

<b>Name of Student:</b> Bhavesh Vasant Laku		
<b>Roll Number:</b> 28		<b>LAB Practical No:</b> 3
<b>Title of LAB Assignment:</b> Remote Method Invocation		
<b>DOP:</b> 29-08-2024		<b>DOS:</b> 05-09-2024
<b>CO Mapped:</b> CO2	<b>PO Mapped:</b> PO3, PO5, PO7, PO12, PSO1, PSO2	<b>Signature:</b>

## **Practical 3**

**Aim (1):** To find date and time using Remote Method Invocation (Use stubs and skeletons).

### **Theory:**

#### **1. Remote Method Invocation (RMI)**

**Remote Method Invocation (RMI)** is a Java API that allows objects to invoke methods on an object running in another Java Virtual Machine (JVM). The key advantage of RMI is that it makes distributed computing simpler by allowing a Java program to call methods on remote objects as if they were local, abstracting the complexities of the underlying network communication.

#### **Key Concepts in RMI:**

- **Remote Interface:** The interface that declares the methods that can be called remotely. This interface extends `java.rmi.Remote`.
- **Remote Object:** An object whose methods can be invoked remotely. It implements the remote interface and extends `java.rmi.server.UnicastRemoteObject`.
- **RMI Registry:** A simple naming service that allows clients to obtain a reference to a remote object. Remote objects are registered with the registry under a specific name.

#### **2. Stub and Skeleton**

In Java RMI, **Stub** and **Skeleton** are two important components used to facilitate remote communication between the client and server.

#### **Stub:**

- **Definition:** The stub acts as a proxy for the remote object in the client's JVM. It is a client-side representation of the remote object and is responsible for marshalling (packing) the method arguments, sending them to the server, and unmarshalling (unpacking) the results received from the server.
- **Process:**
  1. When a client calls a remote method, the call is actually made to the stub.
  2. The stub marshals the method parameters and sends the request to the skeleton on the server side over the network.

**Skeleton (Deprecated in Java 2):**

- **Definition:** The skeleton was used in earlier versions of Java (before Java 2) to unmarshall the method arguments, invoke the actual method on the remote object, and then marshal the return value back to the stub.
- **Process:**
  1. The skeleton receives the method call from the stub, unmarshals the arguments, and invokes the method on the actual remote object.
  2. The return value is marshalled by the skeleton and sent back to the stub.

However, starting with Java 2 (Java 1.2), the skeleton was made obsolete, and the functionality it provided was integrated into the runtime environment, making it unnecessary to generate skeletons separately.

**3. RMI Architecture**

The RMI architecture is based on a layered model, with each layer having specific responsibilities:

- **Application Layer:** Contains the actual implementation of the methods and the remote interface that defines the methods that can be invoked remotely.
- **Stub/Skeleton Layer:** Handles communication between the client and server, allowing method calls to be passed between them. As mentioned earlier, skeletons are no longer used in Java 2 and later.
- **Remote Reference Layer:** Responsible for managing references to remote objects and handling the protocol-specific details.
- **Transport Layer:** Manages the actual network communication between the client and server, usually using TCP/IP.

**4. RMI Process Flow:**

1. **Server Side:**
  - A remote object is created and its reference is exported to the RMI runtime.
  - The remote object is registered with the RMI registry using a unique name.
2. **Client Side:**
  - The client obtains the remote object reference from the RMI registry by looking it up using the unique name.
  - The client invokes methods on the remote object via the stub, which communicates with the remote object on the server side.

**Code:****1) DateTimeService.java**

```
package com.example.rmi;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;
public interface DateTimeService extends Remote {
    Date getDateTime() throws RemoteException;
}
```

**2) DateTimeServiceImpl**

```
package com.example.rmi;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.util.Date;
public class DateTimeServiceImpl extends UnicastRemoteObject implements DateTimeService {
    protected DateTimeServiceImpl() throws RemoteException {
        super();
    }
    @Override
    public Date getDateTime() throws RemoteException {
        return new Date();
    }
}
```

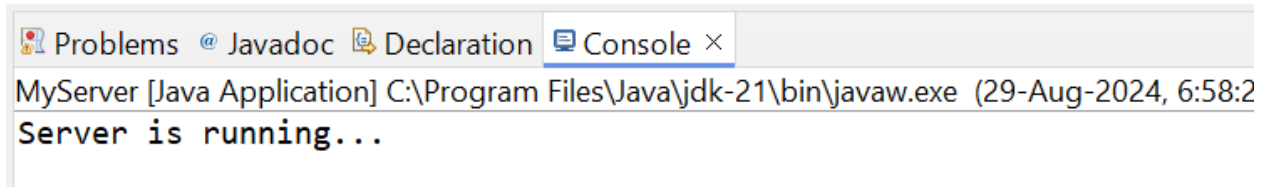
**3) MyServer.java**

```
package com.example.rmi;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class MyServer {
    public static void main(String[] args) {
        try {
            DateTimeService dateTimeService = new DateTimeServiceImpl();
            Registry registry = LocateRegistry.createRegistry(2000);
            registry.rebind("DateTimeService", dateTimeService);
            System.out.println("Server is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

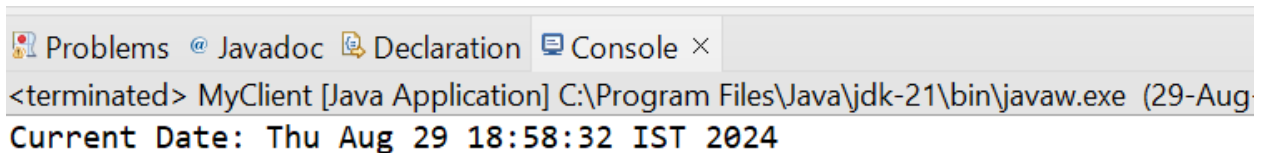
#### 4) MyClient.java

