AISSMS
Institute of Information Technology

SAVITRIBAI PHULE PUNE UNIVERSITY
सावित्रीबाई फुले पुणे विद्यापीठ
॥ यः क्रियावान् स पण्डितः॥

WELCOME

WORKSHOP ON
COMPUTER LABORATORY-IX
OF
BE IT 2015 COURSE (SEMESTER II)
BY
SHYAM DESHMUKH

17 JANUARY 2019

OBJECTIVES

1. To understand the learning outcomes and depth of the revised subject contents.
2. To discuss the effective subject delivery technique during the semester.
3. To identify the lab assignments contents and the reference book mapping that helps during the formal teaching of the subject.
4. To share the study material, prepare lab manual, practical exam conduction guidelines.
5. To provide uniform guidelines towards conducting practicals across university.

SCHEDULE

Time	Activity
10:15 am - 11:15 am	Session-I:
	Overview on Required background for All assignments.
11:30 am to 1:00 pm	Session-II
	Demonstration of Assignments + Hands-on Session
1:45 pm to 4:00 pm	Hands-On Session
	Practice with provided code and related material.

SL-V (2012 COURSE)

1. Design a distributed application using RMI for remote computation where client submits two strings to the server and server returns the concatenation of the given strings.
2. Design a distributed application using RPC for remote computation where client submits an integer value to the server and server calculates factorial and returns the result to the client program.
3. Design a distributed application using Message Passing Interface (MPI) for remote computation where client submits a string to the server and server returns the reverse of it to the client.
4. Design a distributed application which consist of a server and client using threads.
5. Design a distributed application which consists of an agent program that program travels in the network and performs a given task on the targeted machine. You may assign any task to the agent e.g. to carry out the existing file opening and reading number of vowels present in that file.
6. Design a distributed application using **MapReduce** which processes a log file of a system. List out the users who have logged for maximum period on the system. Use simple log file from the Internet and process it using a pseudo distribution mode on Hadoop platform.
7. Design and develop a distributed application to find the coolest/hottest year from the available weather data. Use weather data from the Internet and process it using **MapReduce**.
8. Design and develop a distributed Hotel booking application using Java RMI.

WHAT'S IN (2015 COURSE)?

1. One of the aspects of introducing modern technologies like web-services, microservices is **to improve the quality of learning.**

2. Students must be serious about performing the assignment and try to change their study habits.(**Industry application** oriented assignments).

3. **Attempt to use single programming language:** Unlike previous pattern, here the attempt is to make use of at the most two language platform.

Of course, this will lead us to help in providing uniform guidelines towards conducting practical sessions across university.

COURSE OBJECTIVES &OUTCOME

Course Objectives :
1. The course aims to provide an understanding of the principles on which the distributed systems are based; their architecture, algorithms and how they meet the demands of Distributed applications. 2. The course covers the building blocks for a study related to the design and the implementation of distributed systems and applications. 3. To prepare students for an industrial programming environment.
Course Outcomes :
Upon successful completion of this course student will be able to:
CO-1: Demonstrate knowledge of the core concepts and techniques in distributed systems.
CO-2: Learn how to apply principles of state-of-the-Art Distributed systems in practical application.
CO-3: Design, build and test application programs on distributed systems.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

LIST OF ASSIGNMENTS

Sr. No.	Assignments	Mapping	
		CO	PO
1	To develop any distributed application through implementing client-server communication programs based on **Java Sockets and RMI** techniques.	CO-1	PO-1, PO-3, PO-5
2	To develop any distributed application using Message Passing Interface **(MPI)**.	CO-1	PO-1, PO-3, PO-5
3	To develop any distributed application with **CORBA program using JAVA IDL**.	CO-2	PO-2, PO-3 PO-5
4	To develop any distributed algorithm for **leader election**.	CO-1	PO-1, PO-3
5	To create a simple **web service** and write any distributed application to consume the web service.	CO-2, CO-3	PO-2,3 PO-5
6	To develop any distributed application using Messaging System in **Publish - Subscribe paradigm**.	CO-2, CO-3	PO-2,3 PO-5
7	To develop **Microservices** framework based distributed application.	CO-2, CO-3	PO-2,3 PO-5

ASSIGNMENT NO. 1	
Assignment No. 1 A)	To develop any distributed application through implementing client-server communication programs based on **TCP /UDP Java Sockets**
Objective(s):	By the end of this assignment, the student will be able to implement any distributed multi-threaded client-server programs using Java sockets.
Assignment No. 1 B)	To develop any distributed application through implementing client-server communication programs based on **Java RMI.**
Objective(s):	By the end of this assignment, the student will be able to implement any distributed applications based on RMI.
Tools	Java Programming Environment, jdk 1.8, rmiregistry.

JAVA SOCKET PROGRAMMING

- Java Socket programming is used for communication between the applications running on different JRE.
- Java Socket programming can be **connection-oriented** or **connection-less**.
- **Socket** and **ServerSocket** classes are used for connection-oriented socket programming.
- **DatagramSocket** and **DatagramPacket** classes are used for connection-less socket programming.
- **Socket class and methods:**
  - A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.
  - public InputStream getInputStream(): returns the InputStream attached with this socket.
  - public OutputStream getOutputStream():returns the OutputStream attached with this socket.
  - public synchronized void close(): closes this socket

JAVA API FOR STREAM COMMUNICATION

- **ServerSocket class and methods:**
  - The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.
  - public Socket accept(): returns the socket and establish a connection between server and client.
  - public synchronized void close(): closes the server socket.
- **Java.net.DatagramSocket class :**
  - Every packet sent from a datagram socket is individually routed and delivered.
  - It can also be used for sending and receiving broadcast messages.
  - Datagram Sockets is the java's mechanism for providing network communication via UDP instead of TCP.
  - **DatagramSocket() :** Creates a datagramSocket and binds it to any available port on local machine. If this constructor is used, the OS would assign any port to this socket.

JAVA API FOR DATAGRAM COMMUNICATION

- **DatagramPacket** : A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. In Java, `DatagramPacket` represents a datagram.
- You can create a DatagramPacket object by using one of the following constructors:
- `DatagramPacket(byte[] buf, int length)`
- `DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
- The data must be in the form of an array of bytes.
- The first constructor is used to create a `DatagramPacket` to be received.
- The second constructor creates a `DatagramPacket` to be sent, so you need to specify the address and port number of the destination host.
- The parameter length specifies the amount of data in the byte array to be used, usually is the length of the array (`buf.length`).

JAVA API FOR DATAGRAM COMMUNICATION

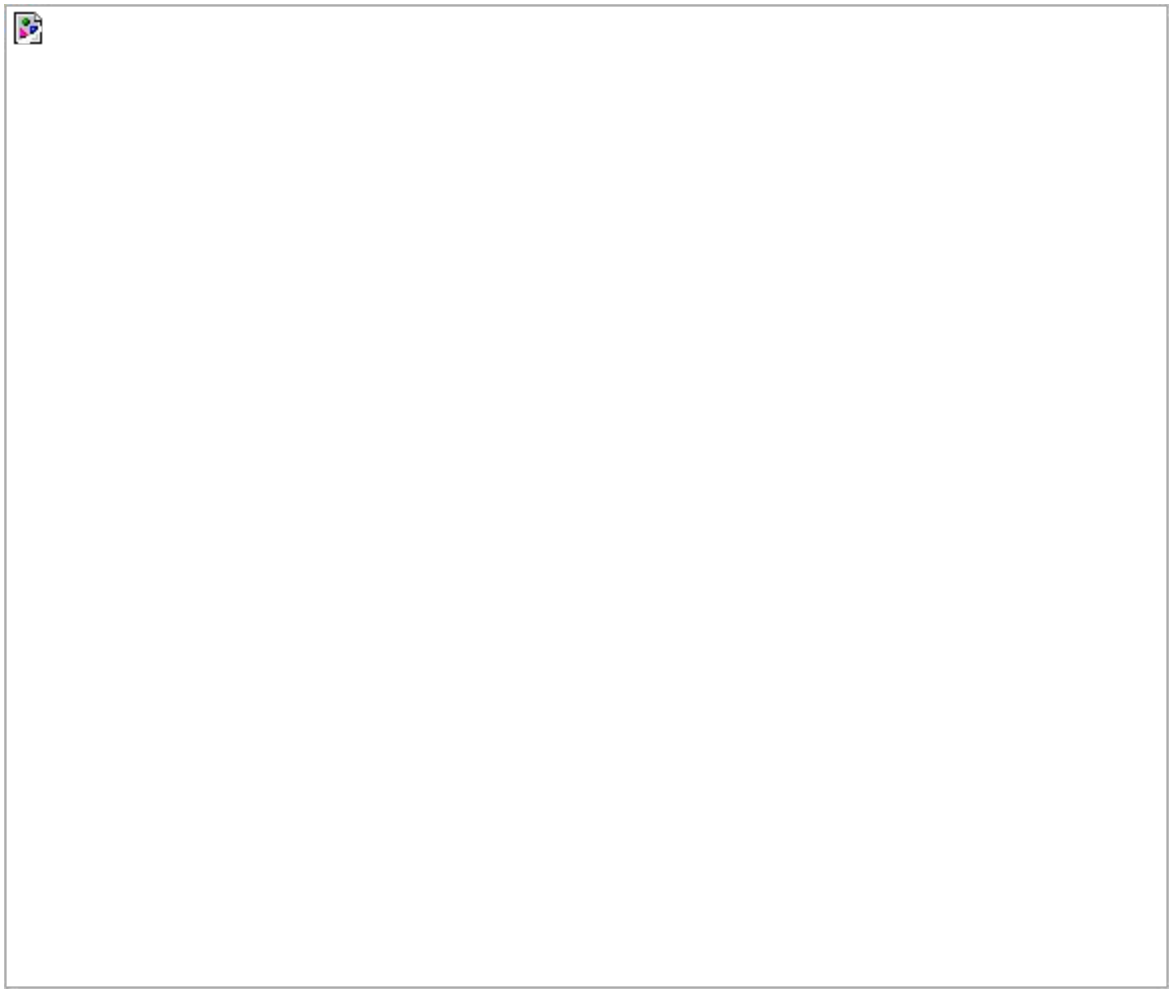
- **DatagramSocket :** `DatagramSocket` is used to send and receive `DatagramPackets`.
- In Java, `DatagramSocket` is used for both client and server. There are no separate classes for client and server like TCP sockets.
- `DatagramSocket` creates object to establish a UDP connection for sending and receiving datagram, by using the following constructors:
- `DatagramSocket(int port, InetAddress laddr)`: This constructor binds the server to the specified IP address (in case the computer has multiple IP addresses).
- The key methods of the `DatagramSocket` include:
- **`send`**`(DatagramPacket p)`: sends a datagram packet.
- **`receive`**`(DatagramPacket p)`: receives a datagram packet.
- **`close`**`()`: closes the socket.

JAVA TCP SOCKET PROGRAMMING

- Create two java files, **Server.java** and **Client.java**.
- Server file contains two classes namely: **Sever** (public class for creating server) and **ClientHandler** (for handling any client using multithreading).
- Client file contain only one public class **Client** (for creating a client).
- Java API networking package (java.net) to be imported which takes care of all network programming.

QUICK OVERVIEW

- The diagram shows how these three classes interact with each other.



HOW THESE PROGRAM WORKS TOGETHER?

- When a client, say client1 sends a request to connect to server, the server assigns a new thread to handle this request.
- The newly assigned thread is given the access to streams for communicating with the client.
- After assigning the new thread, the server via its while loop, again comes into accepting state.
- When a second request comes while first is still in process, the server accepts this requests and again assigns a new thread for processing it. In this way, multiple requests can be handled even when some requests are in process.

SERVER SIDE PROGRAMMING (Server.java)

- **Server class :** The steps involved on server side:

1. **Establishing the Connection:** Server socket object is initialized and inside a while loop a socket object continuously accepts incoming connection.
2. **Obtaining the Streams:** The inputstream object and outputstream object is extracted from the current requests' socket object. Streams reads data (numbers) instead of just bytes.
3. **Creating a handler object:** After obtaining the streams and port number, a new clientHandler object is created with these parameters.
4. **Invoking the start() method :** The start() method is invoked on this newly created thread object.

SERVER SIDE PROGRAMMING (Server.java)

```
// Java implementation of Server side
// It contains two classes : Server and ClientHandler
// Save file as Server.java

import java.io.*;
import java.text.*;
import java.util.*;
import java.net.*;

// Server class
public class Server
{
    public static void main(String[] args) throws IOException
    {
        // server is listening on port 5056
        ServerSocket ss = new ServerSocket(5056);

        // running infinite loop for getting
        // client request
        while (true)
        {
            Socket s = null;

            try
            {
                // socket object to receive incoming client requests
                s = ss.accept();

                System.out.println("A new client is connected : " + s);
```

SERVER SIDE PROGRAMMING (Server.java)

```
// ClientHandler class
class ClientHandler extends Thread
{
    DateFormat fordate = new SimpleDateFormat("yyyy/MM/dd");
    DateFormat fortime = new SimpleDateFormat("hh:mm:ss");
    final DataInputStream dis;
    final DataOutputStream dos;
    final Socket s;


    // Constructor
    public ClientHandler(Socket s, DataInputStream dis, DataOutputStream dos)
    {
        this.s = s;
        this.dis = dis;
        this.dos = dos;
    }

    @Override
    public void run()
    {
        String received;
        String toreturn;
        while (true)
        {
            try {

                // Ask user what he wants
                dos.writeUTF("What do you want?[Date | Time]..\n"+
                             "Type Exit to terminate connection.");

                // receive the answer from client
                received = dis.readUTF();
```

SERVER SIDE PROGRAMMING (Server.java)

```
            if(received.equals("Exit"))
            {
                System.out.println("Client " + this.s + " sends exit...");
                System.out.println("Closing this connection.");
                this.s.close();
                System.out.println("Connection closed");
                break;
            }

            // creating Date object
            Date date = new Date();

            // write on output stream based on the
            // answer from the client
            switch (received) {

                case "Date" :
                    toreturn = fordate.format(date);
                    dos.writeUTF(toreturn);
                    break;

                case "Time" :
                    toreturn = fortime.format(date);
                    dos.writeUTF(toreturn);
                    break;

                default:
                    dos.writeUTF("Invalid input");
                    break;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
                try
                {
                    // closing resources
                    this.dis.close();
                    this.dos.close();

                }catch(IOException e){
                    e.printStackTrace();
                }
            }
        }
```

CLIENT SIDE PROGRAMMING (Client.java)

- Client side programming is similar as in general socket programming program with the following steps-

1. Establish a Socket Connection **2.** Communication **3.** Close the connection

```
import java.io.*;
import java.net.*;
import java.util.Scanner;

// Client class
public class Client
{
    public static void main(String[] args) throws IOException
    {
        try
        {
            Scanner scn = new Scanner(System.in);

            // getting localhost ip
            InetAddress ip = InetAddress.getByName("localhost");

            // establish the connection with server port 5056
            Socket s = new Socket(ip, 5056);

            // obtaining input and out streams
            DataInputStream dis = new DataInputStream(s.getInputStream());
            DataOutputStream dos = new DataOutputStream(s.getOutputStream());
```

CLIENT SIDE PROGRAMMING (Client.java)

```
            // the following loop performs the exchange of
            // information between client and client handler
            while (true)
            {
                System.out.println(dis.readUTF());
                String tosend = scn.nextLine();
                dos.writeUTF(tosend);

                // If client sends exit,close this connection
                // and then break from the while loop
                if(tosend.equals("Exit"))
                {
                    System.out.println("Closing this connection : " + s);
                    s.close();
                    System.out.println("Connection closed");
                    break;
                }

                // printing date or time as requested by client
                String received = dis.readUTF();
                System.out.println(received);
            }

            // closing resources
            scn.close();
            dis.close();
            dos.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

STEPS FOR COMPILATION AND EXECUTION

- **If you're using Eclipse :**
- Compile both of them on two different terminals or tabs
- First run the Server.java followed by the Client.java.
- Run multiple instances from the same program
- Type messages in the Client Window which will be received and showed by the Server Window simultaneously.
- Type 'Exit' to end.

EXPECTED OUTPUT

```
What do you want?[Date | Time]..
Type Exit to terminate connection.
Date
2019/01/17
What do you want?[Date | Time]..
Type Exit to terminate connection.
Time
05:35:28
What do you want?[Date | Time]..
Type Exit to terminate connection.
Over
Invalid input
What do you want?[Date | Time]..
Type Exit to terminate connection.
Exit Closing this connection :
  Socket[addr=localhost/127.0.0.1,port=5056,localport=605
  36] Connection closed
```

JAVA UDP CLIENT PROGRAM

- Write a code for a client program that requests for quotes from a server that implements the Quote of the Day (QOTD) service .
- The code of the full client program that parameterizes the hostname and port number, handles exceptions and gets a quote from the server for every 10 seconds:

```
import java.io.*;
import java.net.*;

/**
 * This program demonstrates how to implement a UDP client program.
 *
 *
 * @author www.codejava.net
 */
public class QuoteClient {

    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Syntax: QuoteClient <hostname> <port>");
            return;
        }

        String hostname = args[0];
        int port = Integer.parseInt(args[1]);
```

JAVA UDP CLIENT PROGRAM

- Write a code for a client program that requests for quotes from a server that implements the Quote of the Day (QOTD) service .

```
        try {
            InetAddress address = InetAddress.getByName(hostname);
            DatagramSocket socket = new DatagramSocket();

            while (true) {

                DatagramPacket request = new DatagramPacket(new byte[1], 1, address, port);
                socket.send(request);

                byte[] buffer = new byte[512];
                DatagramPacket response = new DatagramPacket(buffer, buffer.length);
                socket.receive(response);

                String quote = new String(buffer, 0, response.getLength());

                System.out.println(quote);
                System.out.println();

                Thread.sleep(10000);
            }

        } catch (SocketTimeoutException ex) {
            System.out.println("Timeout error: " + ex.getMessage());
            ex.printStackTrace();
        } catch (IOException ex) {
            System.out.println("Client error: " + ex.getMessage());
            ex.printStackTrace();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

JAVA UDP SERVER PROGRAM

- The sample program demonstrates how to implement a server for the above client. The code creates a UDP server listening on port 17 and waiting for client's request:

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * This program demonstrates how to implement a UDP server program.
 *
 *
 * @author www.codejava.net
 */
public class QuoteServer {
    private DatagramSocket socket;
    private List<String> listQuotes = new ArrayList<String>();
    private Random random;

    public QuoteServer(int port) throws SocketException {
        socket = new DatagramSocket(port);
        random = new Random();
    }

    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Syntax: QuoteServer <file> <port>");
            return;
        }

        String quoteFile = args[0];
        int port = Integer.parseInt(args[1]);
```

JAVA UDP SERVER PROGRAM

- The sample program demonstrates how to implement a server for the above client. The code creates a UDP server listening on port 17 and waiting for client's request:

```
    try {
        QuoteServer server = new QuoteServer(port);
        server.loadQuotesFromFile(quoteFile);
        server.service();
    } catch (SocketException ex) {
        System.out.println("Socket error: " + ex.getMessage());
    } catch (IOException ex) {
        System.out.println("I/O error: " + ex.getMessage());
    }
}

private void service() throws IOException {
    while (true) {
        DatagramPacket request = new DatagramPacket(new byte[1], 1);
        socket.receive(request);

        String quote = getRandomQuote();
        byte[] buffer = quote.getBytes();

        InetAddress clientAddress = request.getAddress();
        int clientPort = request.getPort();

        DatagramPacket response = new DatagramPacket(buffer, buffer.length, clientAddress, clientPort);
        socket.send(response);
    }
}
```

JAVA UDP SERVER PROGRAM

- The sample program demonstrates how to implement a server for the above client. The code creates a UDP server listening on port 17 and waiting for client's request:

```
    private void loadQuotesFromFile(String quoteFile) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(quoteFile));
        String aQuote;

        while ((aQuote = reader.readLine()) != null) {
            listQuotes.add(aQuote);
        }

        reader.close();
    }

    private String getRandomQuote() {
        int randomIndex = random.nextInt(listQuotes.size());
        String randomQuote = listQuotes.get(randomIndex);
        return randomQuote;
    }
}
```

- Suppose we have a Quotes.txt file with the following content (each quote is in a single line):

```
1  Whether you think you can or you think you can't, you're right - Henry Ford
2  There are no traffic jams along the extra mile - Roger Staubach
3  Build your own dreams, or someone else will hire you to build theirs - Farrah Gray
4  What you do today can improve all your tomorrows - Ralph Marston
5  Remember that not getting what you want is sometimes a wonderful stroke of luck - Dalai Lama
```

COMPILE AND RUN

- Type the following command to run the server program:

```
java QuoteServer Quotes.txt 17
```

- And run the client program (on the same computer):

```
java QuoteClient localhost 17
```

CONCLUSION

- Thus students have learnt how to develop a client/server distributed application relying on TCP protocol. Based on this knowledge, they are able to develop client programs that communicate with servers via TCP, and developing their own TCP client/server applications.

- Thus students have learnt how to develop a client/server distributed application relying on UDP protocol. Based on this knowledge, they are able to develop client programs that communicate with servers via UDP, and developing their own UDP client/server applications.

CONCLUSION

- Thus students have learnt how to develop a client/server distributed application relying on TCP protocol. Based on this knowledge, they are able to develop client programs that communicate with servers via TCP, and developing their own TCP client/server applications.

- Thus students have learnt how to develop a client/server distributed application relying on UDP protocol. Based on this knowledge, they are able to develop client programs that communicate with servers via UDP, and developing their own UDP client/server applications.

REFERENCES

- Java UDP Client Server Program Example: https://www.codejava.net/java-se/networking/java-udp-client-server-program-example.
- DatagramPacket Javadoc: https://docs.oracle.com/javase/8/docs/api/java/net/DatagramPacket.html
- DatagramSocket Javadoc: https://docs.oracle.com/javase/8/docs/api/java/net/DatagramSocket.html
- Socket Programming in Java: https://www.geeksforgeeks.org/socket-programming-in-java/

ASSIGNMENT NO. 1B)	
Assignment No. 1 B)	To develop any distributed application through implementing client-server communication programs based on **Java 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 INTERFACE (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 int-er mediate 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:

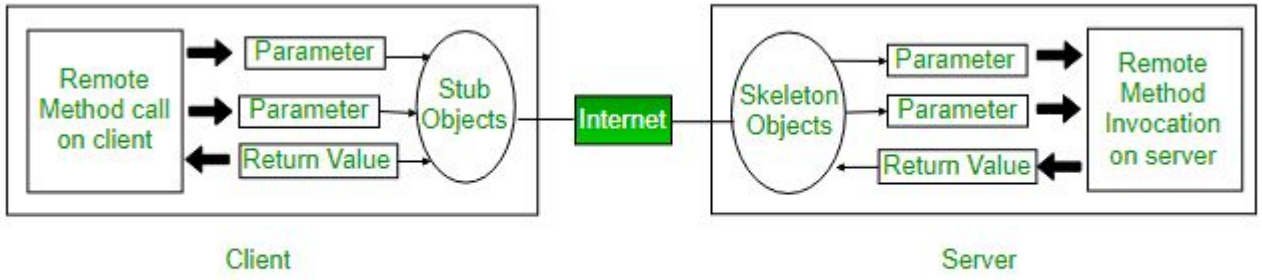
  An identifier of the remote object to be used

  Method name which is to be invoked

  Parameters to the remote JVM.

RMI IMPLEMENTATION

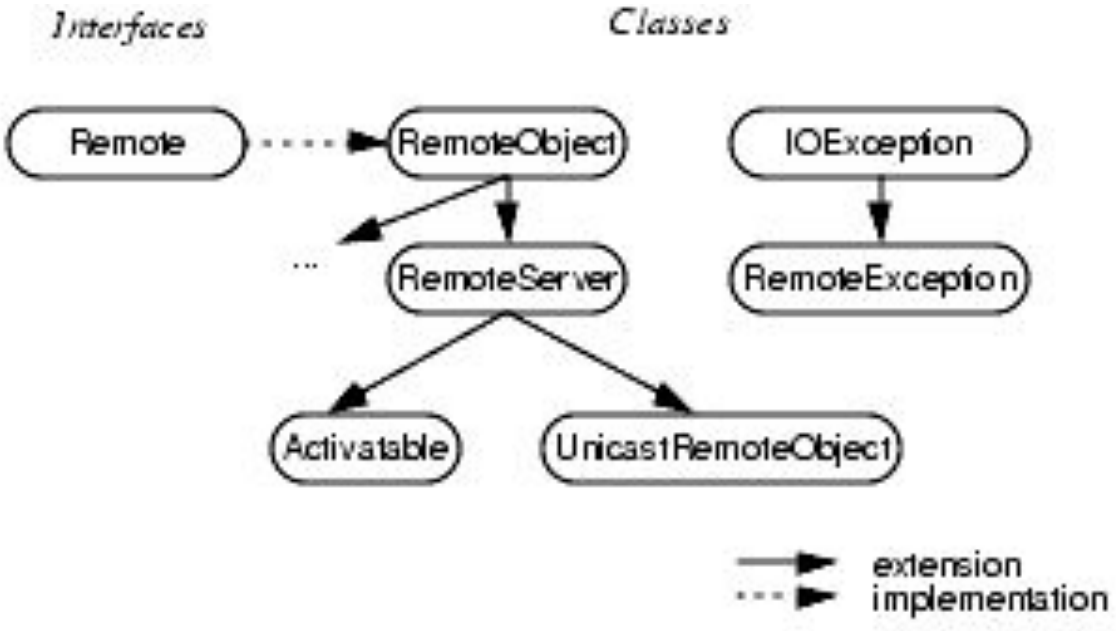
- **Skeleton Object**
- The skeleton object passes the request from the stub object to the remote object. It performs following tasks:
  - It calls the desired method on the real object present on the server.
  - It forwards the parameters received from the stub object to the method .



Working of RMI

REMOTE METHOD INTERFACE (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 INTERFACE (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 INTERFACE (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 rmi c (rmi complier)
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 cl9
            Naming.rebind("rmi://localhost:1900"+
                        "/cl9",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 6: 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 ClientRequet Application:

#Javac ClientRequest.java

#java ClientRequest

OUTPUT

```
dos@ddos:~/Desktop/CL9$ javac SearchQuery.java
dos@ddos:~/Desktop/CL9$ rmic SearchQuery
Warning: generation and use of skeletons and static stubs for JRMP
is deprecated. Skeletons are unnecessary, and static stubs have
been superseded by dynamically generated stubs. Users are
encouraged to migrate away from using rmic to generate skeletons and static
stubs. See the documentation for java.rmi.server.UnicastRemoteObject.
dos@ddos:~/Desktop/CL9$ rmiregistry
```

```
dos@ddos:~/Desktop/CL9$ javac SearchServer.java
dos@ddos:~/Desktop/CL9$ java SearchServer
```

```
dos@ddos:~$ cd Desktop/CL9/
dos@ddos:~/Desktop/CL9$ javac ClientRequest.java
dos@ddos:~/Desktop/CL9$ java ClientRequest
Article on RMI in Java Found at CL9
```

```
dos@ddos:~$ cd Desktop/CL9/
dos@ddos:~/Desktop/CL9$ ls
ClientRequest.java  SearchQuery.class       SearchServer.java
Search.class        SearchQuery.java
Search.java         SearchQuery_Stub.class
dos@ddos:~/Desktop/CL9$ javac SearchServer.java
dos@ddos:~/Desktop/CL9$ java SearchServer
```

REFERENCES

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

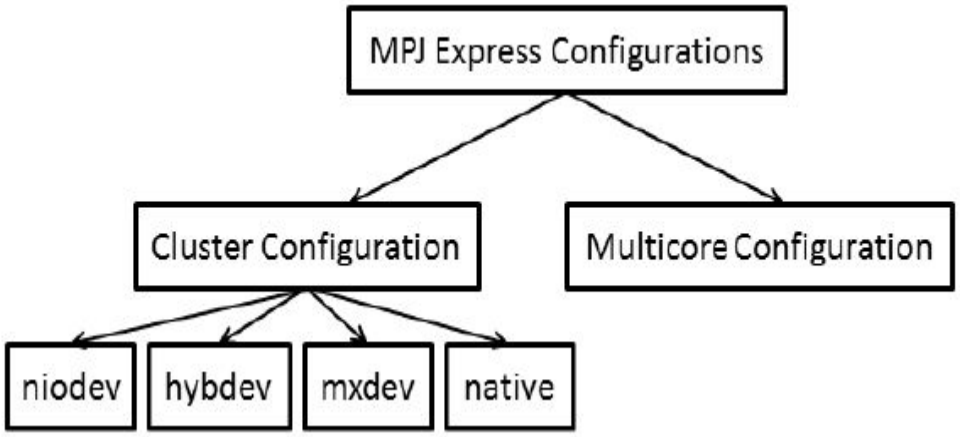
ASSIGNMENT NO. 2	
Assignment No. 2	To develop any distributed application using Message Passing Interface (MPI).
Objective(s):	By the end of this assignment, the student will be able to implement any distributed applications based on MPI.
Tools	MPJ Express Software (Version 0.44), Java 8.

MESSAGE PASSING INTERFACE

- **Message passing** is a popularly renowned mechanism to implement parallelism in applications; it is also called MPI.
- A basic prerequisite for message passing is a good communication API.
- **MPJ Express** is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers.
- **MPJ is a familiar Java API for MPI implementation.**
- MPJ Express is essentially a middleware that supports communication between individual processors of clusters.
- The programming model followed by MPJ Express is Single Program Multiple Data (SPMD).
- **MPJ Express** is designed for distributed memory machines like network of computers or clusters, it is possible to **efficiently execute parallel user applications on desktops or laptops t**hat contain shared memory or multicore processors.

MPJ: MPI using JAVA

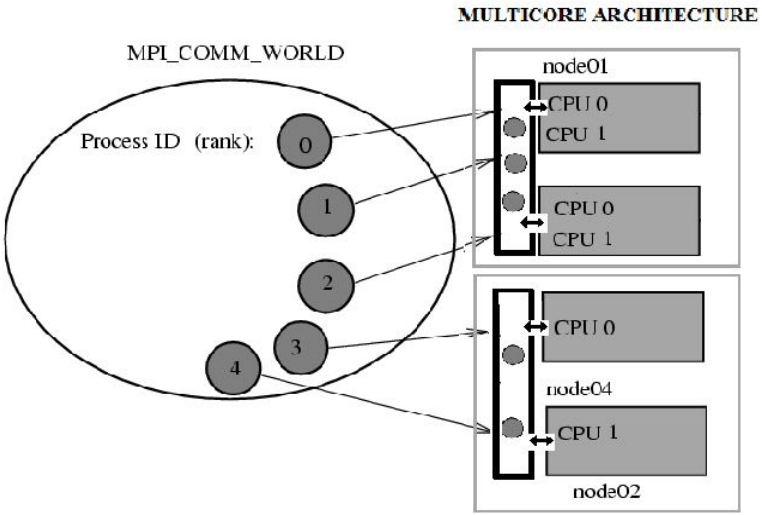
- **MPJ Express Configuration:**
- The MPJ Express software can be configured in two ways as shown in Figure.
- Multicore configuration—is used to execute MPJ Express user programs on laptops and desktops.
- The cluster configuration—is used to execute MPJ Express user programs on clusters or network of computers.



MPJ: MPI using JAVA

- **Multicore configuration:**
- The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops—typically such hardware contains shared memory and multicore processors.
- In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors.

Also this configuration is preferred for teaching purposes since students can execute message passing code on their personal laptops and desktops. The user applications stay the same when executing the code in multicore or cluster configuration.



GETTING STARTED WITH MPJ Express

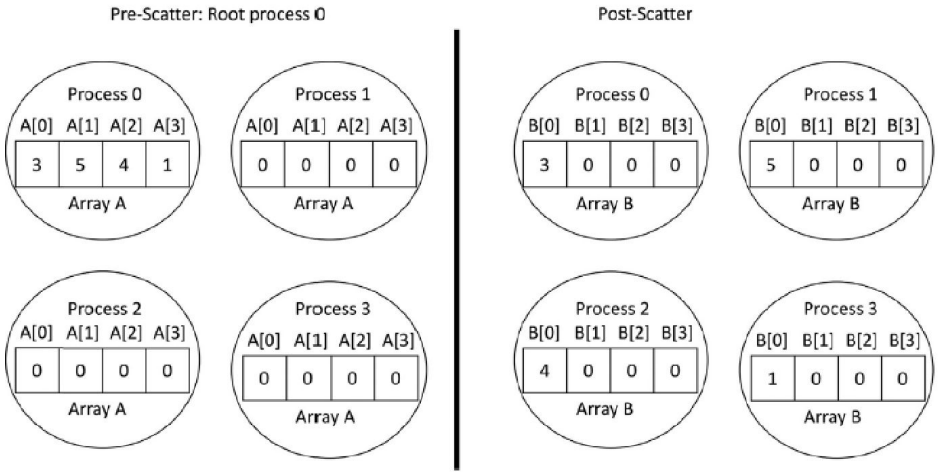
- **Installing MPJ Express:**
- Download MPJ Express (mpj.jar) and unpack it.
- Set environment variables MPJ_HOME and PATH:
  - `export MPJ_HOME=/path/to/mpj/`
  - `export PATH=$MPJ_HOME/bin:$PATH`
- Create a new working directory for MPJ Express programs. e.g. /mpj-user directory.
- Compile the MPJ Express library : cd $MPJ_HOME; ant

MPI ENVIRONMENT

- MPI is for communication among processes, which have separate address spaces.
- **Group** is the set of processes that communicate with one another.
- **Communicator** is the central object for communication in MPI.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.
- Every MPI program must contain `import mpi.MPI`
- **MPI_Init** initializes the execution environment for MPI.
- A process is identified by its ***rank*** in the group associated with a communicator.
- How many processes are participating in this computation?
  - **`MPI_Comm_size`** function reports the number of processes.
  - **`MPI_Comm_rank`** function reports the *rank*, a number between 0 and size-1, identifying the calling process.
- **MPI_Finalize** cleans up all the extraneous mess that was first put into place by MPI_Init.

COMMUNICATION COLLECTIVES:
Collective Operations in MPI

- *Collective communications* refer to set of MPI functions that transmit data among all processes specified by a given communicator.
- *Scatter-gather:* Ability to "scatter" data items in a message into multiple memory locations and "gather" data items from multiple memory locations into one message.
- In MPI_Gather, each process sends content of send buffer to the root process. Root receives and stores in rank order.
- IN sendbuf starting address of send buffer.
- OUT recvbuf (address of receive buffer).



MPI IMPLEMENTATION

- Firstly, we will create a file named, "`ScatterGather.java`".
- At the beginning of the code, import `mpi.MPI`.

```
import mpi.MPI
```

- To import the above package you have to add `MPJ.jar` during compilation. This will provide support for multicore architecture and creates an MPI environment i.e. it allows to execute parallel java applications.
- Create a class `ScatterGather` with main method

```
public class ScatterGather {
    public static void main(String args[]){
//Initialize MPI execution environment
    MPI.Init(args);
//Get the id of the process
    int rank = MPI.COMM_WORLD.Rank();
//total number of processes is stored in size
    int size = MPI.COMM_WORLD.Size();
    int root=0;
```

MPI IMPLEMENTATION

- Root process initializes the `sendbuff[ ]` data array.

```
//array which will be filled with data by root process
    int sendbuf[]=null;

    sendbuf= new int[size];

//creates data to be scattered
    if(rank==root){
        sendbuf[0] = 10;
        sendbuf[1] = 20;
        sendbuf[2] = 30;
        sendbuf[3] = 40;

    //print current process number
        System.out.print("Processor "+rank+" has data: ");
        for(int i = 0; i < size; i++){
            System.out.print(sendbuf[i]+" ");
        }
        System.out.println();
    }
```

MPI IMPLEMENTATION

- When Scatter() method is called, it sends data from the root process to other processes.

```
//following are the args of Scatter method
//send, offset, chunk_count, chunk_data_type, recv, offset, chunk_count, chunk_data_type, root_process_id
    MPI.COMM_WORLD.Scatter(sendbuf, 0, 1, MPI.INT, recvbuf, 0, 1, MPI.INT, root);
    System.out.println("Processor "+rank+" has data: "+recvbuf[0]);
    System.out.println("Processor "+rank+" is doubling the data");
```

- Each process doubles the data received.

```
//logic for doubling the number
recvbuf[0]=recvbuf[0]*2;
```

MPI IMPLEMENTATION

- Then root process gathers each individual doubled number with `Gather()` method.
- `Gather()` method takes elements from each process and gathers them to the root process. The elements are ordered by the rank of the process from which they were received.

```
//following are the args of Gather method
//Object sendbuf, int sendoffset, int sendcount, Datatype sendtype,
//Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root
    MPI.COMM_WORLD.Gather(recvbuf, 0, 1, MPI.INT, sendbuf, 0, 1, MPI.INT, root);


//display the gathered result
    if(rank==root){
       System.out.println("Process 0 has data: ");
        for(int i=0;i<4;i++){
            System.out.print(sendbuf[i]+ " ");
        }
    }
//Terminate MPI execution environment
    MPI.Finalize();
}
```

STEPS FOR COMPILATION AND EXECUTION

- **Install**ing MPJ Express Programs in the Multicore Configuration

1. Download MPJ Express and unpack it.
2. Set MPJ_HOME and PATH environment variables:

   export MPJ_HOME=/path/to/mpj/

   export PATH=$MPJ_HOME/bin:$PATH

   (These above two lines can be added to ~/.bashrc)

- **Compile:** `ScatterGather.java`

  `javac -cp $MPJ_HOME/lib/mpj.jar ScatterGather.java` (`mpj.jar` is inside lib folder in the downloaded MPJ Express)
- **Execute:** $MPJ_HOME/bin/`mpjrun.sh -np 4 ScatterGather`

EXPECTED OUTPUT

```
pict@ubuntu:~/Documents/Lab/A2$ export MPJ_HOME=/home/pict/Documents/Lab/A2/mpj-v0_44
pict@ubuntu:~/Documents/Lab/A2$ export PATH=$MPJ_HOME/bin:$PATH
pict@ubuntu:~/Documents/Lab/A2$ javac -cp $MPJ_HOME/lib/mpj.jar ScatterGatherTest.java
pict@ubuntu:~/Documents/Lab/A2$ $MPJ_HOME/bin/mpjrun.sh -np 4 ScatterGatherTest
MPJ Express (0.44) is started in the multicore configuration
Processor 0 has data: 10 20 30 40
Processor 0 has data: 10
Processor 0 is doubling the data
Processor 1 has data: 20
Processor 1 is doubling the data
Processor 2 has data: 30
Processor 2 is doubling the data
Processor 3 has data: 40
Processor 3 is doubling the data
Process 0 has data:
20 40 60 80
```

CONCLUSION

- There has been a large amount of interest in parallel programming using Java. mpj is an MPI binding with Java along with the support for multicore architecture so that user can develop the code on it's own laptop or desktop. This is an effort to develop and run parallel programs according to MPI standard.

REFERENCES

1. MPJ Express Documentation: http://mpj-express.org/docs/guides/linuxguide.pdf

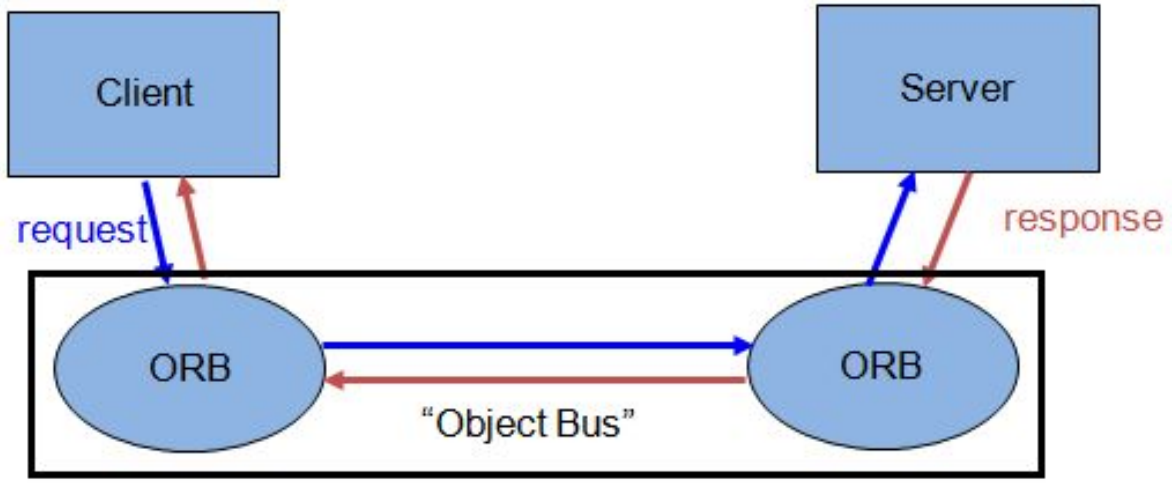
2. How Scatter Gather Works - https://www.open-mpi.org/papers/mpi-java-presentation/mpi-java1995.pdf

3. Implementation Specific Scatter-Gather: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

ASSIGNMENT NO. 3	
Assignment No. 3	To develop any distributed application with CORBA program using JAVA IDL.
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

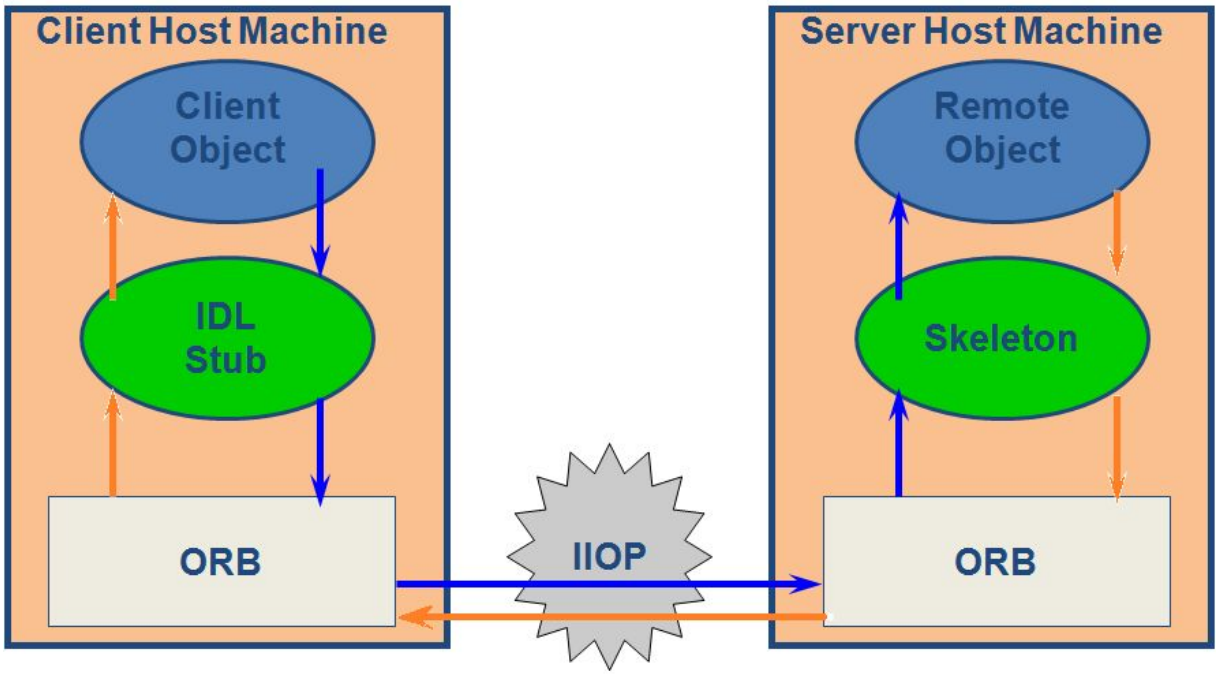
- A CORBA (Common Object Request Broker Architecture) application is developed using IDL (Interface Definition Language).
- IDL is used to define interfaces and the Java IDL compiler generates skeleton code.

- 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.

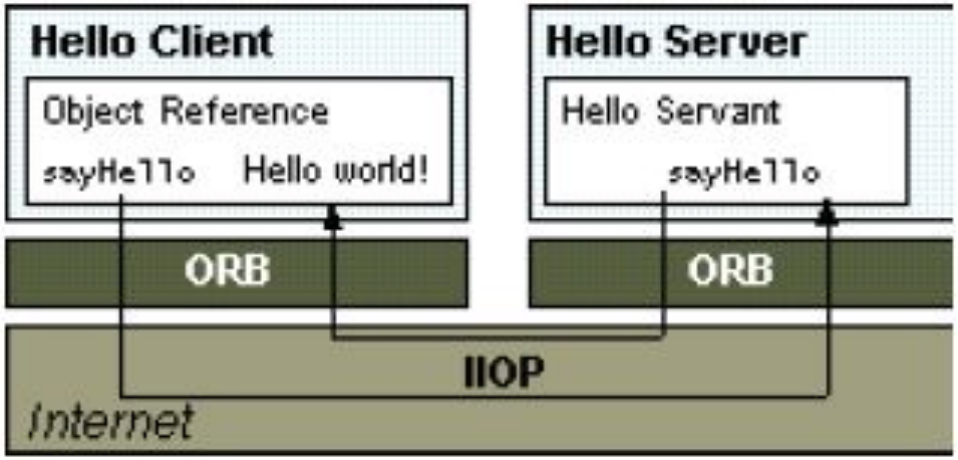


JAVA IDL-USING CORBA FROM JAVA

- 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 Langauge (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:

i. **`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.

ii. **`_HelloStub.java`** : This class is the **client stub** providing CORBA functionality for the client. It implements the `Hello.java` interface.

iii. **`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`.

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

```
HelloServant helloRef = new HelloServant();
```

- The next step is **to connect the servant to the ORB**, so that the ORB can recognize invocations on it and pass them along to the correct servant:

```
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:**

IMPLEMENT THE SERVER

- **Waiting for Invocation**
- This 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 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);
```

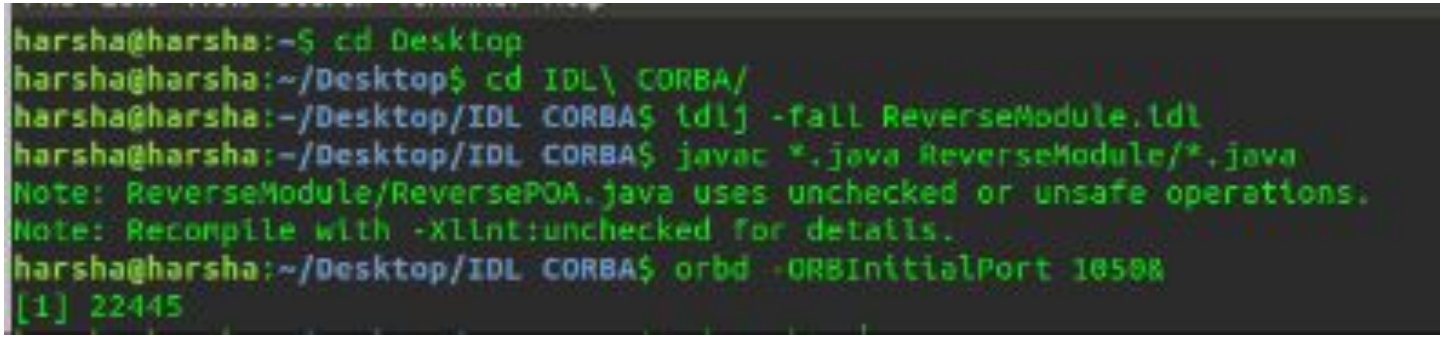
Implementation Details:String Reverse

- 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&
```



Implementation Details:String Reverse

- Start the server. To start the server from a UNIX command shell, enter :

