**LAB : 1**

1. **Import Statement**:

import java.rmi.\*;

This line imports classes from the **java.rmi** package, which is used for Remote Method Invocation (RMI) in Java.

1. **Interface Definition**:

public interface AddServerIntf extends Remote {

This line defines an interface named **AddServerIntf** that extends the **Remote** interface. In RMI, interfaces that define remote methods must extend the **Remote** interface to mark them as remote.

1. **Method Declaration**:

double add(double d1, double d2) throws RemoteException;

This line declares a method named **add** inside the interface. It specifies that the method takes two double values (**d1** and **d2**) as parameters, adds them together, and returns the result as a double value. Additionally, it declares that it may throw a **RemoteException**. In RMI, methods declared in remote interfaces must declare **RemoteException** because remote communication may encounter network-related issues.

1. **Explanation**:
   * This interface serves as a contract or blueprint for defining the methods that will be accessible remotely.
   * In this case, it defines a single method **add**, which is intended to be invoked remotely to perform addition.
   * Any class that implements this interface is expected to provide an implementation for the **add** method.
   * When a client interacts with a server in an RMI application, it uses this interface to know what methods it can call remotely.
2. **Import Statements**:

import java.rmi.\*; import java.rmi.server.\*;

These lines import necessary classes from the **java.rmi** and **java.rmi.server** packages, which are used for Remote Method Invocation (RMI) in Java.

1. **Class Definition**:

public class AddServerImpl extends UnicastRemoteObject implements AddServerIntf {

This line defines a class named **AddServerImpl** that extends **UnicastRemoteObject** and implements the **AddServerIntf** interface.

1. **Constructor**:

public AddServerImpl() throws RemoteException { }

This is the constructor of the **AddServerImpl** class. It doesn't do anything specific but declares that it can throw a **RemoteException**, which is required because it extends **UnicastRemoteObject**.

1. **Method Implementation**:

public double add(double d1, double d2) throws RemoteException { return d1 + d2; }

This method implements the **add** method declared in the **AddServerIntf** interface. It takes two double values as parameters, adds them together, and returns the result.

1. **Import Statements**:

java

Copy code

import java.rmi.\*;

This line imports the necessary classes from the **java.rmi** package, which is used for Remote Method Invocation (RMI) in Java.

1. **Class Definition**:

public class AddServer {

This line defines a class named **AddServer**.

1. **Main Method**:

public static void main(String args[]) {

This line defines the main method, which serves as the entry point of the program. It accepts command-line arguments as an array of strings, although in this case, it doesn't use them.

1. **Try-Catch Block**:

try {

The program enters a try block, where code that may throw exceptions is placed. This allows for graceful handling of exceptions.

1. **Creating Remote Object**:

AddServerImpl addServerImpl = new AddServerImpl();

This line creates an instance of the **AddServerImpl** class. **AddServerImpl** presumably implements the **AddServerIntf** interface, which defines the methods that can be invoked remotely.

1. **Binding Remote Object**:

Naming.rebind("AddServer", addServerImpl);

This line binds the remote object to the RMI registry using the name "AddServer". When clients look up this name, they will get a reference to the **addServerImpl** object. This effectively makes the **addServerImpl** object available for remote method invocation.

1. **Exception Handling**:

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

Any exceptions thrown within the try block are caught here. The exception object **e** is printed to the console to provide information about the error.

1. **Import Statements**:

import java.rmi.\*;

This line imports the necessary classes from the **java.rmi** package, which is used for Remote Method Invocation (RMI) in Java.

1. **Class Definition**:

public class AddClient {

This line defines a class named **AddClient**.

1. **Main Method**:

public static void main(String args[]) {

This line defines the main method, which serves as the entry point of the program. It accepts command-line arguments as an array of strings.

1. **Try-Catch Block**:

try {

The program enters a try block, where code that may throw exceptions is placed. This allows for graceful handling of exceptions.

1. **RMI Lookup**:

String addServerURL = "rmi://" + args[0] + "/AddServer";

This line constructs the URL used to look up the remote object. It concatenates **"rmi://"** with the first command-line argument (**args[0]**), which is expected to be the hostname or IP address where the RMI registry is running, and **"/AddServer"** to specify the name of the remote object.

1. **Naming.lookup()**:

AddServerIntf addServerIntf = (AddServerIntf) Naming.lookup(addServerURL);

This line uses the **Naming.lookup()** method to obtain a reference to the remote object. It takes the URL constructed earlier and returns a reference to the remote object casted to **AddServerIntf**, which is the interface that defines the methods available on the remote object.

1. **Parsing Command-line Arguments**:

double d1 = Double.parseDouble(args[1]);

This line parses the second command-line argument (**args[1]**) to a double value and assigns it to the variable **d1**.

double d2 = Double.parseDouble(args[2]);

Similarly, this line parses the third command-line argument (**args[2]**) to a double value and assigns it to the variable **d2**.

1. **Remote Method Invocation**:

System.out.println("The sum is: " + addServerIntf.add(d1, d2));

This line invokes the **add()** method on the remote object referenced by **addServerIntf**. It passes the parsed double values **d1** and **d2** as arguments to the remote method. The result is printed to the console.

1. **Exception Handling**:

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

Any exceptions thrown within the try block are caught here. The exception object **e** is printed to the console to provide information about the error.

1. **AddServerImpl.java**:
   * This file represents Bob in our analogy. It's the server-side code that actually performs the addition operation.
   * In detail:
     + It implements the **AddServerIntf** interface, which defines the method **add(double d1, double d2)**.
     + It extends **UnicastRemoteObject**, which makes it available to accept remote method invocations.
     + The **add** method is where the actual addition operation happens. This method receives two numbers from the client, adds them together, and returns the result.
2. **AddServer.java**:
   * This file is responsible for running the server and making the **AddServerImpl** object available to clients.
   * In detail:
     + It creates an instance of **AddServerImpl** and binds it to a name ("AddServer") in the RMI registry so that clients can look it up and communicate with it.
3. **AddClient.java**:
   * This file represents Alice in our analogy. It's the client-side code that sends numbers to be added to the server and receives the result.
   * In detail:
     + It looks up the remote object (**AddServerIntf**) in the RMI registry using the server's URL.
     + It takes two numbers as input from the user (representing what Alice wants to add) and sends them to the server using the **add** method.
     + It receives the result from the server and displays it to the user.
4. **AddServerIntf.java**:
   * This file defines the interface that both the server and client use to communicate.
   * In detail:
     + It declares the method **add(double d1, double d2)** which the server implements and the client calls remotely.
     + Both the server and client need to know about this interface so they can communicate effectively.

A remote object is an object that exists in one Java Virtual Machine (JVM) but can be accessed and used by code running in another JVM, possibly on a different physical machine. In simpler terms, it's an object that can be used by code in a different location or process.

In the context of RMI (Remote Method Invocation), when we say "remote object," we're referring to an object that is made accessible remotely over the network. This allows methods of that object to be invoked from another Java program running on a different machine.

RMI, or Remote Method Invocation, is a Java API that allows communication between different Java Virtual Machines (JVMs) over a network. It enables Java objects to invoke methods on objects located in remote JVMs as if they were local objects. RMI provides a mechanism for distributed computing, where Java programs running on different machines can interact with each other seamlessly.

Here's how RMI works and its key components:

1. **Remote Interface**: RMI starts with defining a remote interface. This interface specifies the methods that can be invoked remotely by clients. The methods declared in the interface must throw **RemoteException** to handle remote invocation errors. This interface acts as a contract between the client and server, ensuring that they agree on the methods that can be called remotely.
2. **Remote Object**: In RMI, there's a remote object that implements the remote interface. This object contains the actual implementation of the methods declared in the remote interface. The server makes an instance of this remote object available to clients so that they can invoke its methods remotely.
3. **RMI Registry**: The RMI registry acts as a lookup service where clients can find the remote objects they need to interact with. Servers bind their remote objects to names in the registry, making them accessible to clients. Clients then look up these objects by name in the registry to obtain a reference to them.
4. **Stubs and Skeletons (deprecated)**: In older versions of Java, stubs and skeletons were used to facilitate communication between the client and server. Stubs acted as proxies for remote objects on the client side, while skeletons acted as proxies for remote objects on the server side. However, starting from Java 5, stubs and skeletons are generated automatically by the JVM and are no longer required to be implemented manually.

Relevant Parts of Our Code:

AddServerIntf.java: This file defines the remote interface for our RMI implementation. It declares the add(double d1, double d2) method, which clients can invoke remotely.

AddServerImpl.java: This class implements the remote interface (AddServerIntf). It represents the remote object that provides the actual implementation of the add method.

AddServer.java: This file is responsible for running the RMI server. It creates an instance of AddServerImpl and binds it to a name in the RMI registry.

AddClient.java: This file represents the RMI client. It looks up the remote object (AddServerIntf) in the RMI registry and invokes its methods remotely.