RMI in Component-Based Development

1. Introduction

RMI (Remote Method Invocation) is Java API, it is an integral part of component-based development, offering a means to interact with distributed components or objects seamlessly across different machines on a network.

2. RMI is an API

An API, or Application Programming Interface, serves as a set of rules, protocols, and tools that allows different software applications to communicate and interact with each other. the API facilitates communication between different software systems. It specifies how different software components should interact, making it easier for developers to use predefined functions to perform specific tasks without needing to understand the complexities of the underlying code or infrastructure.

APIs come in various forms:

1. **Web APIs:** used to enable remote communication between different web services or systems.
2. **Library APIs:** These are sets of functions and procedures exposed by libraries or frameworks to enable developers to use their functionalities within their own applications.
3. **Operating System APIs:** These provide interfaces for the operating system services. Developers can use these APIs to access system functionalities like file systems, hardware devices, memory management, etc.

3. RMI is a JAVA API

RMI allows objects in a Java Virtual Machine (JVM) to invoke remoted methods on objects residing in another JVM as if they were local, which may be running on different physical machines across a network. In other words: RMI enables developers to create distributed applications.

4. RMI architecture

The architecture of Java RMI involves several components and layers that work together to enable communication and interaction between distributed Java objects. In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client). Here's an overview of the key elements:

1. **Client:** The client initiates requests for remote method invocations. It interacts with remote objects as if they were local, invoking methods on these objects.
2. **Server:** The server hosts the remote objects and provides the services that clients request. It listens for incoming requests, processes them, and executes the methods on the remote objects.
3. **Stub and Skeleton:**
    - **Stub:** On the client-side, the stub acts as a local representative of the remote object. It receives method calls from the client and forwards them to the actual remote object on the server. The stub handles the serialization of method parameters and the transmission of requests to the server.
    - **Skeleton:** On the server-side, the skeleton receives incoming requests from the stub, deserializes the parameters, and delegates the method invocations to the actual implementation of the remote object.
4. **Remote Reference Layer:** This layer manages the creation, serialization, transmission, and management of references to remote objects. It handles the serialization and deserialization of object references to facilitate their transmission between JVMs.
5. **TCP/IP:** Java RMI uses the TCP/IP protocol suite as the underlying transport mechanism for communication between distributed Java objects. TCP/IP ensures reliable transmission of data packets over the network.
6. **JRMP (Java Remote Method Protocol):** JRMP is the default protocol used by Java RMI to enable remote method invocations between Java-based applications. It operates at a higher level than TCP/IP and is specifically designed for communication between Java objects.

7. **Registry:** The registry acts as a directory service that allows clients to look up and obtain references to remote objects hosted on the server. It maintains a mapping of names to remote object references.

The flow of communication involves the client making requests through the stub, which marshals the method parameters and sends them over the network using the underlying TCP/IP protocol. The server receives these requests through the skeleton, which then invokes the methods on the actual remote objects. The Remote Reference Layer manages the serialization and transmission of references, while the registry helps in locating remote objects.

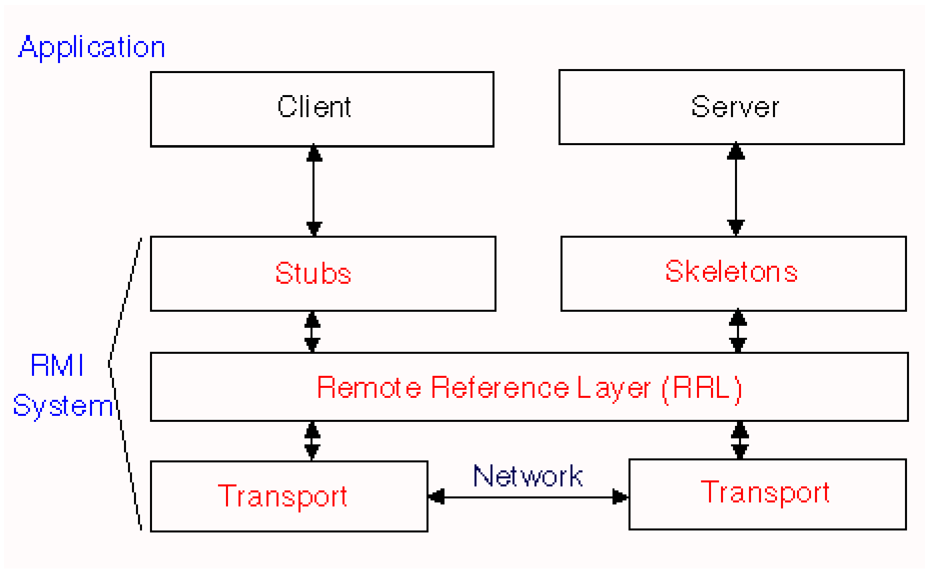


Fig.1: RMI architecture

LAB SESSION

Let's assume you have a remote service that provides a method to add two numbers, here is an

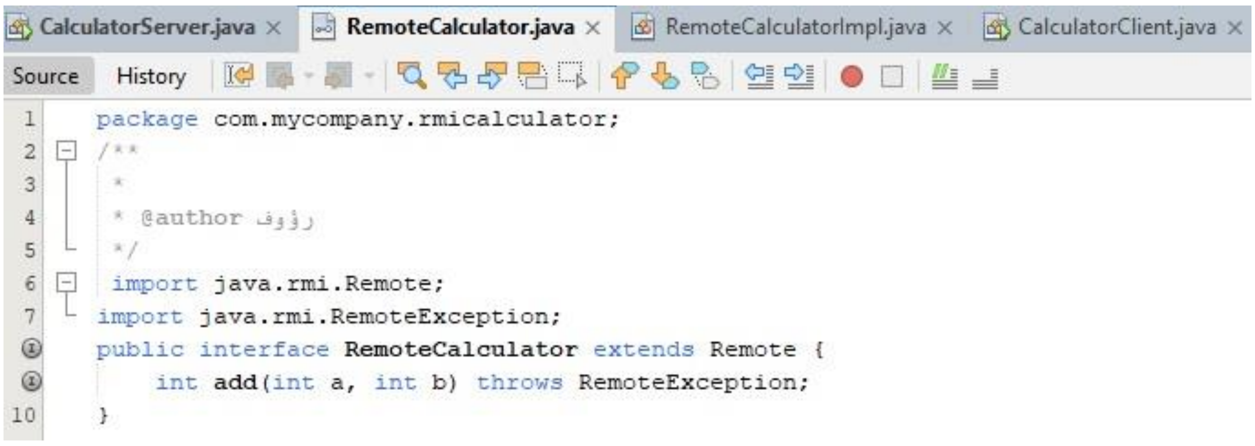
example using NetBeans to create an RMI project.

Step 1: Create a project in NetBeans

1. Open NetBeans and create a new Java project: **File -> New Project -> Java -> Java Application**.
2. Name the project (e.g., **RMICalculator**) and click "Next" to choose the project location.
3. In the project configuration step, make sure to check the "Create Main Class" box and name it, for example, **CalculatorServer**. This will create a class with a **main** method to start the RMI server.
4. Click "Finish" to complete creating the project.

Step 2: Define the remote interface

1. In the project's package, create a new interface named **RemoteCalculator**:
2. Right-click on the package -> New -> Java Interface

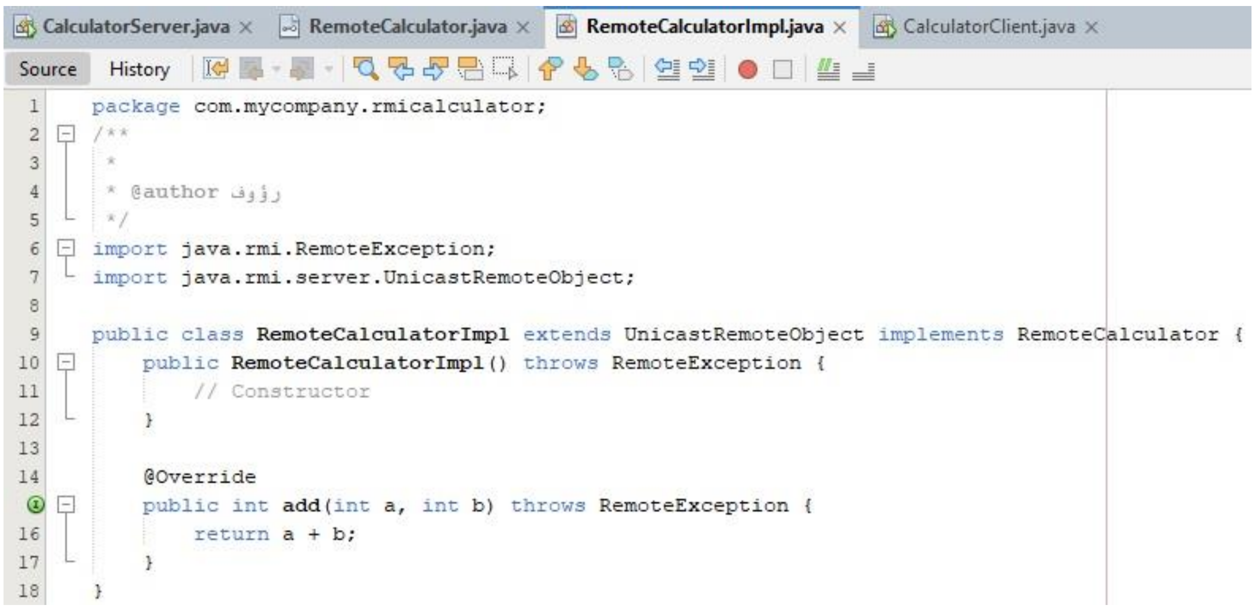


```java
package com.mycompany.rmicalculator;
/**
 *
 * @author رؤوف
 */
 import java.rmi.Remote;
import java.rmi.RemoteException;
public interface RemoteCalculator extends Remote {
    int add(int a, int b) throws RemoteException;
}
```

Step 3: Implement the remote interface

Create a class implementing the **RemoteCalculator** interface. Name it **RemoteCalculatorImpl**.

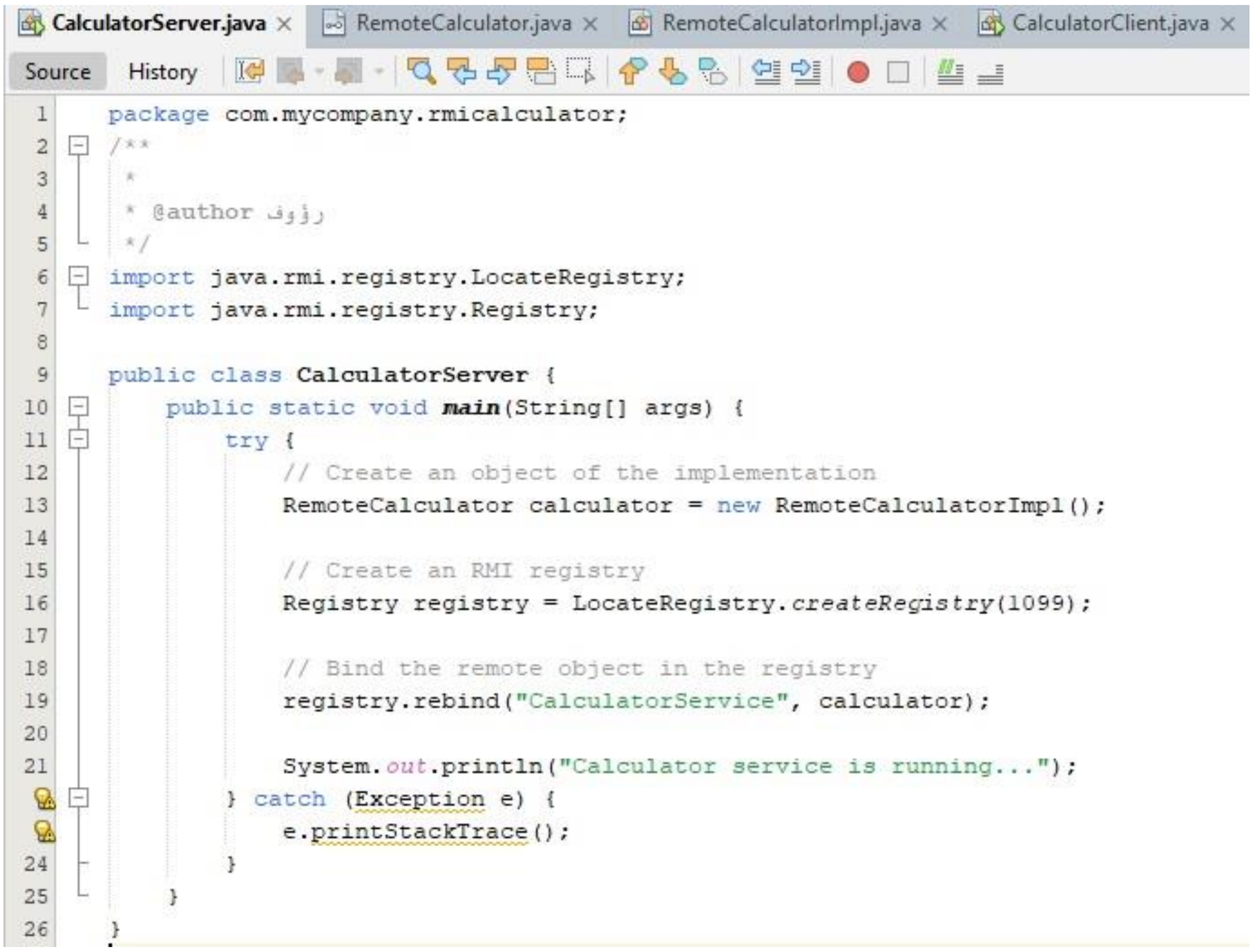
Right-click on the package -> New -> Java Class



```java
package com.mycompany.rmicalculator;
/**
 *
 * @author رؤوف
 */
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class RemoteCalculatorImpl extends UnicastRemoteObject implements RemoteCalculator {
    public RemoteCalculatorImpl() throws RemoteException {
        // Constructor
    }

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

Step 4: Create the server

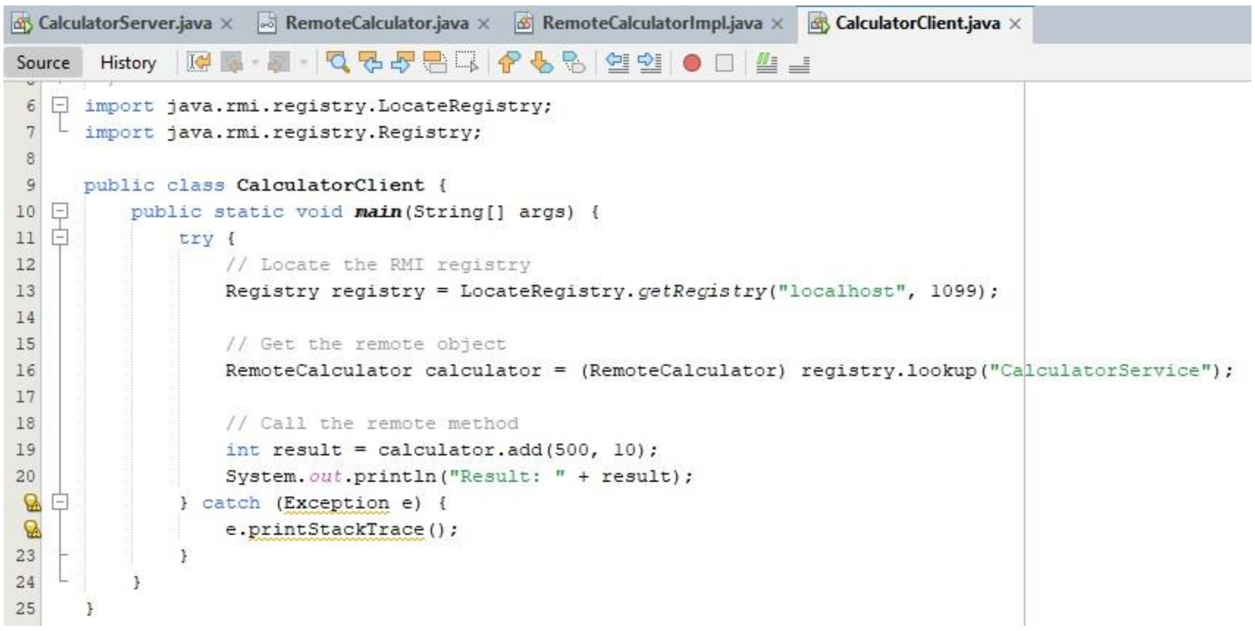
Modify the main class (**CalculatorServer**) to initialize the RMI server and register the remote object.



```java
package com.mycompany.rmicalculator;
/**
 *
 * @author رؤوف
 */
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class CalculatorServer {
    public static void main(String[] args) {
        try {
            // Create an object of the implementation
            RemoteCalculator calculator = new RemoteCalculatorImpl();

            // Create an RMI registry
            Registry registry = LocateRegistry.createRegistry(1099);

            // Bind the remote object in the registry
            registry.rebind("CalculatorService", calculator);

            System.out.println("Calculator service is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Step 5: Create the client

Create a new class for the RMI client (**CalculatorClient**) that will use the remote service.



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

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            // Locate the RMI registry
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);

            // Get the remote object
            RemoteCalculator calculator = (RemoteCalculator) registry.lookup("CalculatorService");

            // Call the remote method
            int result = calculator.add(500, 10);
            System.out.println("Result: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Execution:

1. Compile the files, Start the server by executing **CalculatorServer**.
2. Run the client by executing **CalculatorClient**.

This example demonstrates how RMI allows a client to call the **add()** method on a remote object (**RemoteCalculatorImpl**) using the **RemoteCalculator** interface.