Title : Write a program for Addition and Subtraction using concept of RMI programming

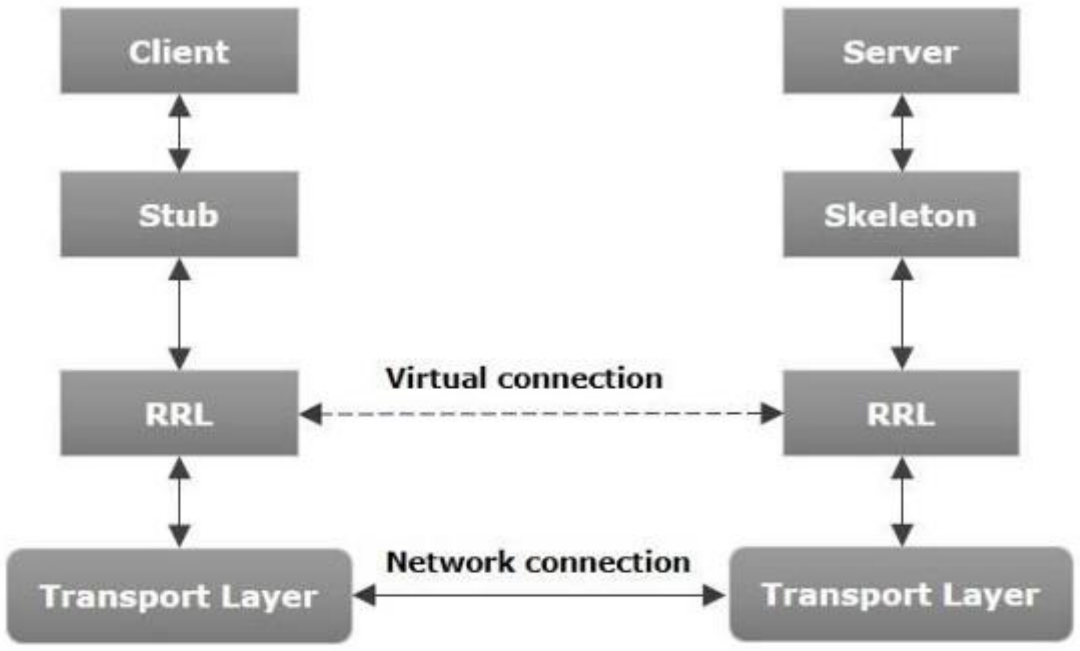
RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

Architecture of an RMI Application

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).

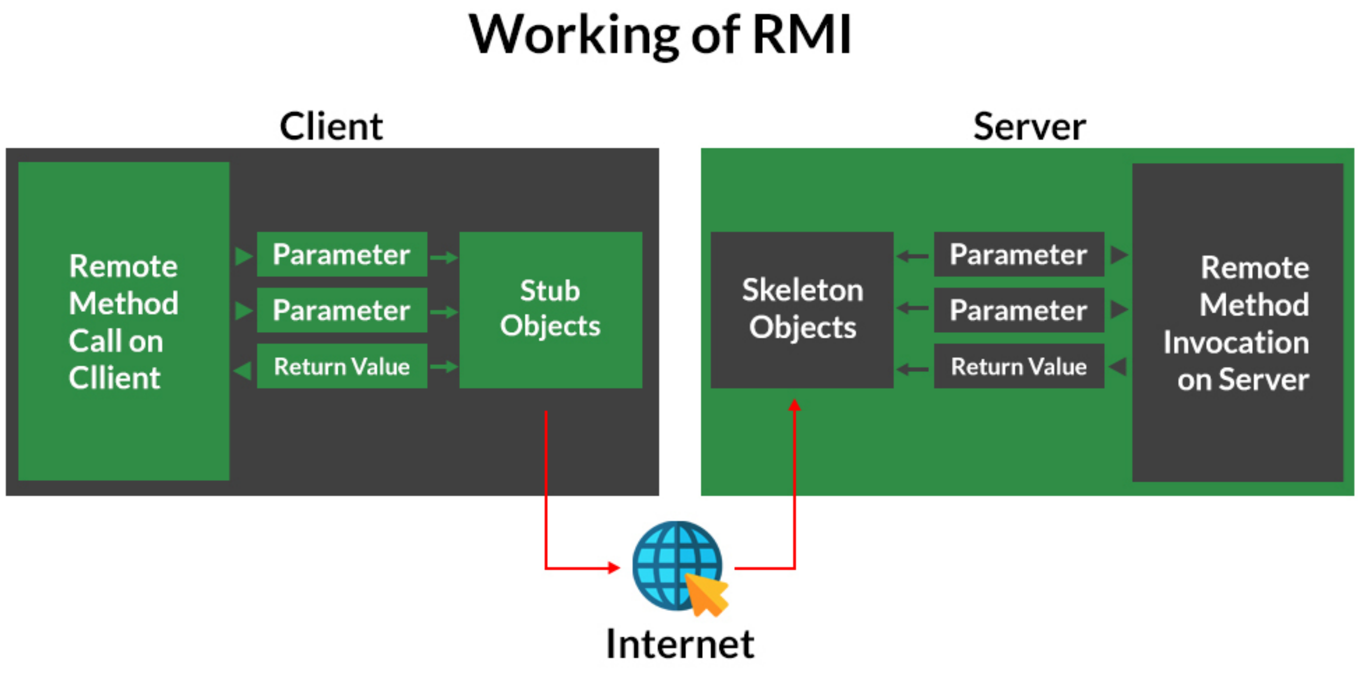
- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.



- **Transport Layer –** This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub –** A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton –** This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer) –** It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called invoke() of the object remoteRef. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.



Marshalling:

This is the process of converting a Java object into a format suitable for storage or transmission. This format can be a binary stream, a text-based format like JSON or XML, or another external data representation. The purpose of marshalling is to prepare the object's state so it can be sent over a network, saved to a file, or processed by a different system or language. This process is essentially a form of serialization.

Unmarshalling:

This is the reverse process of marshalling. It involves converting the external data representation (e.g., a JSON string, an XML document, or a binary stream) back into a Java object. Unmarshalling reconstructs the object in memory, allowing it to be used within the Java application. This process is essentially a form of deserialization.

Steps for RMI Application

1. Define the remote interface
2. Develop the implementation class (remote object)
3. Develop the server program
4. Develop the client program
5. Compile the application
6. Execute the application

Define the remote interface

A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface. To create a remote interface –

- Create an interface that extends the predefined interface Remote which belongs to the package.
- Declare all the business methods that can be invoked by the client in this interface.
- Since there is a chance of network issues during remote calls, an exception named RemoteException may occur; throw it.

Develop the implementation class (remote object)

The next step is to implement the remote interface. To implement the remote interface, the class should extend to UnicastRemoteObject class of java.rmi package. Also, a default constructor needs to be created to throw the java.rmi.RemoteException from its parent constructor in class.

Developing the Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the RMIregistry. To develop a server program –

- Create a client class from where you want invoke the remote object.
- Create a remote object by instantiating the implementation class as shown below.
- Export the remote object using the method exportObject() of named UnicastRemoteObject which belongs to the package java.rmi.server. the class
- Get the RMI registry using the getRegistry() method of the LocateRegistry class which belongs to the package java.rmi.registry.
- Bind the remote object created to the registry using the bind() method of the class named Registry. To this method, pass a string representing the bind name and the object exported, as parameters.

Developing the Client Program

Write a client program in it, fetch the remote object and invoke the required method using this object. To develop a client program –

- Create a client class from where your intended to invoke the remote object.
- Get the RMI registry using the getRegistry() method of the LocateRegistry class which belongs to the package java.rmi.registry.
- Fetch the object from the registry using the method lookup() of the class Registry which belongs to the package java.rmi.registry. To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.
- The lookup() returns an object of type remote, down cast it to get int.
- Finally invoke the required method using the obtained remote object.

Compiling the Application To compile the application

- Compile the Remote interface.
- Compile the implementation class.
- Compile the server program.
- Compile the client program.

Or

Open the folder where you have stored all the programs and compile all the Java files as shown below.

```
Javac *.java
```

Executing the Application :

Step 1 − Start the rmi registry using the following command.

```
start rmiregistry
```

Step 2 − Run the server class file.

```
Java Server
```

Step 3 − Run the client class file.

```
java Client
```

Program :

Calculator.java

```java
import java.rmi.*;

public interface Calculator extends Remote {
    int add(int a, int b) throws RemoteException;
    int subtract(int a, int b) throws RemoteException;
}
```

CalculatorImpl.java

```java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {

    protected CalculatorImpl() throws RemoteException {
        super();
    }

    public int add(int a, int b) throws RemoteException {
        return a + b;
    }

    public int subtract(int a, int b) throws RemoteException {
        return a - b;
    }
}
```

CalculatorServer.java

```java
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class CalculatorServer {
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1099); // Start RMI registry on port 1099
            CalculatorImpl obj = new CalculatorImpl();
            Naming.rebind("CalculatorService", obj);
            System.out.println("Calculator RMI Server is running...");
        } catch (Exception e) {
            System.out.println("Server Exception: " + e);
        }
    }
}
```

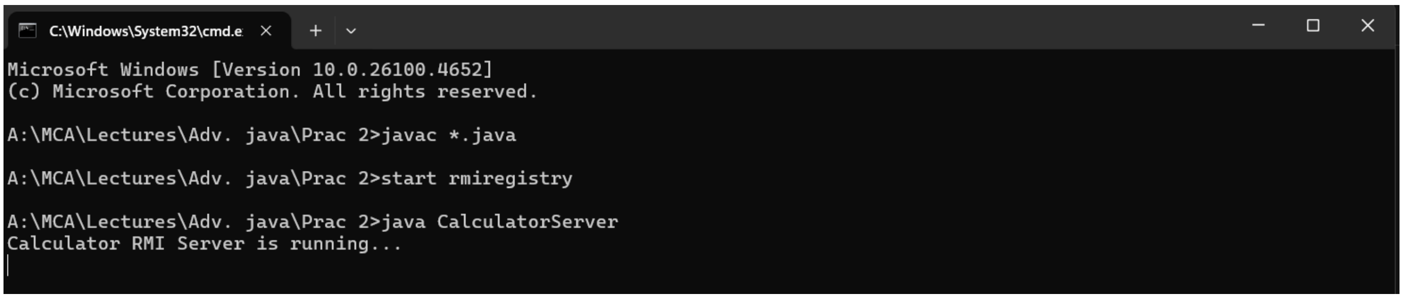
CalculatorClint.java

```java
import java.rmi.Naming;

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Calculator stub =
(Calculator)Naming.lookup("rmi://localhost:2000/CalculatorService");
            int result1 = stub.add(20, 10);
            int result2 = stub.subtract(20, 10);

            System.out.println("Addition: " + result1);
            System.out.println("Subtraction: " + result2);
        } catch (Exception e) {
            System.out.println("Client Exception: " + e);
        }
    }
}
```

Output :

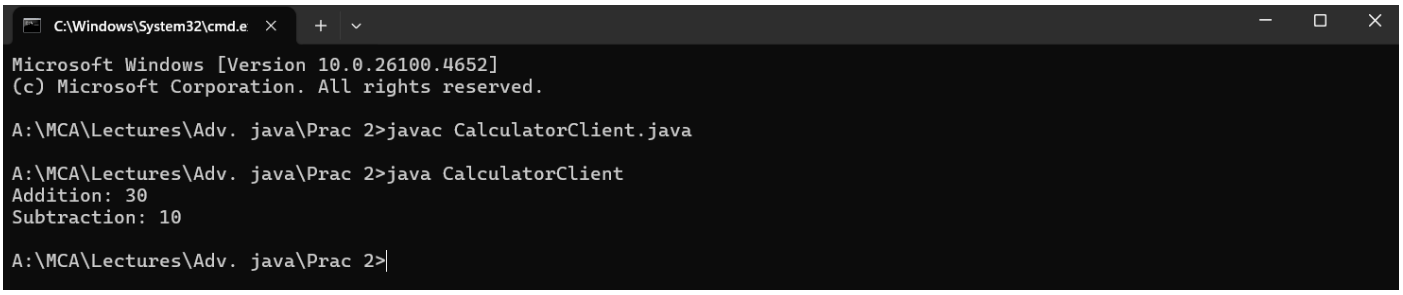


```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

A:\MCA\Lectures\Adv. java\Prac 2>javac *.java

A:\MCA\Lectures\Adv. java\Prac 2>start rmiregistry

A:\MCA\Lectures\Adv. java\Prac 2>java CalculatorServer
Calculator RMI Server is running...
```



```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

A:\MCA\Lectures\Adv. java\Prac 2>javac CalculatorClient.java

A:\MCA\Lectures\Adv. java\Prac 2>java CalculatorClient
Addition: 30
Subtraction: 10

A:\MCA\Lectures\Adv. java\Prac 2>
```