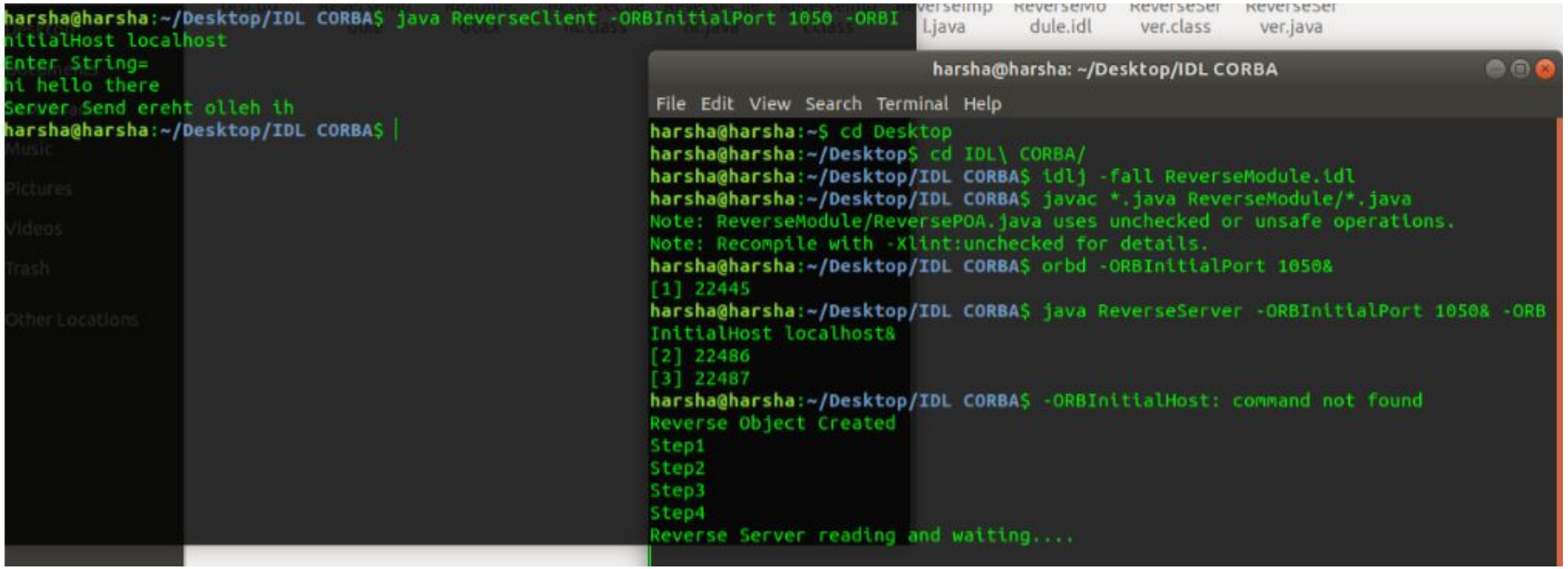
- `java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&`

```
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`



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

ASSIGNMENT NO. 4	
Assignment No. 4	To develop any distributed algorithm for **leader election**.
Objective(s):	By the end of this assignment, the student will be able to explain the concept of Leader Election Algorithms.
Tools	Eclipse, Java 8

Distributed Algorithm

- Distributed Algorithm is a algorithm that runs on a distributed system. Each processor has its own memory and they communicate via communication networks.
- Many algorithms used in distributed system require a coordinator that performs functions needed by other processes in the system.

- **Election algorithms are designed to choose a coordinator**.

- Why Election Algorithm ?
  1. Many distributed algorithms require a process to act as a coordinator.
  2. The coordinator can be any process that organizes actions of other processes.
  3. A coordinator may fail.
  4. How is a new coordinator chosen or elected?

Election Algorithm

- **Assumptions :**
  - Each process has a unique number to distinguish them. Processes know each other's process number.

  There are two types of Election Algorithms:

  1. Bully Algorithm
  2. Ring Algorithm

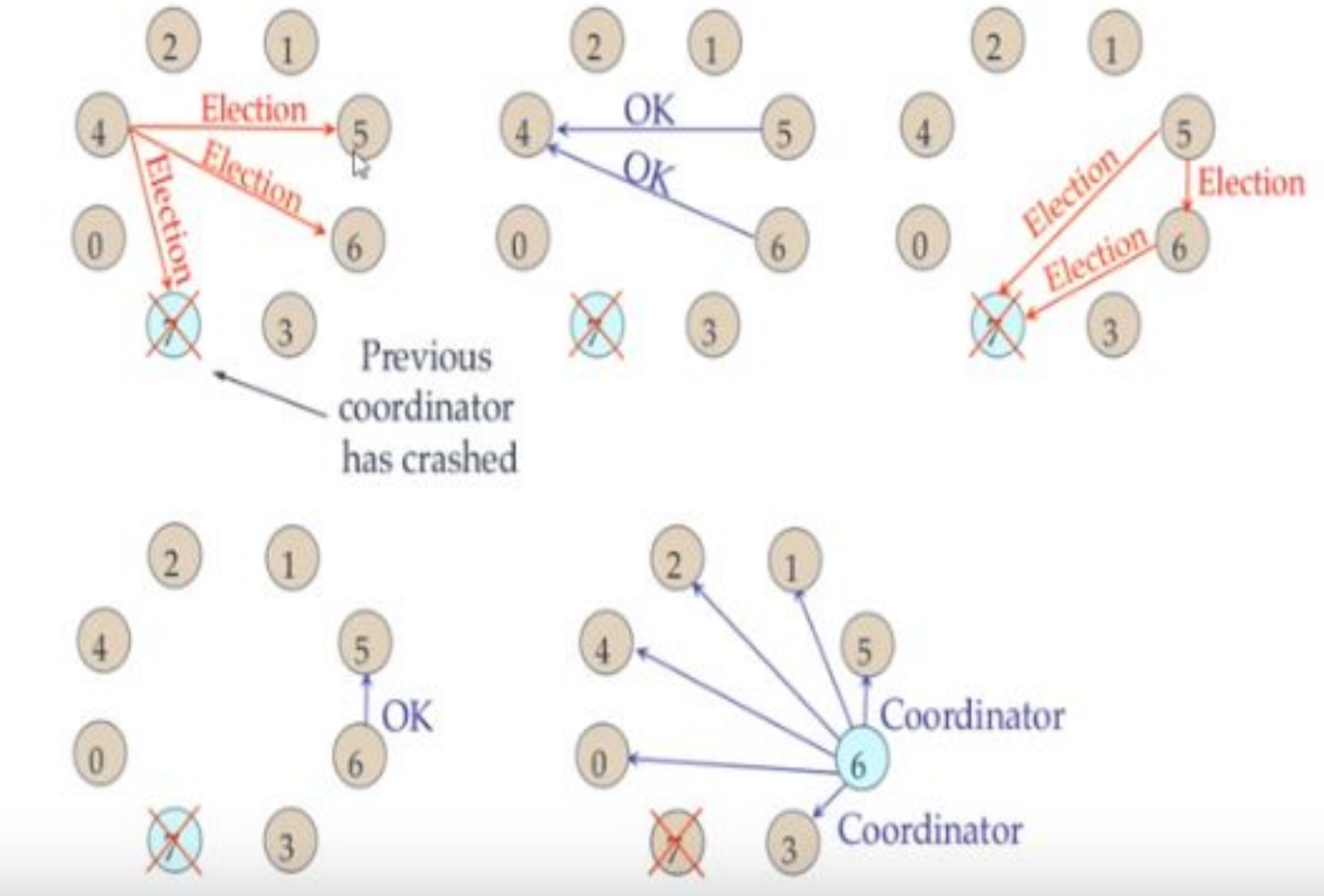
Bully Algorithm

- **Bully Algorithm :**
  This algorithm applies to system where every process can send a message to every other process in the system.

Algorithm – Suppose process P sends a message to the coordinator.

1. If coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
2. Now process P sends election message to every process with high priority number.
3. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.
4. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
5. However, if an answer is received within time T from any other process Q,
   - (I) Process P again waits for time interval T' to receive another message from Q that it has been elected as coordinator.
   - (II) If Q doesn't responds within time interval T' then it is assumed to have failed and algorithm is restarted.

Bully Algorithm



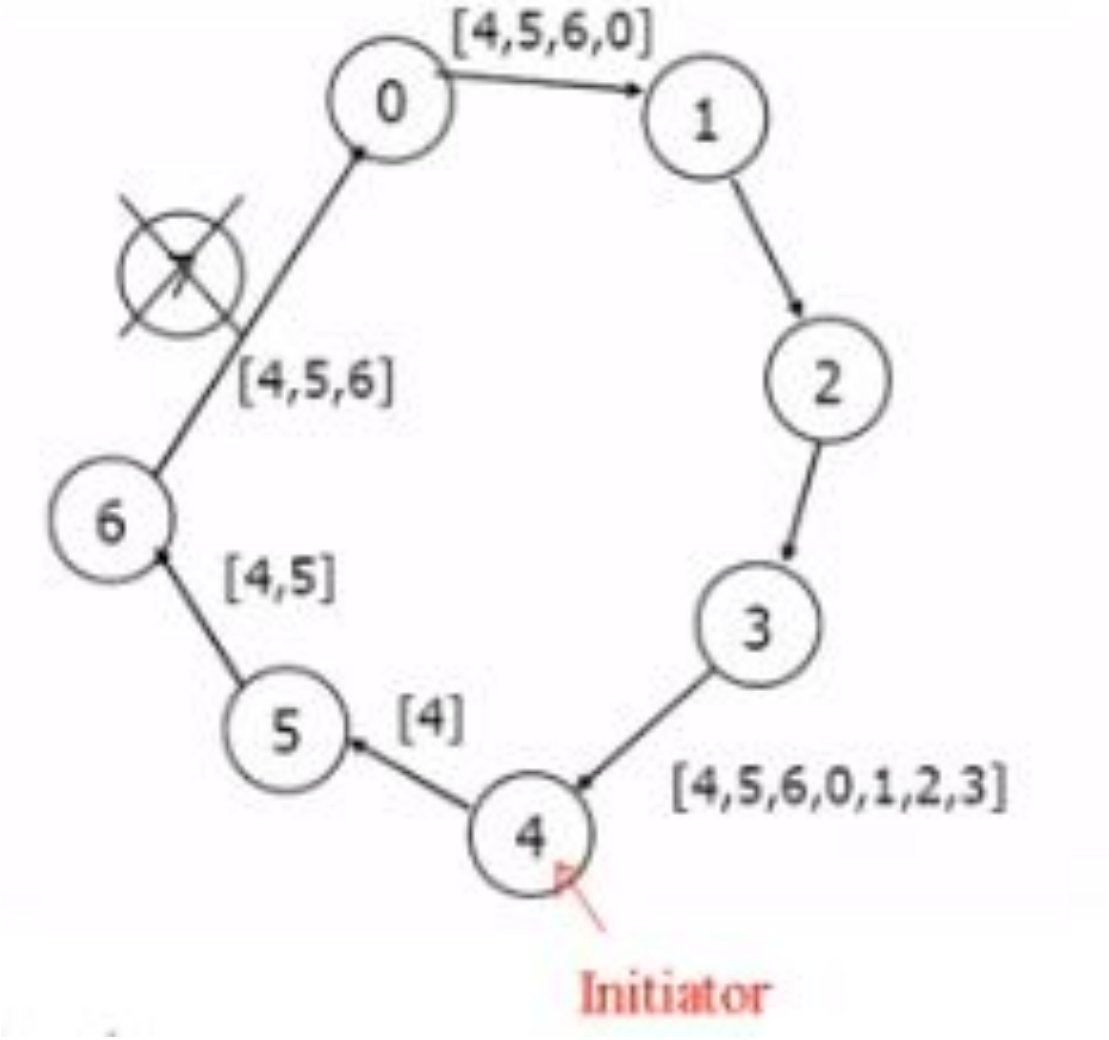
Ring Algorithm

- **Ring Algorithm :** This algorithm applies to systems organized as a ring(logically or physically). In this algorithm we assume that the link between the process are unidirectional and every process can message to the process on its right only. Data structure that this algorithm uses is active list, a list that has priority number of all active processes in the system.

Algorithm –

1. If process P1 detects a coordinator failure, it creates new active list which is empty initially. It sends election message to its neighbour on right and adds number 1 to its active list.
2. If process P2 receives message elect from processes on left, it responds in 3 ways:
   a. If message received does not contain 1 in active list then P1 adds 2 to its active list and forwards the message.
   b. If this is the first election message it has received or sent, P1 creates new active list with numbers 1 and 2. It then sends election message 1 followed by 2
   c. If Process P1 receives its own election message 1 then active list for P1 now contains numbers of all the active processes in the system. Now Process P1 detects highest priority number from list and elects it as the new coordinator.

Ring Algorithm



Implementing the solution For Ring Algorithm

1. Creating Class for Process which includes
   i) State: Active / Inactive
   ii) Index: Stores index of process.
   iii) ID: Process ID

```
133 class Rr {
134
135     public int index;   // to store the index of process
136     public int id;      // to store id/name of process
137     public int f;
138     String state;       // indiactes whether active or inactive state of node
139
140 }
```

2. Import Scanner Class for getting the no of processes as an input

```
16
17 // scanner used for getting input from console
18         Scanner in = new Scanner(System.in);
19         System.out.println("Enter the number of process : ");
20         int num = in.nextInt();
21
```

Implementing the solution For Ring Algorithm

3. Getting input from User for number of Processes and store them into object of classes.

```
23 // getting input from users
24         for (i = 0; i < num; i++) {
25             proc[i].index = i;
26             System.out.println("Enter the id of process : ");
27             proc[i].id = in.nextInt();
28             proc[i].state = "active";
29             proc[i].f = 0;
30         }
31
```

4. Sort these objects on the basis of process id.

```
33 // sorting the processes from on the basis of id
34         for (i = 0; i < num - 1; i++) {
35             for (j = 0; j < num - 1; j++) {
36                 if (proc[j].id > proc[j + 1].id) {
37                     temp = proc[j].id;
38                     proc[j].id = proc[j + 1].id;
39                     proc[j + 1].id = temp;
40                 }
41             }
42         }
43
```

Implementing the solution For Ring Algorithm

5. Make the last process id as "inactive".

```
proc[num - 1].state = "inactive";

System.out.println("\n process " + proc[num - 1].id + "select as co-ordinator");
```

6. Ask for menu 1.Election 2.Quit.

```
63          while (true) {
64              System.out.println("\n 1.election 2.quit ");
65              ch = in.nextInt();
66
```

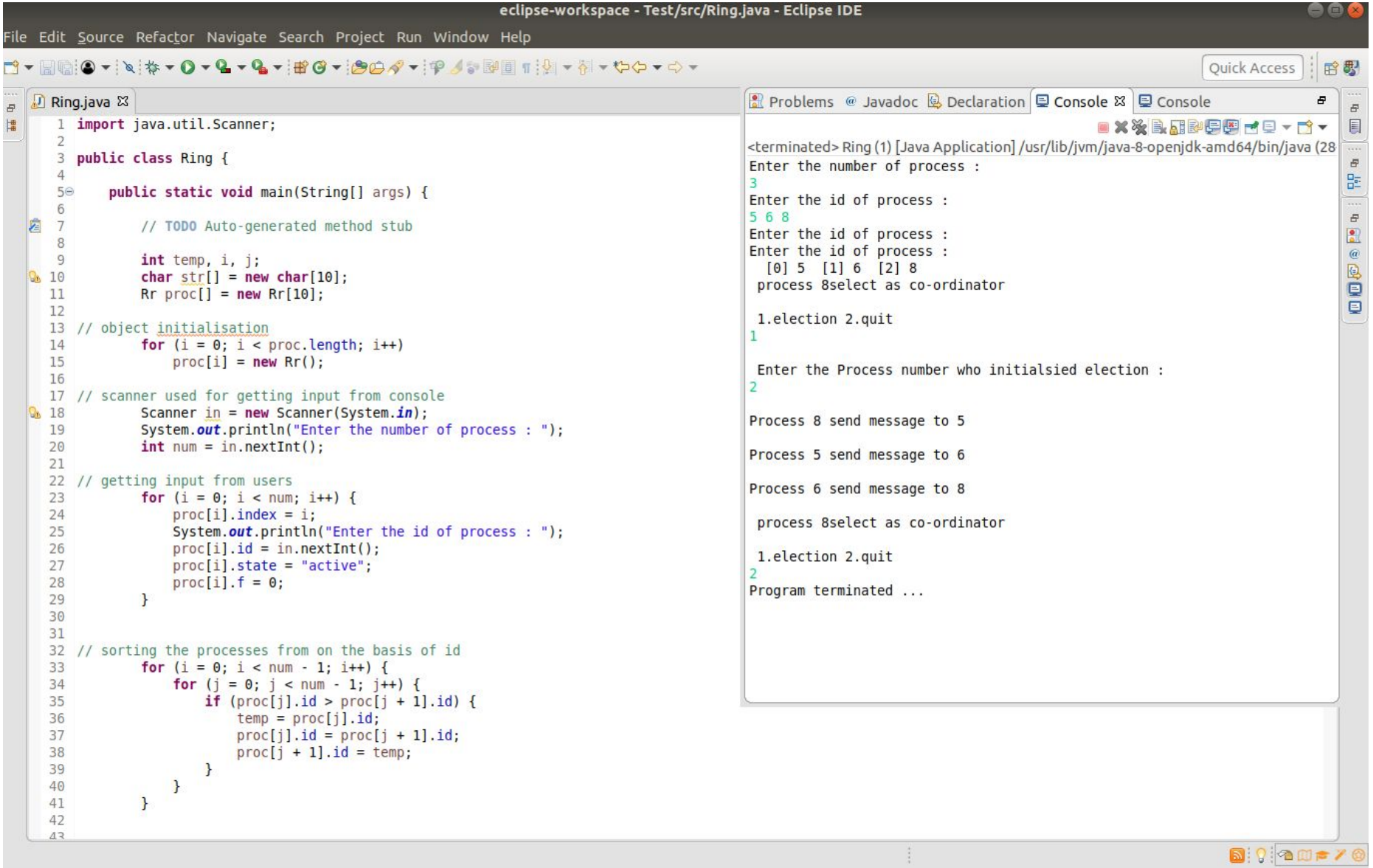
7. Ask for initializing election process.

```
switch (ch) {
case 1:
    System.out.println("\n Enter the Process number who initialsied election : ");
    init = in.nextInt();
```

8.These inputs will be used by Ring Algorithm.

Implementing the solution For Ring Algorithm

8.These inputs will be used by Ring Algorithm.




```java
import java.util.Scanner;

public class Ring {

    public static void main(String[] args) {

        // TODO Auto-generated method stub

        int temp, i, j;
        char str[] = new char[10];
        Rr proc[] = new Rr[10];

// object initialisation
        for (i = 0; i < proc.length; i++)
            proc[i] = new Rr();

// scanner used for getting input from console
        Scanner in = new Scanner(System.in);
        System.out.println("Enter the number of process : ");
        int num = in.nextInt();

// getting input from users
        for (i = 0; i < num; i++) {
            proc[i].index = i;
            System.out.println("Enter the id of process : ");
            proc[i].id = in.nextInt();
            proc[i].state = "active";
            proc[i].f = 0;
        }


// sorting the processes from on the basis of id
        for (i = 0; i < num - 1; i++) {
            for (j = 0; j < num - 1; j++) {
                if (proc[j].id > proc[j + 1].id) {
                    temp = proc[j].id;
                    proc[j].id = proc[j + 1].id;
                    proc[j + 1].id = temp;
                }
            }
        }
```

Implementation of Bully algorithm

eclipse-workspace - Test/src/Bully.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access

Bully.java

```
1
2  import java.io.InputStream;
3  import java.io.PrintStream;
4  import java.util.Scanner;
5
6  public class Bully {
7      static boolean[] state = new boolean[5];
8      int coordinator;
9
10     public static void up(int up) {
11         if (state[up - 1]) {
12             System.out.println("process" + up + "is already up");
13         } else {
14             int i;
15             Bully.state[up - 1] = true;
16             System.out.println("process " + up + "held election");
17             for (i = up; i < 5; ++i) {
18                 System.out.println("election message sent from process" + up + "to p
19             }
20             for (i = up + 1; i <= 5; ++i) {
21                 if (!state[i - 1]) continue;
22                 System.out.println("alive message send from process" + i + "to proce
23                 break;
24             }
25         }
26     }
27
28     public static void down(int down) {
29         if (!state[down - 1]) {
30             System.out.println("process " + down + "is already dowm.");
31         } else {
32             Bully.state[down - 1] = false;
33         }
34     }
35
36     public static void mess(int mess) {
37         if (state[mess - 1]) {
38             if (state[4]) {
39                 System.out.println("0K");
40             } else if (!state[4]) {
41                 int i;
42                 System.out.println("process" + mess + "election");
43                 for (i = mess; i < 5; ++i) {
```

Problems @ Javadoc Declaration Console Console

Bully (1) [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (28-Dec-2018, 7:3

```
5 active process are:
Process up  = p1 p2 p3 p4 p5
Process 5 is coordinator
.........
1 up a process.
2.down a process
3 send a message
4.Exit
2
bring down any process.
5
.........
1 up a process.
2.down a process
3 send a message
4.Exit
3
which process will send message
2
process2election
election send from process2to process 3
election send from process2to process 4
election send from process2to process 5
Coordinator message send from process4to all
.........
1 up a process.
2.down a process
3 send a message
4.Exit
```

References

1. https://www.geeksforgeeks.org/election-algorithm-and-distributed-processing/

ASSIGNMENT NO. 5	
Assignment No. 5	To create a simple **web service** and write any distributed application to consume the web service.
Objective(s):	By the end of this assignment, the student will be able to create, deploy and test Web-service application.
Tools	NetBeans IDE with GlassFish Server, Java 8

WEB-SERVICES

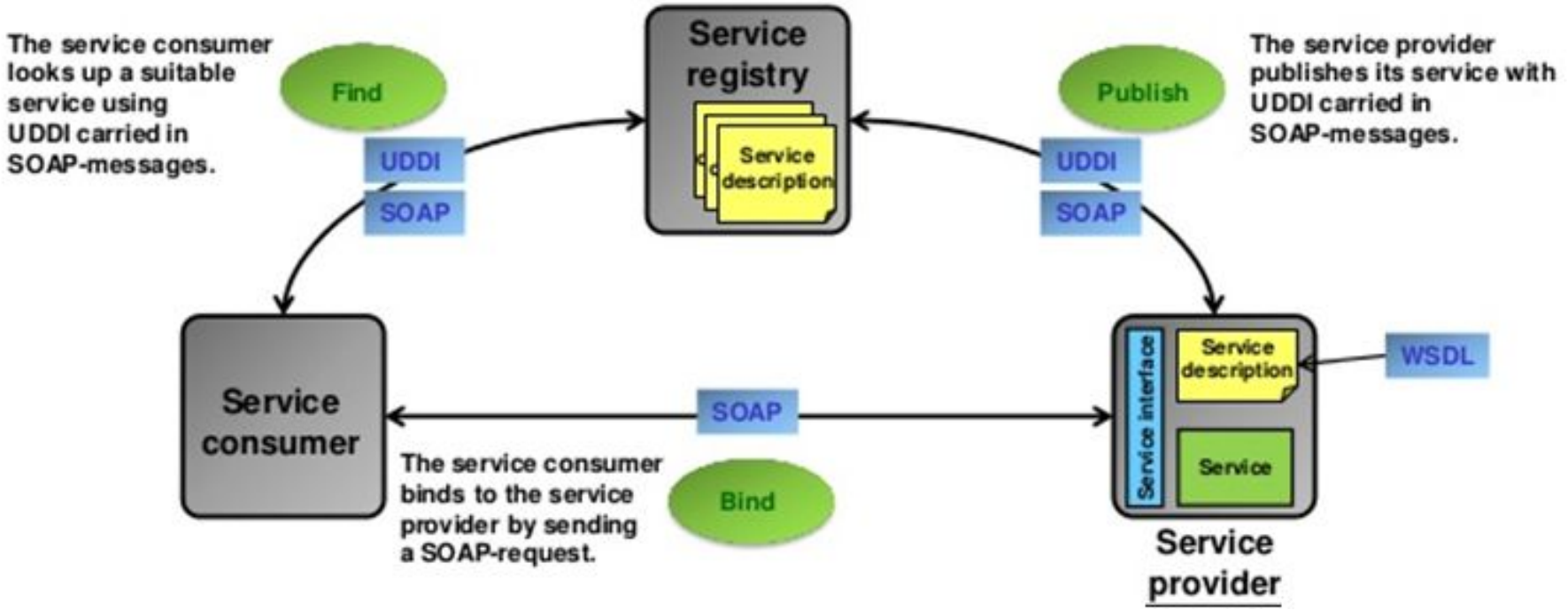
- A Web service, is a method of **communication between two applications** or electronic devices **over the World Wide Web** (WWW).
- A web service can be defined as a **collection of open protocols and standards for exchanging information among systems or applications.**

- A Web service is an abstract notion that must be implemented by a concrete agent. The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterized by the abstract set of functionality that is provided.

WEB SERVICE ARCHITECTURES

The combo SOAP+WSDL+UDDI defines a general model for a web service architecture.

SOAP:	Simple Object Access Protocol
WSDL:	Web Service Description Language
UDDI:	Universal Description and Discovery Protocol
Service consumer:	User of a service
Service provider:	Entity that implements a service (=server)
Service registry:	Central place where available services are listed and advertised for lookup



TYPES OF WEB SERVICES

There are two types of web services:

SOAP-Based Web Services:

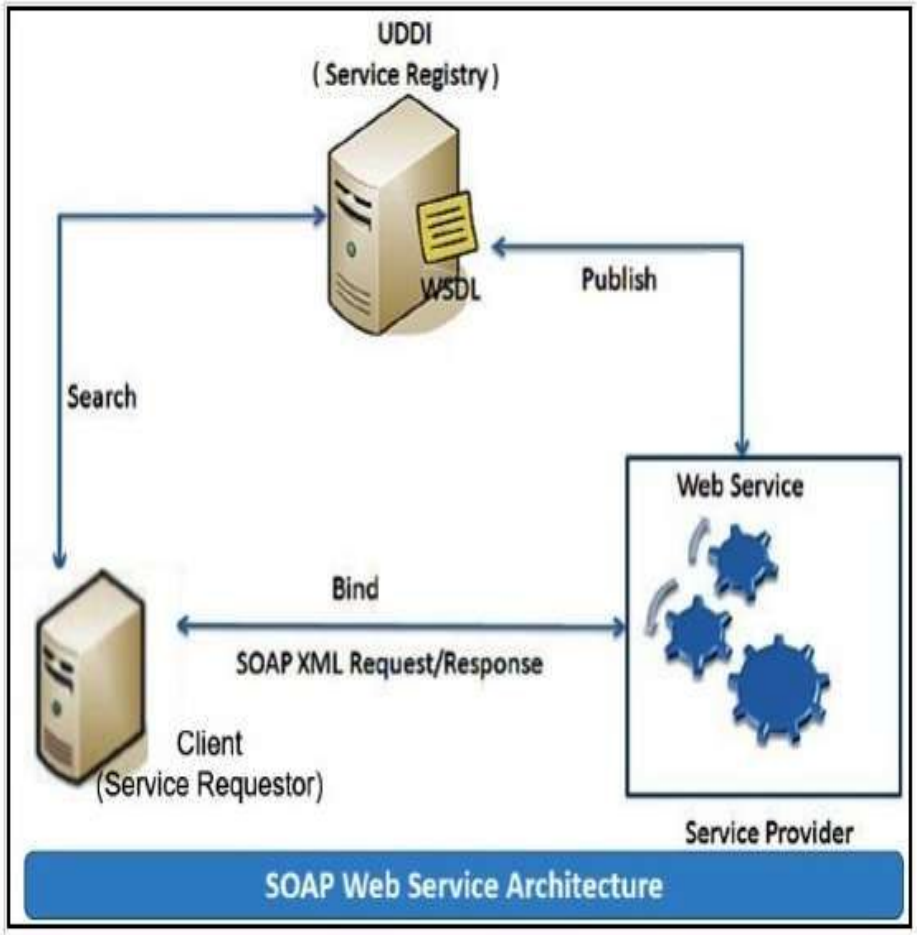
SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it's XML based, it's platform and language independent. So, our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa.

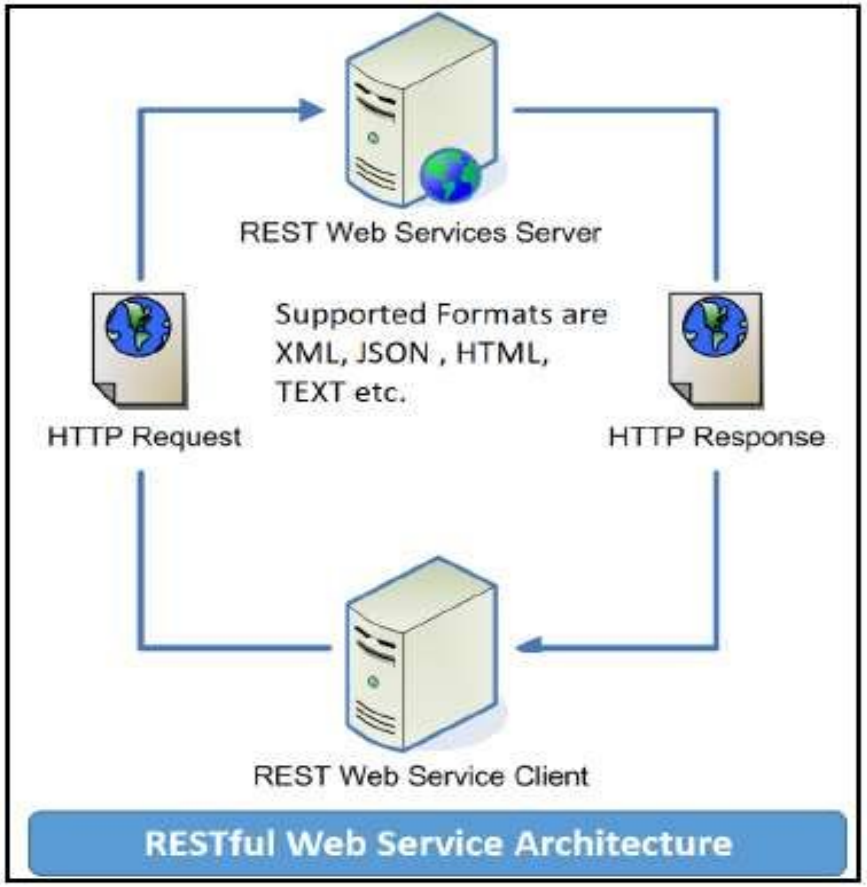
RESTful Web Services-

REST (Representational State Transfer) is an architectural style for developing web services. It's getting popularity recently because it has small learning curve when compared to SOAP. Resources are core concepts of Restful web services and they are uniquely identified by their URIs

SOAP BASED WEB SERVICES Vs. RESTful Web Services

REST stands for **Representational State Transfe**r. RESTful web services are considered a performance-efficient alternative to the SOAP web services. REST is an architectural style, not a protocol.





SOAP vs REST

SOAP	REST
SOAP is a protocol.	REST is an architectural style.
SOAP stands for Simple Object Access Protocol.	REST stands for REpresentational State Transfer.
SOAP can't use REST because it is a protocol.	REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
SOAP uses services interfaces to expose the business logic.	REST uses URI to expose business logic.
JAX-WS is the java API for SOAP web services.	JAX-RS is the java API for RESTful web services.
SOAP defines standards to be strictly followed.	REST does not define too much standards like SOAP.
SOAP requires more bandwidth and resource than REST.	REST requires less bandwidth and resource than SOAP.
SOAP defines its own security.	RESTful web services inherits security measures from the underlying transport.
SOAP permits XML data format only.	REST permits different data format such as Plain text, HTML, XML, JSON etc.
SOAP is less preferred than REST.	REST more preferred than SOAP.

STEPS INVOLVED IN BASIC SOAP WEB SERVICE OPERATIONAL BEHAVIOUR

The following are the steps involved in a basic SOAP web service operational behavior:

1. The client program that wants to interact with another application prepares its request content as a SOAP message.

2. Then, the client program sends this SOAP message to the server web service as an HTTP POST request with the content passed as the body of the request.

3. The web service plays a crucial role in this step by understanding the SOAP request and converting it into a set of instructions that the server program can understand.

4. The server program processes the request content as programmed and prepares the output as the response to the SOAP request.

5. Then, the web service takes this response content as a SOAP message and reverts to the SOAP HTTP request invoked by the client program with this response.

6. The client program web service reads the SOAP response message to receive the outcome of the server program for the request content it sent as a request.

DESIGNING THE SOLUTION

Java provides it's own API to create both SOAP as well as RESTful web services.

1. JAX-WS: JAX-WS stands for Java API for XML Web Services. JAX-WS is XML based Java API to build web services server and client application.

2. JAX-RS: Java API for RESTful Web Services (JAX-RS) is the Java API for creating REST web services. JAX-RS uses annotations to simplify the development and deployment of web services.

Both of these APIs are part of standard JDK installation, so we don't need to add any jars to work with them.

Students are required to implement both i.e. using SOAP and RESTful APIs.

WEB-SERVICES IMPLEMENTATION

- **Steps to implement Web-Services:**

1. Choosing container [Create a Project in Netbeans +Glassfish Server / Eclipse].
2. Creating Web Service from java class
3. Adding an Operation(Methods) to Web Service .
4. Deploying and Testing Web Service.
5. Consuming the Web Service (Creating a client and using web-service).

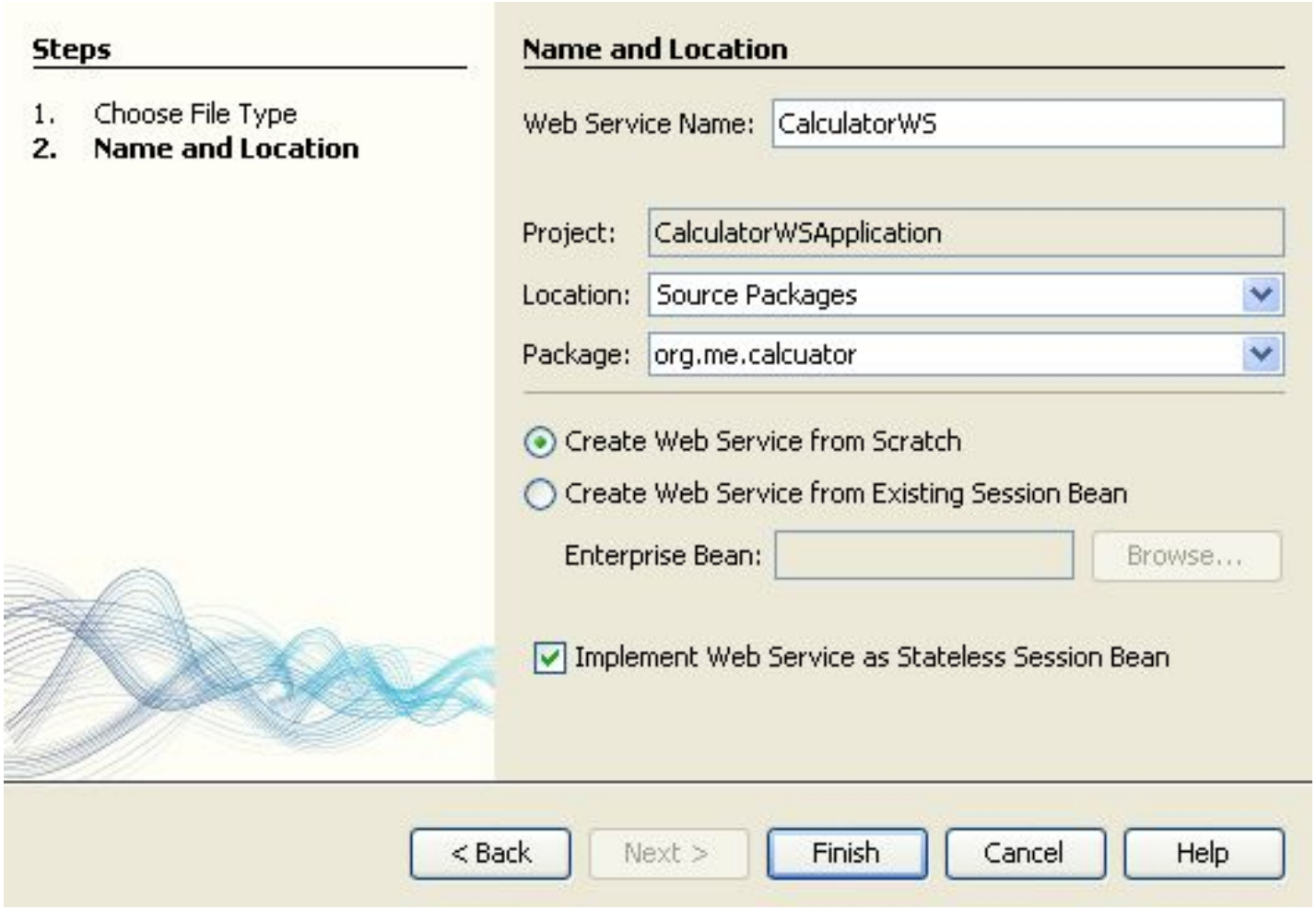
WEB-SERVICES IMPLEMENTATION

- **Step 1: Choosing a container :** You can either deploy your web service in a web container.
- Choose **File** > **New Project** (Ctrl-Shift-N on Linux and Windows).
- Select **Web Application** from the Java **Web** category.
- Name the project `CalculatorWSApplication`.
- Select a location for the project. Click **Next**.
- Select the server **[Glassfish / Tomcat]** and **Java EE** version and click **Finish.**

WEB-SERVICES IMPLEMENTATION

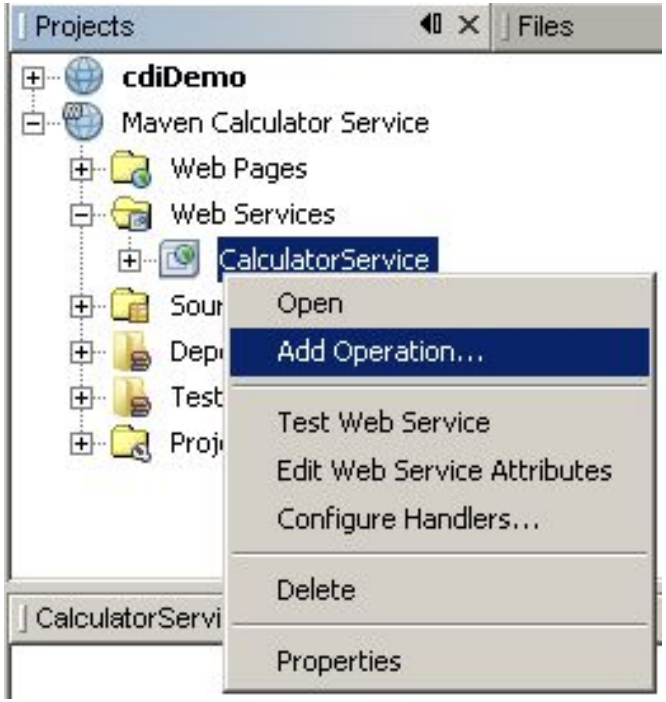
- **Step 2: Creating a Web Service from a Java Class :**
- Right-click the `CalculatorWSApplication` node and **choose New > Web Service**.
- Name the web service `CalculatorWS` and type `org.me.calculator` in Package.
- Keep "**Create Web Service from Scratch**" check box selected.
- If you are creating a Java EE project on GlassFish, select "**Implement Web Service as a Stateless Session Bean"**.
- Click **Finish**. The Projects window displays the structure of the new web service and the source code is shown in the editor area.

WEB-SERVICES IMPLEMENTATION



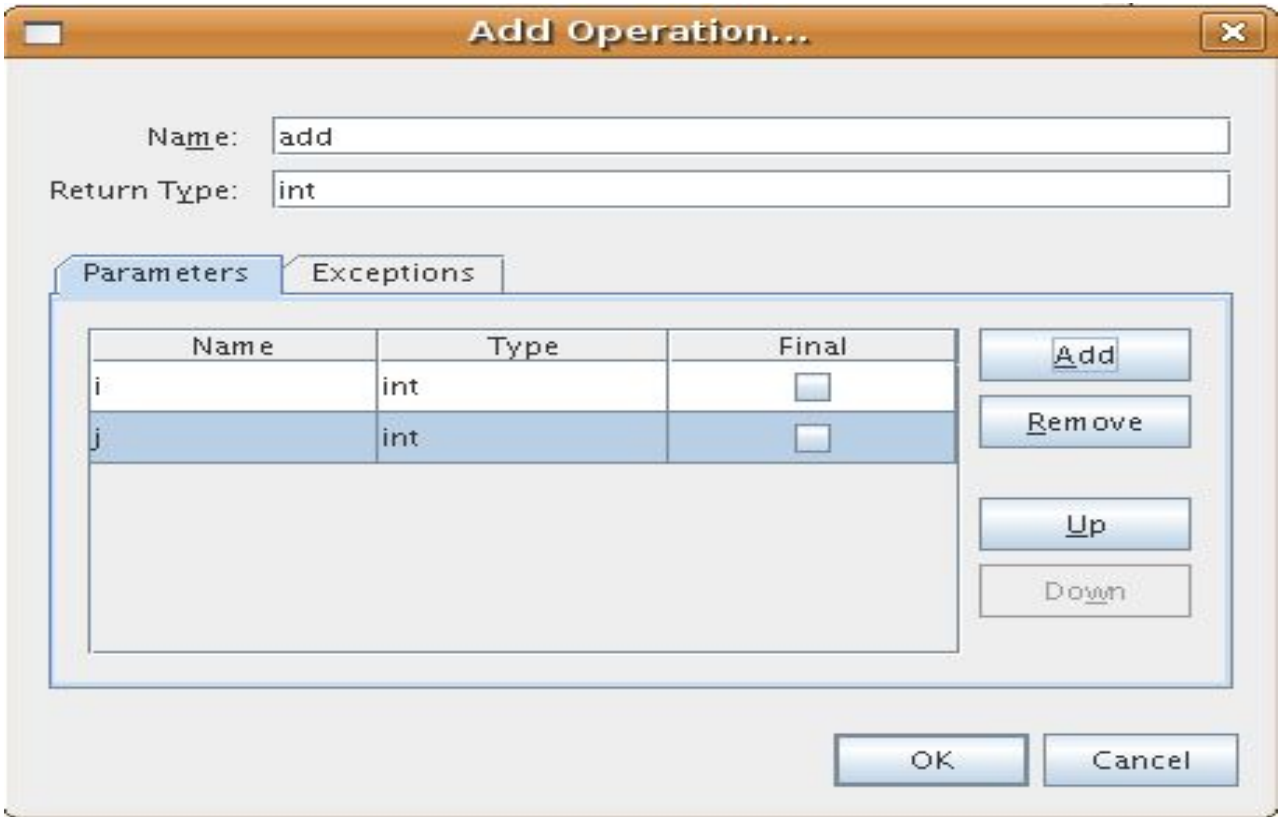
WEB-SERVICES IMPLEMENTATION

- **Step 3: Adding an Operation to the Web Service :**
- Find the web service's node in the Projects window. Right-click that node. A context menu opens.
- Click **Add Operation** in either the visual designer or the context menu. The Add Operation dialog opens.



WEB-SERVICES IMPLEMENTATION

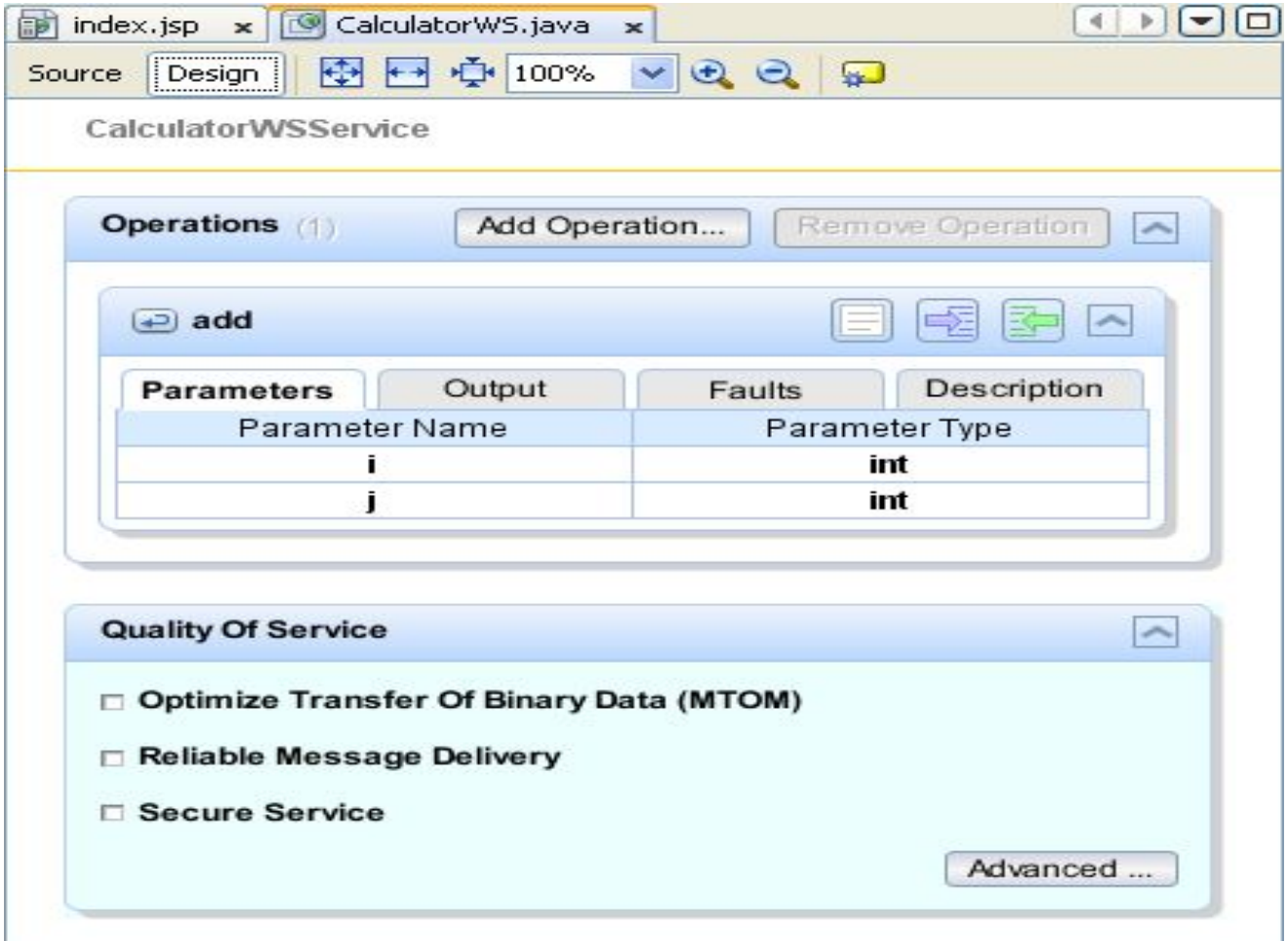
- In the upper part of the **Add Operation dialog box**, type **`add`** in **Name** and type **`int`** in the **Return Type** drop-down list.
- In the lower part of the Add Operation dialog box, click **Add** and create a parameter of type **`int`** named **`i`**. Click **Add** again and create a parameter of type **`int`** called **`j`**.



- Click **OK** at the bottom of the Add Operation dialog box. You return to the editor.

WEB-SERVICES IMPLEMENTATION

- Remove the **default hello operation**, either by deleting the hello() method in the **source code** or by selecting the hello operation in the **visual designer** and clicking **Remove Operation**. The visual designer looks like as below:



WEB-SERVICES IMPLEMENTATION

- Click **Source** menu and can view the generated code as follows:

```
package org.me.calcuator;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.ejb.Stateless;

/**
 *
 * @author jeff
 */
@WebService()
@Stateless()
public class CalculatorWS {

    /**
     * Web service operation
     */
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i")
    int i, @WebParam(name = "j")
    int j) {
        //TODO write your implementation code
        return 0;
    }
}
```

WEB-SERVICES IMPLEMENTATION

- In the editor, extend the skeleton **add operation** to the following (changes are in bold):

```
@WebMethod
public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {
    int k = i + j;
    return k;
  }
```

As you can see from the above code, the web service simply receives two numbers and then returns their sum.

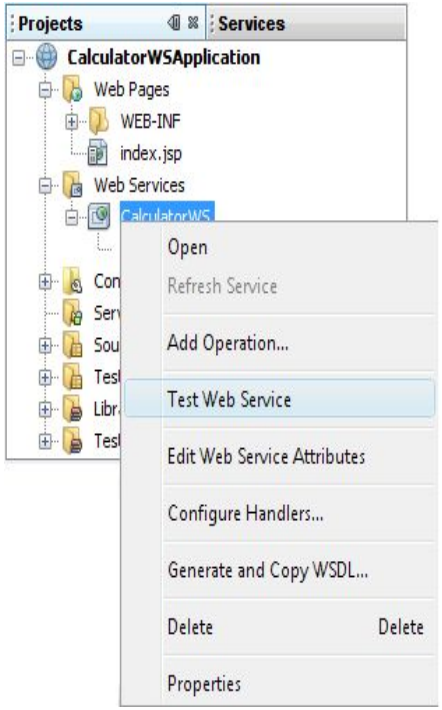
WEB-SERVICES IMPLEMENTATION

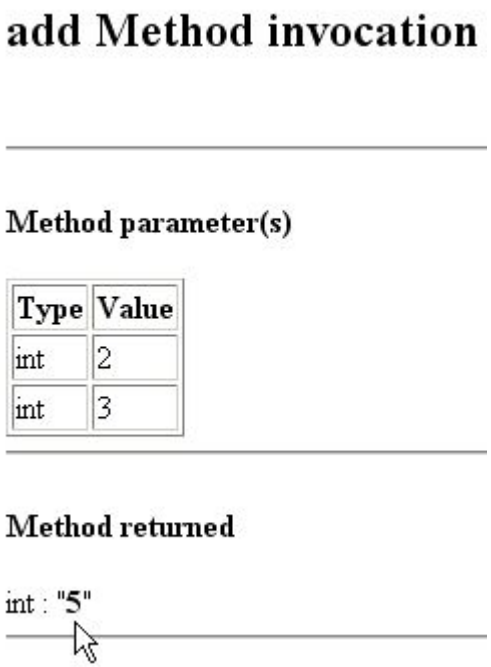
- **Step 4: Deploying and Testing the Web Service :** Once you deploy a web service to a server, you can use the IDE to open the server's test client. The GlassFish server provide test clients whereas in Tomcat Web Server, there is no test client.

- Right-click the **project** and choose **Deploy**. The IDE starts the application server, builds the application, and deploys the application to the server.
- In the IDE's **Projects** tab, expand the **Web Services** node of the `CalculatorWSApplication` project. Right-click the `CalculatorWS` node, and choose **Test Web Service.**

WEB-SERVICES IMPLEMENTATION

- The IDE opens the tester page in the browser, if you deployed a web application to the GlassFish server.





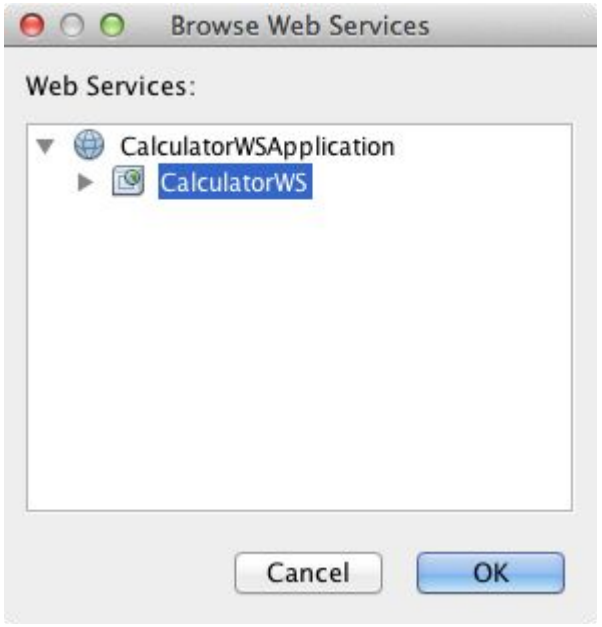
WEB-SERVICES IMPLEMENTATION

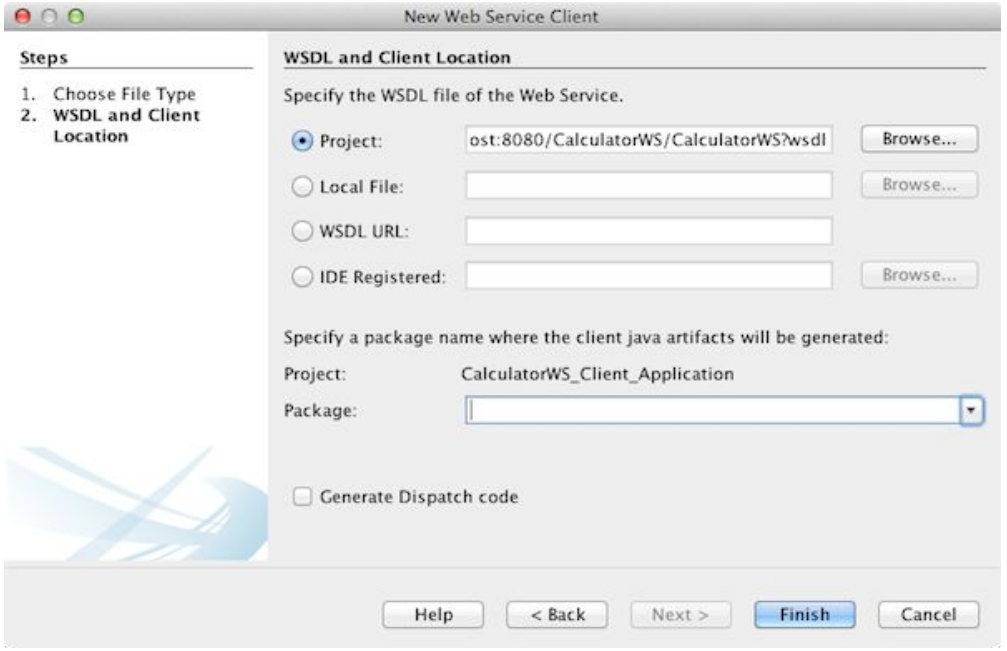
- **Step 5: Consuming the Web Service :** Once the web service is deployed, you need to **create a client** to make use of the web service's **add** method. Here, you can create three types of clients : a Java class in a Java SE application, a servlet, and a JSP page in a web application.

- Client 1: Java Class in Java SE Application
- Choose **File** > **New Project**.
- Select **Java Application** from the **Java** category.
- Name the project `CalculatorWS_Client_Application`.
- Keep "**Create Main Class selected**" and accept all other default settings. Click **Finish**.
- Right-click the `CalculatorWS_Client_Application` node and choose **New** > **Web Service Client**. The New Web Service Client wizard opens.

WEB-SERVICES IMPLEMENTATION

- Select “Project as the **WSDL...** source. Click on **Browse** button. Browse to the CalculatorWS web service in the **CalculatorWSApplication** project. When you have selected the web service, click **OK**.

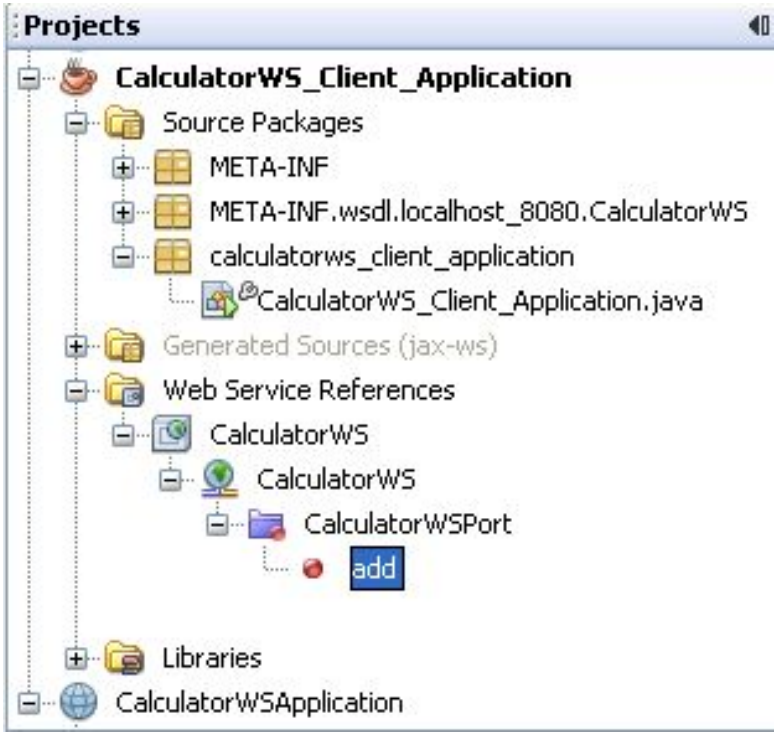




- Do not select **a package** name. Leave this field empty. Keep the other settings at default and click **Finish**.

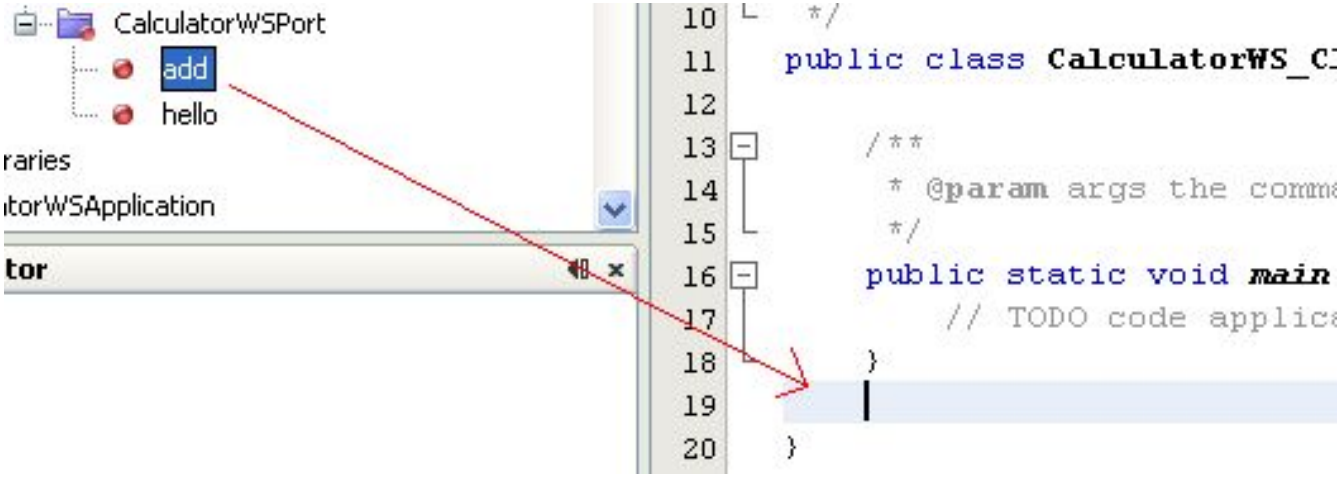
WEB-SERVICES IMPLEMENTATION

- The Projects window displays the new **web service client**, with a node for the **add** method that is created:



WEB-SERVICES IMPLEMENTATION

- Double-click your **main class** so that it opens in the Source Editor. Drag the **add** node below the main() method.



WEB-SERVICES IMPLEMENTATION

Now you can see the code as per below:

```
public static void main(String[] args) {
   // TODO code application logic here
}
private static int add(int i, int j) {
                org.me.calculator.CalculatorWS_Service service = new
org.me.calculator.CalculatorWS_Service();
   org.me.calculator.CalculatorWS port = service.getCalculatorWSPort();
   return port.add(i, j);
}
```

WEB-SERVICES IMPLEMENTATION

- In the main() method body, **replace the TODO comment** with code that initializes values for i and j, calls add(), and prints the result.

```
public static void main(String[] args) {
    try {
        int i = 3;
        int j = 4;
        int result = add(i, j);
        System.out.println("Result = " + result);
    } catch (Exception ex) {
        System.out.println("Exception: " + ex);
    }
}
```

Compile and Execute

- Right-click the project node and choose **Run**.
- The Output window now shows the sum:

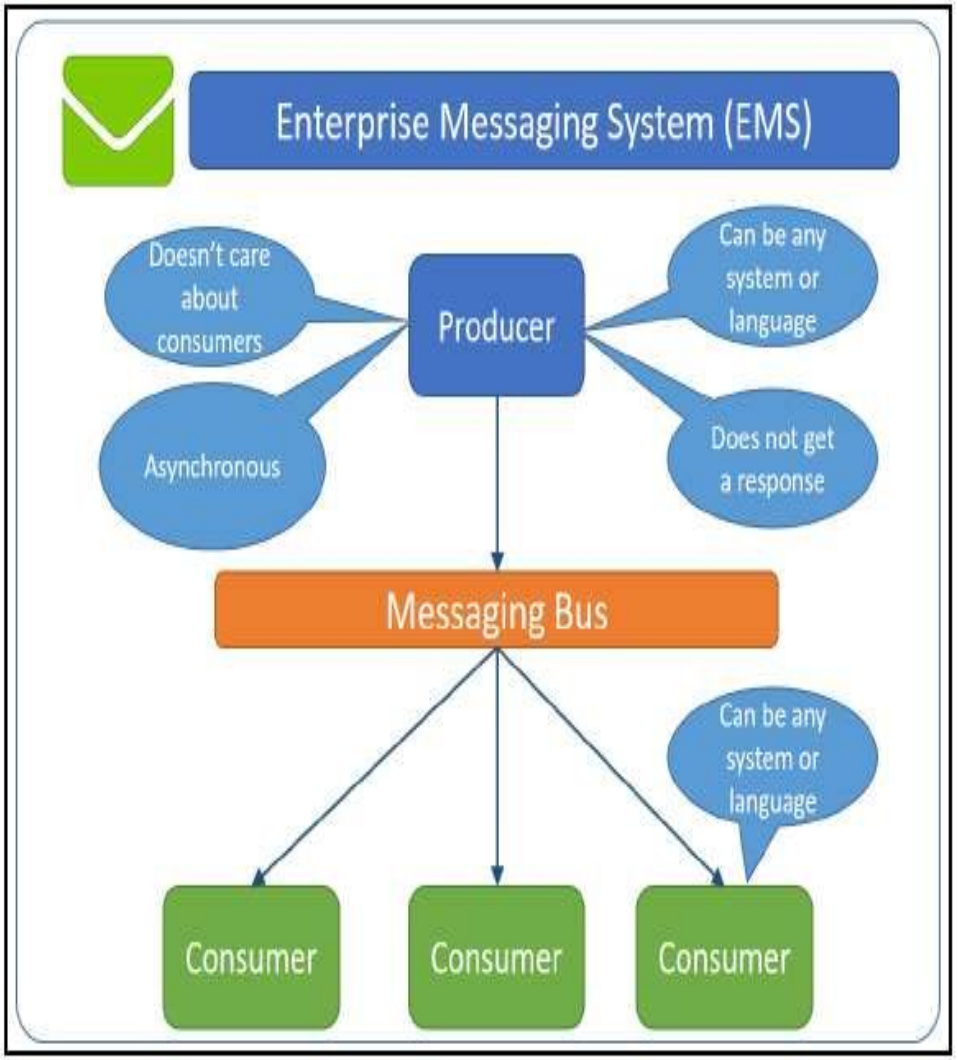
.

```
compile:
run:
Result = 7
  BUILD SUCCESSFUL (total time: 1 second)
```

ASSIGNMENT NO. 6	
Assignment No. 6	To develop any distributed application using Messaging System in **Publish - Subscribe paradigm**.
Objective(s):	By the end of this assignment, the student will be able to create, deploy and test Web-service application.
Tools	Eclipse, Java 8, Apache ActiveMQ

ENTERPRISE MESSAGING SYSTEM

- A **specification/standard** that describes a common way for programs to create, send, receive and read distributed enterprise messages.
- Common formats, such **as XML or JSON,** are used to do this.
- EMS recommends the messaging protocols: Data Distribution Service (DDS), Message Queuing (MSMQ), Advanced Message Queuing Protocol (AMQP), or SOAP web services.
- Systems designed with EMS are termed **Message-Oriented Middleware** (MOM).



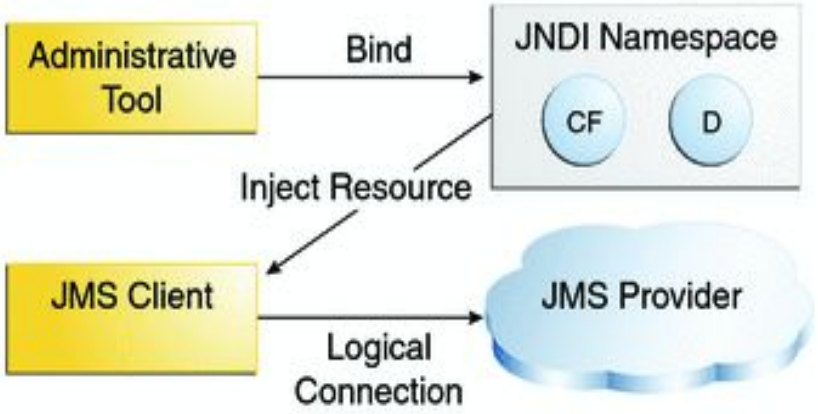
JAVA MESSAGING SERVICE

- The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Messaging enables distributed communication that is **loosely coupled.**
- The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications.
- Messaging is used for communication between software applications or software components.
- A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.

JMS API Architecture

- A JMS application is composed of the following parts:
  - A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features.
  - **JMS clients** are the programs or components, written in the Java programming language, that **produce and consume** messages.
  - **Messages** are the objects that communicate information between JMS clients.
  - **Administered objects** are **preconfigured JMS objects** created by an administrator for the use of clients. The two kinds of JMS administered objects are destinations and connection factories, described in JMS Administered Objects.

•Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.



Public Subscribe Messaging Approach

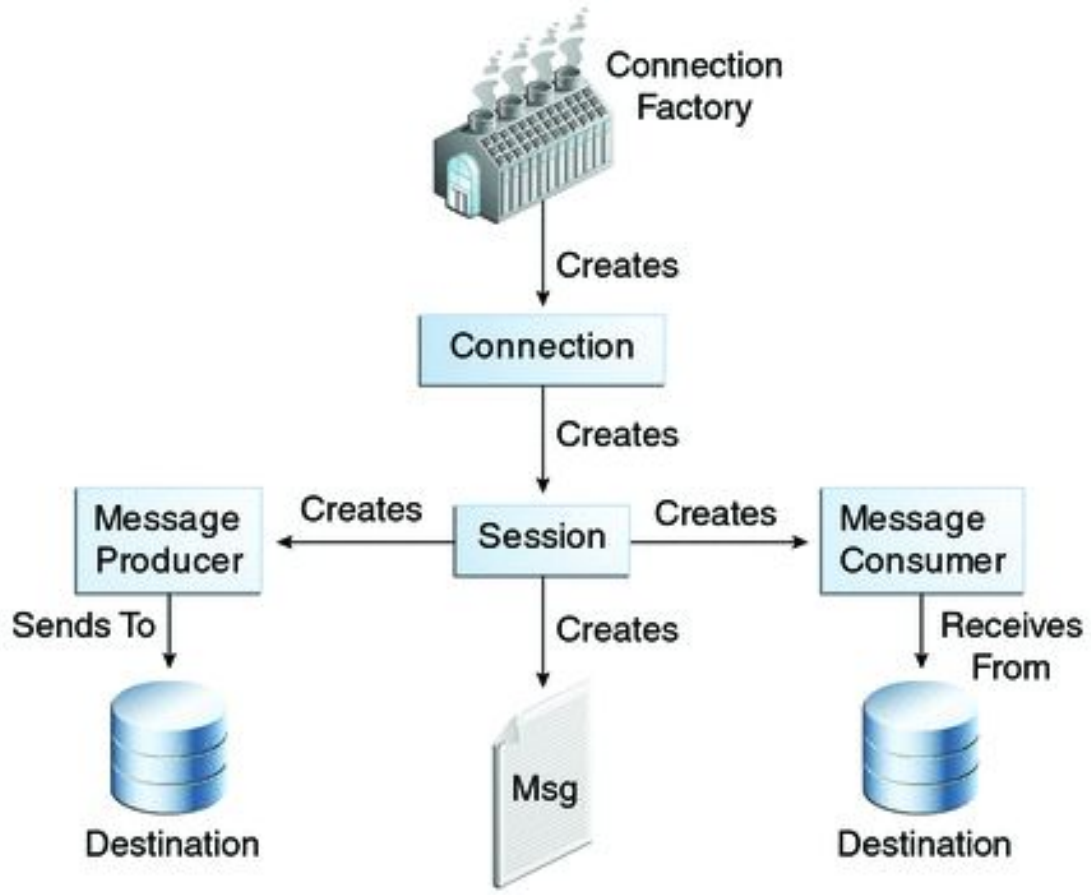
- JMS API support both the point-to-point and the publish/subscribe approach .
- A stand-alone JMS provider can implement one or both domains.
- In a **publish/subscribe** (pub/sub) system, clients address messages to a **topic**, which functions somewhat like a bulletin board.
- Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy.
- The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers.
- Topics retain messages only as long as it takes to distribute them to current subscribers.
- Pub/sub messaging has the following characteristics:
  - Each message can have multiple consumers.
  - Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

Public Subscribe Messaging Approach

- In a **publish/subscribe** (pub/sub) system, clients address messages to a **topic**, which functions somewhat like a bulletin board.

The JMS API Programming Model

- The basic building blocks of a JMS application are:
- **Administered objects:** connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages



The JMS API Programming Model

- The basic building blocks of a JMS application are:
- **Administered objects:** The management of these objects belongs to provider. (connection factories and destinations).
- **JMS Connection Factories:** A **connection factory** is the object a client uses to create a connection to a provider.
- Each connection factory is an instance of the `CoonectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface.
- **JMS Destinations:** A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.
- In the pub/sub messaging domain, destinations are called topics.
- **JMS Connections:** A **connection** encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.
- Before an application completes, you must close any connections you have created.

The JMS API Programming Model

- **JMS Sessions:** A **session** is a single-threaded context for producing and consuming messages.
- Sessions are used to create the Message Producers, Message Consumers, Messages, Topics etc.
- **JMS Message Listeners:** A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

INSTALLING ActiveMQ

- ActiveMQ is an open source message broker written in java.
- **To install the same download Apache ActiveMQ:** http://www.apache.org/dist//activemq/apache-activemq/5.5.0/apache-activemq-5.5.0-bin.tar.gz

```
[suncoma@wso2 ~]$ wget http://www.apache.org/dist//activemq/apache-activemq/5.5.0/apache-activemq-5.5.0-bin.tar.gz
--2011-05-23 04:50:29--  http://www.apache.org/dist//activemq/apache-activemq/5.5.0/apache-activemq-5.5.0-bin.tar.gz
Resolving www.apache.org... 140.211.11.131
Connecting to www.apache.org|140.211.11.131|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 27495046 (26M) [application/x-gzip]
Saving to: `apache-activemq-5.5.0-bin.tar.gz'
```

INSTALLING ActiveMQ

- **Extract the Archive:**
  - If the ActiveMQ start-up script is not executable, change its permissions: `chmod 755 activemq`
- **Run Apache ActiveMQ:**
  - Run ActiveMQ from a command shell:

```
sudo sh activemq start
```

```
[suncoma@wso2 bin]$ sudo sh activemq start
INFO: Using default configuration
(you can configure options in one of these file: /etc/default/activemq /home/suncoma/.activemqrc)

INFO: Invoke the following command to create a configuration file
activemq setup [ /etc/default/activemq | /home/suncoma/.activemqrc ]

INFO: Using java '/usr/bin/java'
INFO: Starting - inspect logfiles specified in logging.properties and log4j.properties to get details
INFO: pidfile created : '/opt/apache-activemq-5.5.0/data/activemq.pid' (pid '4081')
[suncoma@wso2 bin]$
```

INSTALLING ActiveMQ

- **Testing the Installation:**
- ActiveMQ's default port is 61616. From another window, run `netstat` and search for port 61616.

```
netstat -an|grep 61616
```

The port should be open and in LISTEN mode on 61616.

INSTALLING ActiveMQ

- **Monitoring ActiveMQ:**
- You can monitor ActiveMQ, using the Web Console by pointing your browser at:
- `http://localhost:8161/admin`

Publish-Subscriber Implementation

- Create a **new Project** in eclipse with package name **`pubSub`**.
- Create two new class files in the **`src`** directory named **`Publisher.java`** and **`Subscriber.java`**

- In both the java files, import packages java message service (jms) and ActiveMQ.

```
import javax.jms.*;

import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;
```

- `Publisher.java` file will create a **topic** which will be subscribed by the subscribes.

Implementation: Publisher

- In publisher class , the publisher uses the default broker url (URL of JMS server), provided by ActiveMQ.

```
public class Publisher {

    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;
```

- Establish a connection between Publisher and ActiveMQ.
  - Initialize connection between Publisher and the above url using `ActiveMQConnectionFactory()` method and create it with `createConnection()` method.
  - `start()` method starts the created connection.

```
11⊖   public static void main(String[] args) throws JMSException {
12
13        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
14        Connection connection = connectionFactory.createConnection();
15        connection.start();
```

Implementation: Publisher

- Create a session using `createSession()` method

```
17          // JMS messages are sent and received using a Session. We will
18          // create here a non-transactional session object. If you want
19          // to use transactions you should set the first parameter to 'true'
20          Session session = connection.createSession(false,
21                  Session.AUTO_ACKNOWLEDGE);
22
```

- Create a new topic to be published by the publisher with the `createTopic()` method.

```
23          //Set the topic to which the Subscriber will subscribe to
24          Topic topic = session.createTopic("DistributedSystem");
25
```

Implementation: Publisher

- Create a producer object and assign the topic to it from the previous step.

```
26          //Give the topic to the producer
27          MessageProducer producer = session.createProducer(topic);
28
29          // We will send a small text message saying "1.Chapter One"
30          TextMessage message = session.createTextMessage();
31          message.setText("1.Chapter One");
```

- Producer can create and send new messages for the topic using the `createTextMessage()` and `send()` method

```
33          // Here we are sending the message!
34          producer.send(message);
35          System.out.println("Sent message '" + message.getText() + "'");
36
37          connection.close();
```

Implementation: Subscriber

- Create a Subscriber class and create a connection similar to Producer.

```
3  import java.io.IOException;
4
5  import javax.jms.*;
6
7  import org.apache.activemq.ActiveMQConnection;
8  import org.apache.activemq.ActiveMQConnectionFactory;

10 public class Subscriber {
11     // URL of the JMS server
12     private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;

14     // Name of the topic from which we will receive messages from = "DistributedSystem"
15     public static void main(String[] args) throws JMSException {
16         // Getting JMS connection from the server
17         ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
18         Connection connection = connectionFactory.createConnection();
19         connection.start();
```

Implementation: Subscriber

- A subscriber listens to the topics which are published by a publisher by subscribing to the topic .

```
21          //Create a session
22          Session session = connection.createSession(false,
23                  Session.AUTO_ACKNOWLEDGE);
24
25          //This topic is the same as in Publisher
26          Topic topic = session.createTopic("DistributedSystem");
27
```

- The `createConsumer()` method takes the topic as an argument.

```
28          //Give the topic to the producer
29          MessageConsumer consumer = session.createConsumer(topic);
30
```

Implementation: Subscriber

- A `MessageListener` object is used to receive the delivered messages. The `MessageListener` interface has a `onMessage()` method which passes the message to the listener.

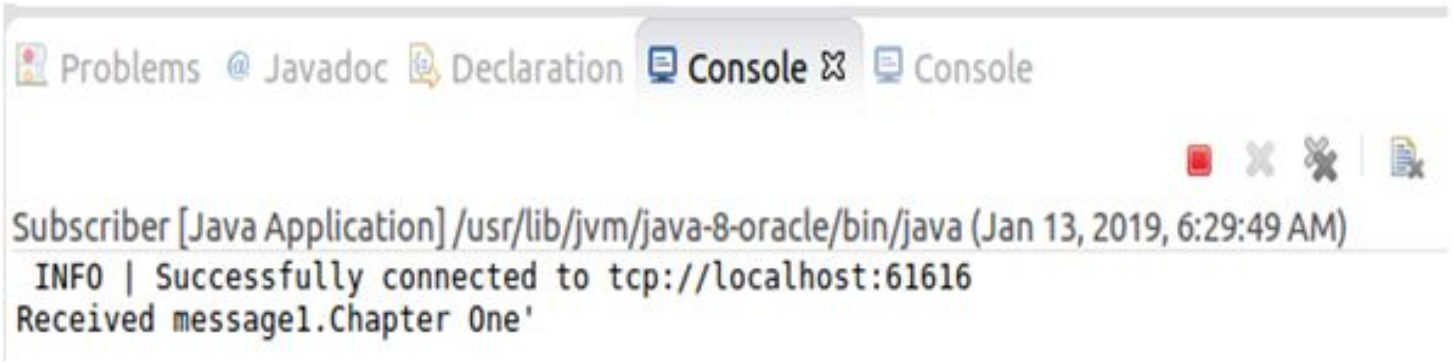
```
34        MessageListener listner = new MessageListener() {
35            public void onMessage(Message message) {
36                try {
37                    if (message instanceof TextMessage) {
38                        TextMessage textMessage = (TextMessage) message;
39                        System.out.println("Received message"
40                                + textMessage.getText() + "'");
41                    }
42                } catch (JMSException e) {
43                    System.out.println("Caught:" + e);
44                    e.printStackTrace();
45                }
46            }
47        };
49        consumer.setMessageListener(listner);
```

- Finally we set the message listener and close the connection.

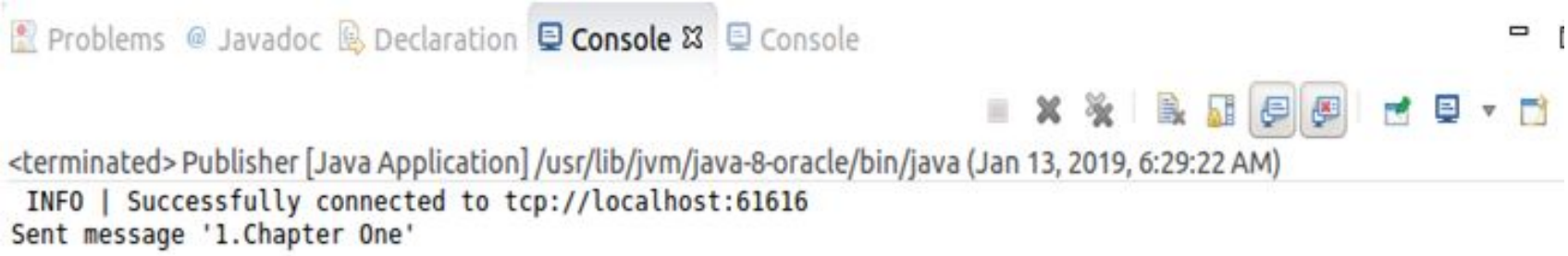
```
connection.close();
```

Steps to Run and Expected Output

- Run `Subscriber.java` in Eclipse



- Run `Publisher.java` in Eclipse. (After this step the Subscriber.java will show a new Message)



References

JMS API Programming Model:

https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html

ASSIGNMENT NO. 7	
Assignment No. 7	To develop **Microservices** framework based distributed application.
Objective(s):	By the end of this assignment, the student will be able to create, deploy and test Web-service application.
Tools	Python 3, Flask

Microservices

- The way developer worked to build application is changing. In the past, software was built as large monolithic application where a team developers would take months to construct a large application built on a common code base.
- Define an architecture that structures the application as a set of loosely coupled, collaborating services. Each service implements a set of narrowly, related functions.
- For example, an application might consist of services such as the order management service, the customer management service etc.
- Each service has its own database in order to be decoupled from other services. Data consistency between services is maintained .

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities.

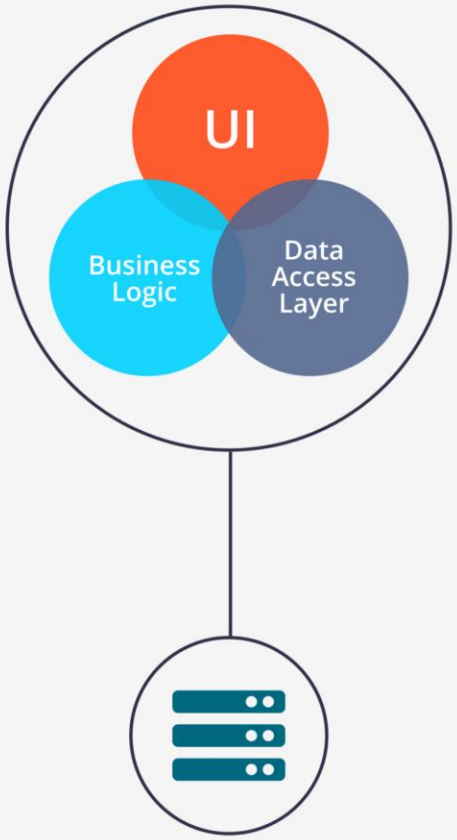
Monolithic Design vs Microservice

- Traditional application design is often called “monolithic” because the whole thing is developed in one piece.
- Even if the logic of the application is modular it’s deployed as one group, like a Java application as a JAR file for example.
- This monolith eventually becomes so difficult to manage as the larger applications require longer and longer deployment timeframes.

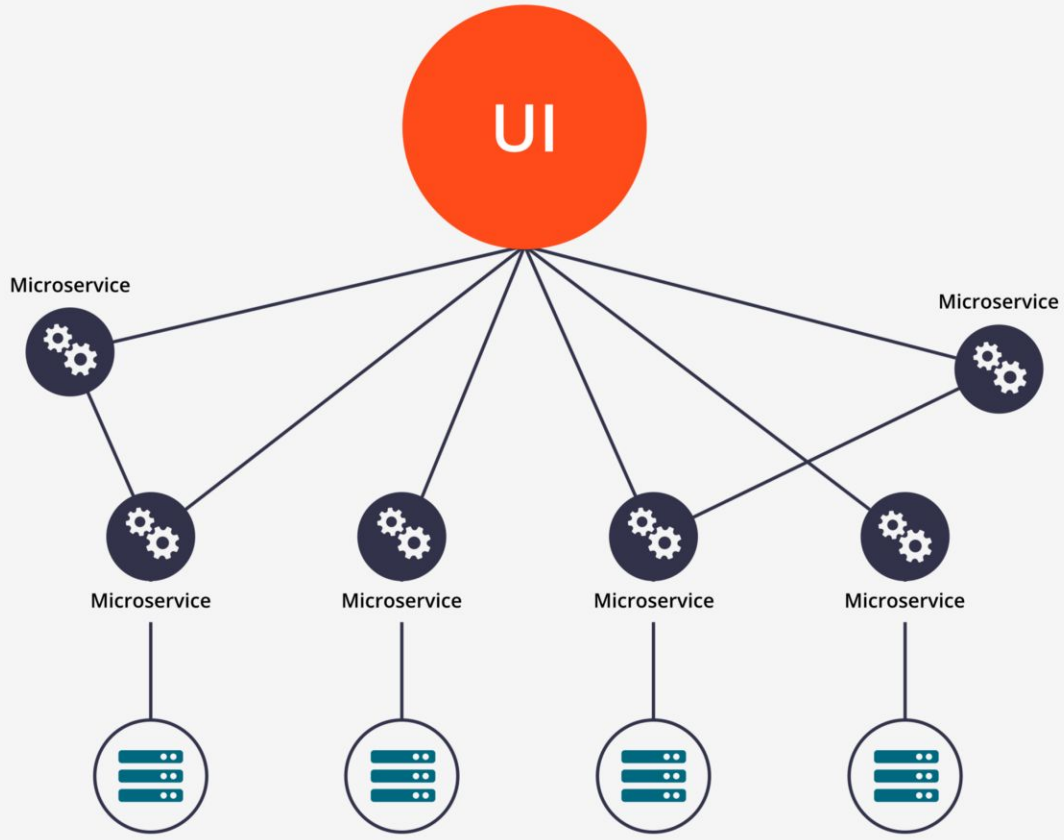
- In contrast, a team designing a microservices architecture for their application will split all of the major functions of an application into independent services.

- Each independent service is usually packaged as an API so it can interact with the rest of the application elements.

Monolithic Design vs Microservice



Monolithic Architecture



Microservice Architecture

Web frameworks

- Web frameworks encapsulate what developers have learned over the past twenty years while programming sites and applications for the web.
- Frameworks make it easier to reuse code for common HTTP operations and to structure projects so other developers with knowledge of the framework can quickly build and maintain the application.
- Common web framework functionality: Frameworks provide functionality in their code or through extensions to perform common operations required to run web applications.
    1. URL routing
    2. Input form handling and validation
    3. HTML, XML, JSON, and other output formats with a templating engine
    4. Database connection configuration and persistent data manipulation through an object-relational mapper (ORM) Web security against Cross-site request forgery (CSRF), SQL Injection, Cross-site Scripting (XSS) and other common malicious attacks.
    5. Session storage and retrieval.

Web frameworks

- **Flask**

  Flask (source code) is a Python web framework built with a small core and easy-to-extend philosophy. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine.

- **WSGI**

Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.

- **Werkzeug**

It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

Virtual Environment

- In Python, by default, every project on the system will use the same directories to store and retrieve site packages (third party libraries) and system packages (packages that are part of the standard Python library).
- Consider the a scenario where there are two projects: ProjectA and ProjectB, both have a **dependency on the same library**, ProjectC. The problem becomes apparent when we start requiring **different versions** of ProjectC.

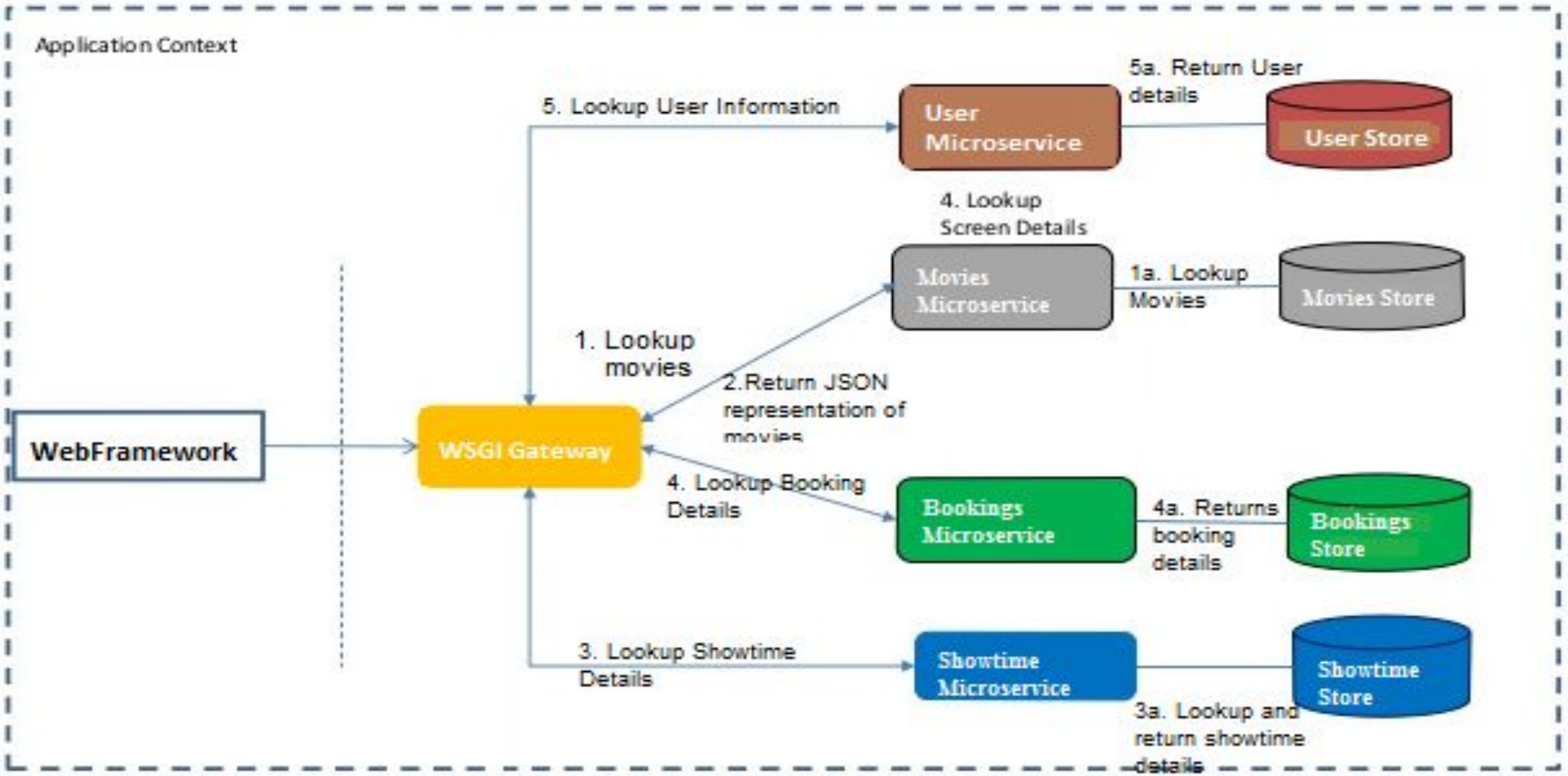
  Maybe ProjectA needs v1.0.0, while ProjectB requires the newer v2.0.0, for example.

Since projects are stored in site-packages directory according to their name and can't differentiate between versions, both projects, ProjectA and ProjectB, would be required to use the same version which is unacceptable in many cases and hence the virtual environment.

The main purpose of Python virtual environments is to create an isolated environment for Python projects. This means that each project can have its own dependencies, regardless of what **dependencies** every other project has.

Virtual Environment

Here, we are attempting to develop a microservice based architecture for Movie ticket Booking web application. The services are being implemented using python and JSON is used as for Data Store.



Implementing the Solution

- **Using Virtual Environments:** Install `virtualenv` for development environment. `virtualenv` is a virtual Python environment builder. It helps a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries.

  The following command installs `virtualenv`:

  ```
  sudo apt-get install virtualenv
  ```

- **Flask Module**: Importing flask module in the project is mandatory. An object of Flask class is our WSGI application. Flask constructor takes the name of current module (__name__) as argument. The route() function of the Flask class is a decorator, which tells the application which URL should call the associated function.

Implementing the Solution

- **Route Decoder:**

  The `route()` decorator in Flask is used to bind URL to a function.

  For example –

```
@app.route('/hello')
def hello_world():
return 'hello world'
```

  Here, URL '/hello' rule is bound to the hello_world() function. As a result, if a user visits http://localhost:5000/hello URL, the output of the hello_world() function will be rendered in the browser.

- **Writing the subroutine for the four microservices:** There are four microservices viz., user, Showtimes, Bookings and Movies for which microservices are to be implemented.

Expected Output

- To install the necessary files and create a virtual environment run:

```
sudo ./setup.sh
```

- To start the 4 microservices run :

```
./startup.sh
```

- To start the command line UI:

```
python cmdline.py
```

Expected Output

Running startup.sh

```
dos@dospc ~/Desktop/Junaid/microservices-20181226T110346Z-001/microservices ./startup.sh
dos@dospc ~/Desktop/Junaid/microservices-20181226T110346Z-001/microservices  * Running on http://127.0.0.1:
5003/ (Press CTRL+C to quit)
 * Running on http://127.0.0.1:5001/ (Press CTRL+C to quit)
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Restarting with stat
 * Restarting with stat
 * Running on http://127.0.0.1:5002/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger is active!
 * Debugger is active!
 * Debugger PIN: 229-444-055
 * Debugger PIN: 229-444-055
 * Debugger PIN: 229-444-055
 * Debugger is active!
 * Debugger PIN: 229-444-055
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:41] "GET /movies HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:44] "GET /showtimes HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:44] "GET /movies HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:53] "GET /bookings/Shreyas HTTP/1.1" 404 -
```

Expected Output

Running cmdline.py

```
dos@dospc ~/Desktop/Junaid/microservices-20181226T1103462-001/microservices python cmdline.py
Welcome to cinema app
1.Get Movie list
2.Get Show Times
3.Get Bookings Info
4.Get User list
5.Book a show
6.Clearscreen
7.Exit
Select an option
1
ID:  276c79ec-a26a-40a6-b3d3-fb242a5947b6
Title:  Avengers Infinity War
Director:  Anthony Russo
Rating:  9.8

ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae
Title:  Stree
Director:  Amar Kaushik
Rating:  9.2

ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6
Title:  Mission Impossible 6
Director:  Christopher McQuarrie
Rating:  9.5
```

Expected Output

```
1.Get Movie list
2.Get Show Times
3.Get Bookings Info
4.Get User list
5.Book a show
6.Clearscreen
7.Exit
Select an option
2
On date: 20180801
ID:  267eedb8-0f5d-42d5-8f43-72426b9fb3e6  MOVIE:  Karwaan
ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6  MOVIE:  Mission Impossible 6
ID:  39ab85e5-5e8e-4dc5-afea-65dc368bd7ab  MOVIE:  The Incredibles 2
ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae  MOVIE:  Stree
On date: 20180803
ID:  720d006c-3a57-4b6a-b18f-9b713b073f3c  MOVIE:  Happy Phirr Bhag Jayegi
ID:  39ab85e5-5e8e-4dc5-afea-65dc368bd7ab  MOVIE:  The Incredibles 2
On date: 20180802
ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae  MOVIE:  Stree
ID:  96798c08-d19b-4986-a05d-7da856efb697  MOVIE:  Gold
ID:  39ab85e5-5e8e-4dc5-afea-65dc368bd7ab  MOVIE:  The Incredibles 2
ID:  276c79ec-a26a-40a6-b3d3-fb242a5947b6  MOVIE:  Avengers Infinity War
On date: 20180805
ID:  96798c08-d19b-4986-a05d-7da856efb697  MOVIE:  Gold
ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae  MOVIE:  Stree
ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6  MOVIE:  Mission Impossible 6
```

Expected Output

```
1.Get Movie list
2.Get Show Times
3.Get Bookings Info
4.Get User list
5.Book a show
6.Clearscreen
7.Exit
Select an option
4
Anuja Kharatmol
Souparnika Patil
Vasundhara Kurtakoti
Yojane Mane
Nachiket Ghorpade
Nayana Patil
Kamraj Ambalkar
```

Expected Output

```
1.Get Movie list
2.Get Show Times
3.Get Bookings Info
4.Get User list
5.Book a show
6.Clearscreen
7.Exit
Select an option
5
>Please enter username for the booking : souparnika_patil
>Please enter the date for the booking : 20180805



ID:  96798c08-d19b-4986-a05d-7da856efb697
Title:  Gold
Director:  Reema Kagdi
Rating:  7.4

ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae
Title:  Stree
Director:  Amar Kaushik
Rating:  9.2

ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6
```

Expected Output

```
ID:  39ab85e5-5e8e-4dc5-afea-65dc368bd7ab
Title:  The Incredibles 2
Director:  Brad Bird
Rating:  7.1

ID:  276c79ec-a26a-40a6-b3d3-fb242a5947b6
Title:  Avengers Infinity War
Director:  Anthony Russo
Rating:  9.8

>Enter the id for the booking : 276c79ec-a26a-40a6-b3d3-fb242a5947b6



Booking the show for the following movie on date 20180802
ID:  276c79ec-a26a-40a6-b3d3-fb242a5947b6
Title:  Avengers Infinity War
Director:  Anthony Russo
Rating:  9.8

Press enter to continue

BOOKING DONE!! Thank you for using Cinema app
```

THANK YOU