```
package com.example.rmi;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Date;
public class MyClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost", 2000);
            DateTimeService dateTimeService = (DateTimeService)
registry.lookup("DateTimeService");
            Date dateTime = dateTimeService.getDateTime();
            System.out.println("Current Date: " + dateTime);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

#### Output:

A screenshot of an IDE's console window. The title bar shows 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console text reads: 'MyServer [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (29-Aug-2024, 6:58:2 Server is running...'.

```
Problems @ Javadoc Declaration Console ×
MyServer [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (29-Aug-2024, 6:58:2
Server is running...
```

A screenshot of an IDE's console window. The title bar shows 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console text reads: '<terminated> MyClient [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (29-Aug-2024, 18:58:32 IST 2024'.

```
Problems @ Javadoc Declaration Console ×
<terminated> MyClient [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (29-Aug-
Current Date: Thu Aug 29 18:58:32 IST 2024
```

**Aim (2):** Implementation of equation solver using Remote Method Invocation (RMI).

**Code:**

**1) EquationSolverServer.java**

```
package com.example.rmi;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface EquationSolver extends Remote {
    int solveEquation(String equation, int a, int b, int c) throws RemoteException;
}
```

**2) EquationSolverImpl.java**

```
package com.example.rmi;

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

public class EquationSolverImpl extends UnicastRemoteObject implements
EquationSolver {

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

    @Override
    public int solveEquation(String equation, int a, int b, int c) throws RemoteException {
        int result = 0;
        switch (equation) {
            case "(a+b)^2":
                result = (a + b) * (a + b);
                break;
        }
    }
}
```

```
        case "(a-b)^2":
            result = (a - b) * (a - b);
            break;
        case "(a+b+c)^2":
            result = (a + b + c) * (a + b + c);
            break;
        case "(a+b)^3":
            result = (a + b) * (a + b) * (a + b);
            break;
        default:
            throw new IllegalArgumentException("Unknown equation: " + equation);
    }
    return result;
}
}
```

### 3) EquationSolverClient.java

```
package com.example.rmi;

import java.rmi.Naming;
import java.util.Scanner;

public class EquationSolverClient {

    public static void main(String[] args) {

        try {

            // Look up the remote equation solver object in the registry

            EquationSolver solver = (EquationSolver)
Naming.lookup("rmi://localhost/EquationSolver");
```

```
Scanner scanner = new Scanner(System.in);

while (true) {
    // Get the equation from the user
    System.out.print("Enter equation: ");
    String equation = scanner.nextLine();

    // Get the values of a, b, and c from the user
    System.out.print("Enter value for a: ");
    int a = scanner.nextInt();
    System.out.print("Enter value for b: ");
    int b = scanner.nextInt();
    int c = 0;
    if (equation.contains("c")) {
        System.out.print("Enter value for c: ");
        c = scanner.nextInt();
    }

    // Solve the equation
    int result = solver.solveEquation(equation, a, b, c);
    System.out.println("Answer: " + result);

    // Clear the newline character from the buffer
    scanner.nextLine();
}
} catch (Exception e) {
    e.printStackTrace();
}
```

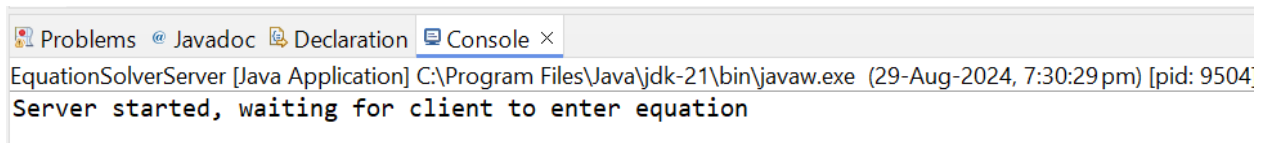


```
    }  
  }  
}
```

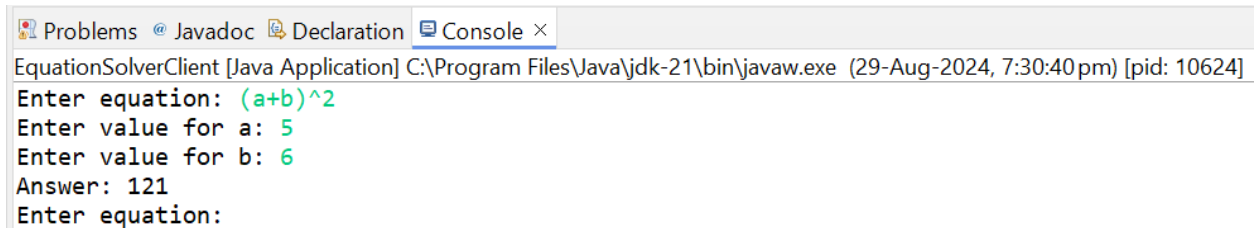
#### 4) EquationSolver.java

```
package com.example.rmi;  
import java.rmi.Remote;  
import java.rmi.RemoteException;  
public interface EquationSolver extends Remote {  
    int solveEquation(String equation, int a, int b, int c) throws RemoteException;  
}
```

#### Output:



EquationSolverServer [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (29-Aug-2024, 7:30:29 pm) [pid: 9504]  
Server started, waiting for client to enter equation



EquationSolverClient [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (29-Aug-2024, 7:30:40 pm) [pid: 10624]  
Enter equation: (a+b)^2  
Enter value for a: 5  
Enter value for b: 6  
Answer: 121  
Enter equation: