalcClient.JAVA

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import CalcApp.*;
import CalcApp.CalcPackage.DivisionByZero;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import static java.lang.System.out;

        public class CalcClient {

        static Calc calcImpl;
static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        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 = "Calc";
            calcImpl = CalcHelper.narrow(ncRef.resolve_str(name));

            // System.out.println(calcImpl);
```

```
    while (true) {
        out.println("1. Sum");
        out.println("2. Sub");
        out.println("3. Mul");
        out.println("4. Div");
        out.println("5. exit");
        out.println("--");
        out.println("choice: ");

        try {
            String opt = br.readLine();
            if (opt.equals("5")) {
                break;
                } else if (opt.equals("1")) {
                out.println("a+b= " + calcImpl.sum(getFloat("a"),
                getFloat("b")));
                } else if (opt.equals("2")) {
                out.println("a-b= " + calcImpl.sub(getFloat("a"),
                getFloat("b")));
                } else if (opt.equals("3")) {
                out.println("a*b= " + calcImpl.mul(getFloat("a"),
                getFloat("b")));
                } else if (opt.equals("4")) {
                try {
                    out.println("a/b= " + calcImpl.div(getFloat("a"),
                            getFloat("b")));
                    } catch (DivisionByZero de) {
                    out.println("Division by zero!!!");
                    }
                }
            } catch (Exception e) {
            out.println("===");
            out.println("Error with numbers");
            out.println("===");
            }
        out.println("");

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

static float getFloat(String number) throws Exception {
out.print(number + ": ");
return Float.parseFloat(br.readLine());
}
```