Distributed Programming in Java

Session: 9

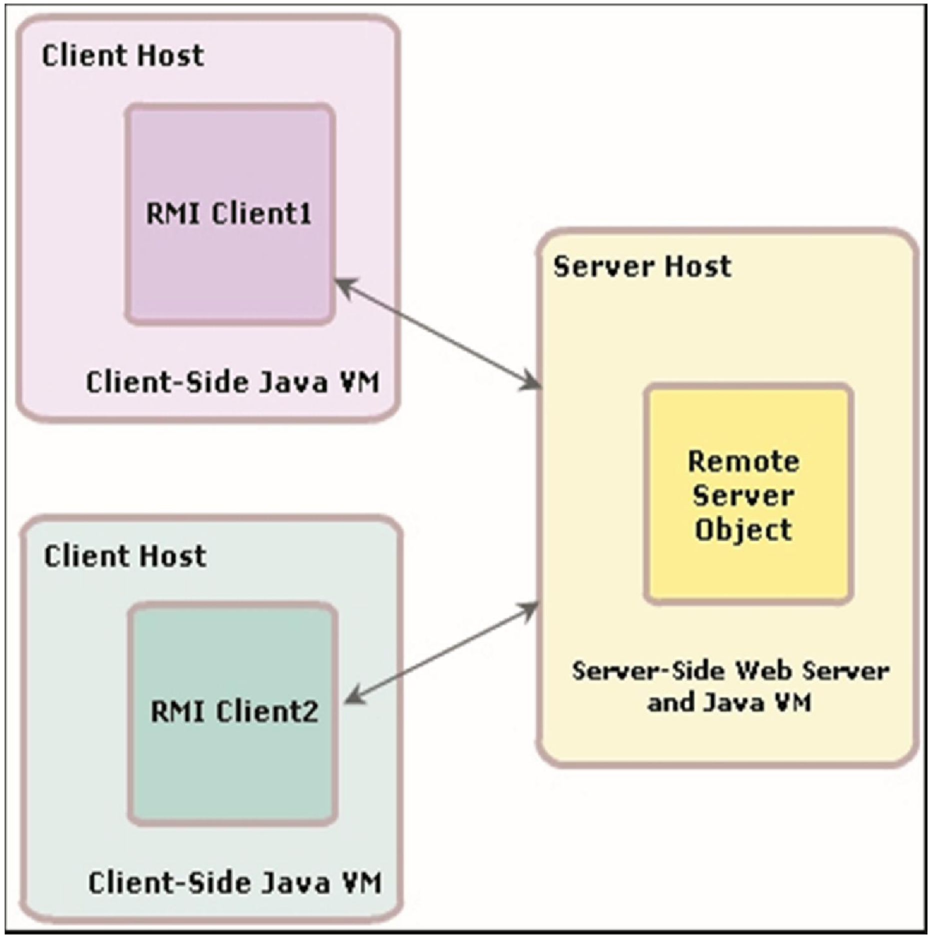
Remote Method Invocation

Objectives

- Describe RMI
- Explain RMI Architecture
- Explain the steps to implementing RMI

Distributed Computing

- Revolves around client-server technology where several client programs communicate with one or more server applications.
- Remote Method Invocation:
  - Allows Java applications to support the distributed computing architecture through the use of client-server technology.
  - Allows a Java program running inside a JVM to execute a method of a class available in another JVM.
- Figure displays the distributed computing.



RMI and its Need

Java provides the `java.rmi` package which contains all the necessary classes and interfaces to support distributed computing in Java.

Distributed computing applications developed using RMI are platform-independent.

The client and server applications can be executed on diverse machines and yet be made to communicate amongst themselves.

- **Need of RMI**
  - RMI allows to serialize and transmit objects.
  - RMI, unlike Socket Programming, takes into account the endian during method invocations.

RMI Architecture [1-3]

RMI architecture works by making objects in different JVMs look and act like they are local objects.

The transparency of RMI design does not allow applications to differentiate between local and remote objects. So any object is treated in the same way in any method invocation.

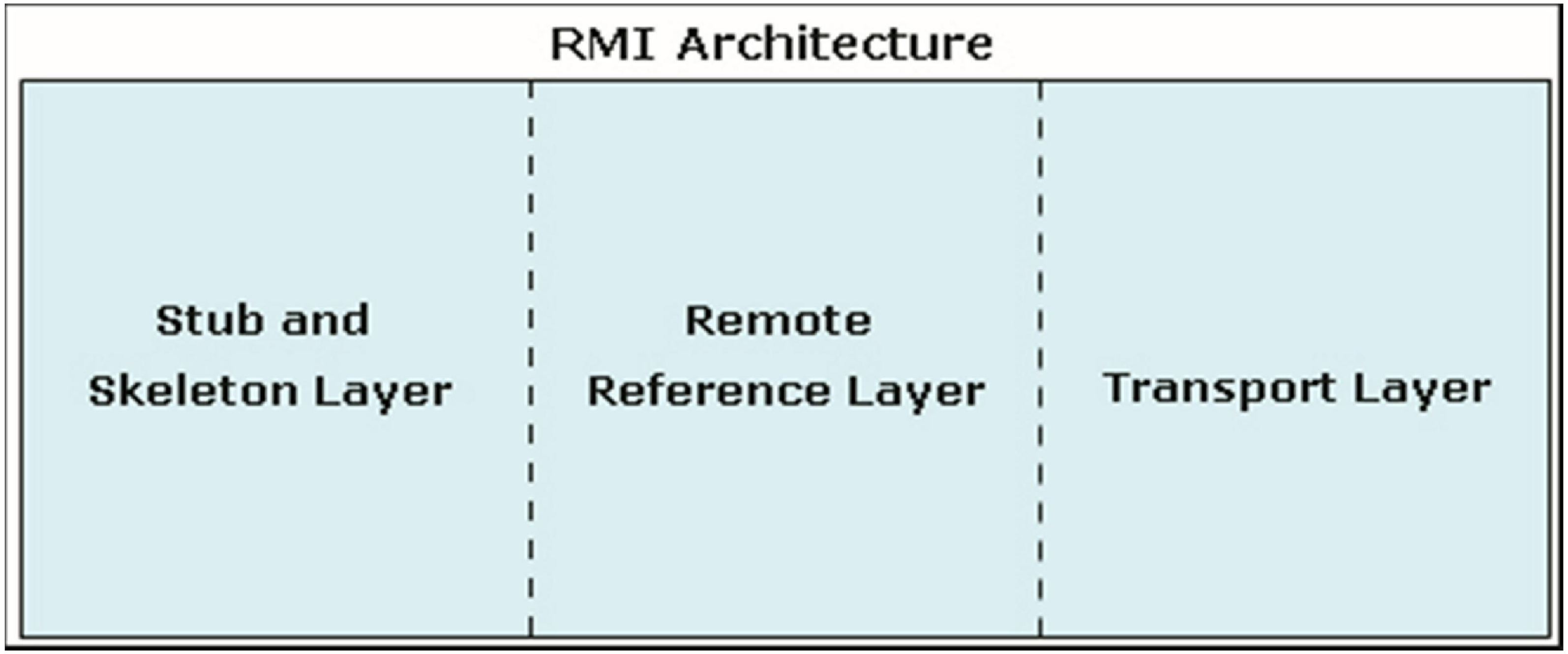
A JVM that calls remote object is called the client and the JVM that contains the remote object is called the server.

Obtaining the reference for a remote object is different that for a local object. But once this reference is obtained, all objects are treated equally.

A remote object is always accessed through its remote interface.

RMI Architecture [2-3]

- RMI is based on a three layered architecture.
- These three layers are just below the application layer on the server and client side.
- Figure shows the three layers of the RMI architecture.



RMI Architecture [3-3]

Stub and Skeleton Layer

- The Stub resides on the client side, and the Skeleton on the server side.
- Undertakes Marshalling and Unmarshalling of data.
- Transmits and receives data to and from the Remote Reference Layer.

Remote Reference Layer

- Is responsible for the invocation of the remote method.
- Receives the remote reference and marshaled arguments from the stub.
- Converts the client request into a low-level RMI transport request and forwards it to transport layer.

Transport Protocol Layer

- Is responsible for setting up connections using sockets.
- Listens for incoming calls and manages requests from remote reference layer.
- The transport layer of stub and skeleton communicates with the low-level RMI transport protocol.

RMI Architecture Layer

Remote Interface

- Methods which are meant to be remote should be defined in an interface which extends the `java.rmi.Remote` interface.
- The `java.rmi.Remote` is a marker interface and simply serves to distinguish interfaces which are non-remote.

Implementation Class

- The Implementation class implements the remote interface and provides the implementation of the remote methods.
- This implementation class should extend from the `java.rmi.server.UnicastRemoteObject`.

Stub

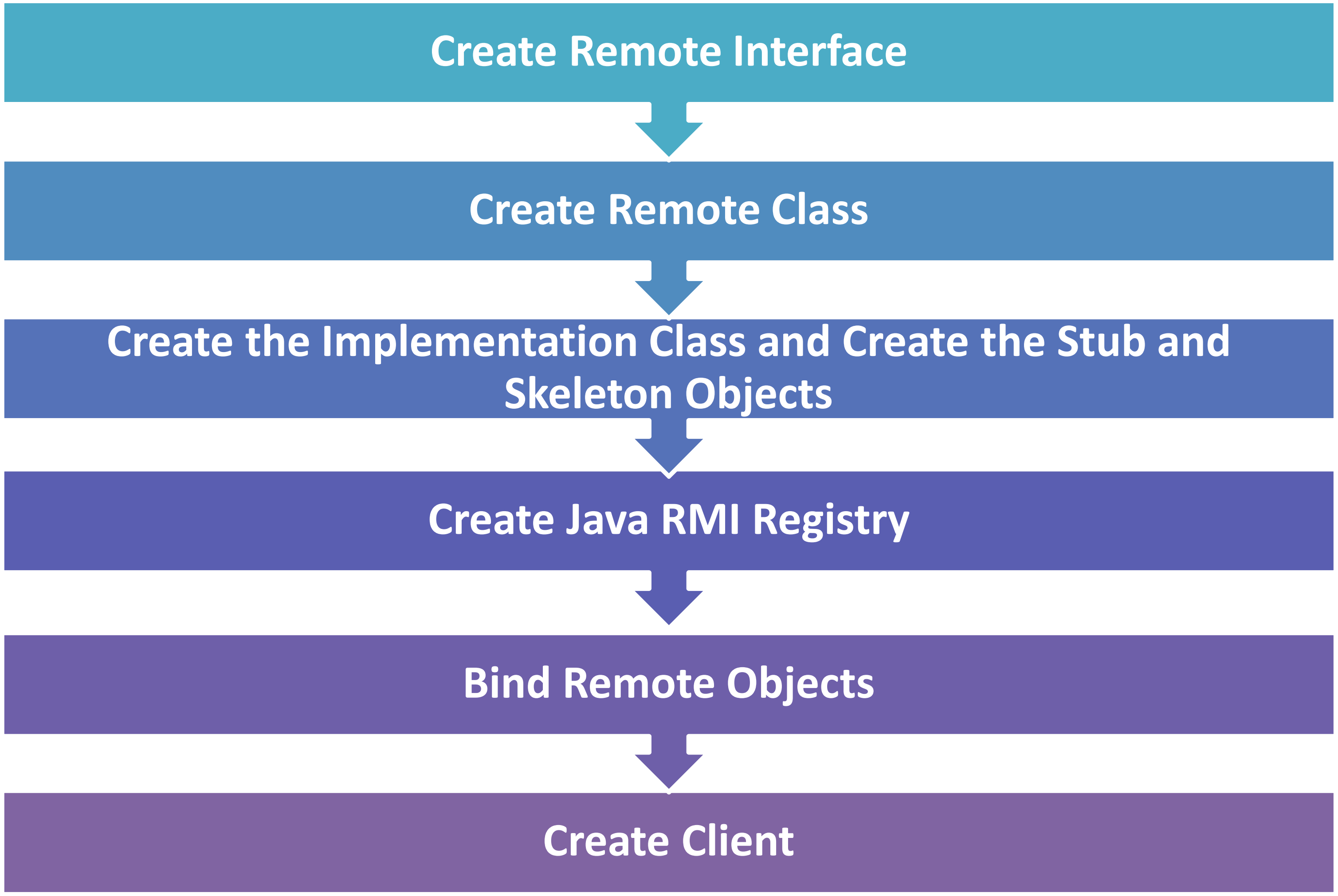
- The Stub resides on the client side; if it is not available, the JVM will download it to the client side.

Skeleton

- The Skeleton class constitutes one part of Stub and Skeleton Layer. The Skeleton resides on the server side.

Implementing RMI

- The process to create RMI program is to create server code and remote interface.
- To create and run an RMI application, the following steps need to be followed:



Creating Remote Interface [1-3]

- **Define an interface which extends `java.rmi.Remote`**
  - Marker interface indicates that your interface is a remote interface.

Syntax:

```
public interface Interface-Name extends java.rmi.Remote {}
```

  - Code Snippet shows how to define an interface `HelloInterface`.

Code Snippet

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloInterface extends Remote {
    . . .
}
```

- **Declare the remote methods signature**
  - All the methods which are meant to be remotely invoked are declared in the remote interface.
  - Remote clients can only invoke these methods.
  - Methods from other interfaces which are non-remote are not available for clients to invoke.

Creating Remote Interface [2-3]

Syntax:

```
public interface Interface-Name extends java.rmi.Remote {
   public Return-type methodName(Parameters if any)
}
```

- Code Snippet shows how to declare the remote method signature.

Code Snippet

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloInterface extends Remote {
 // Declares all the remote method signatures
    public String sayHello(String name);
    }
```

- **All these methods should be qualified to throw `java.rmi.RemoteException`**
  - All the remote methods declared should be qualified to throw `java.rmi.RemoteException`.
  - This is required to identify catastrophic events, such as network failure, server not ready, and so on.

Creating Remote Interface [3-3]

Syntax:

```
public interface Interface-Name extends java.rmi.Remote {
   public Return-type methodName(Parameters if any) throws
                                                RemoteException
}
```

- Code Snippet shows how to qualify the remote method to throw a `RemoteException`.

Code Snippet

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloInterface extends Remote
{
    // Qualifies method to throw RemoteException
    public String sayHello(String name) throws RemoteException;
}
```

Creating Remote Class [1-3]

- The remote class is the implementation class which provides the implementations of remote methods.
- **Define a class which extends `java.rmi.server.UnicastRemoteObject`**
  - The remote implementation class should extend from the UnicastRemoteObject.
  - This super class is required to export the remote object using the RMI transport mechanism.

  Syntax:

  ```
  public class HelloServerImpl extends UnicastRemoteObject { }
  ```

- **Implement the remote interface(s)**
  - The remote implementation class should implement all the remote interfaces.
  - Only the methods declared in the remote interfaces are exposed to the client for remote invocations.

  Syntax:

  ```
  public class ImplementationClassName extends
             UnicastRemoteObject implements RemoteInterface
  ```

Creating Remote Class [2-3]

- **Provide a constructor which throws Remote Exception**
  - The implementation class which extends the `UnicastRemoteObject` should have a constructor which throws a `RemoteException`.
  - This is required because the super class constructor attempts to create and export a new remote object using an anonymous port.

Syntax:

```
public class ImplementationClassName extends
 UnicastRemoteObject implements RemoteInterface
```

  - Code Snippet shows how to provide a constructor which throws exception.

Code Snippet

```
public class HelloServerImpl extends

UnicastRemoteObject implements HelloInterface {

public HelloServerImpl() throws RemoteException {

super(); }        }
```

Creating Remote Class [3-3]

- **Provide the implementation of all remote methods**
  - The implementation class implements the remote interfaces and should provide the method implementation of all method signatures from the remote interfaces.
  - **Syntax:**

```
 public Return-type methodName(Parameters if any) throws
RemoteException { }
```

  - Code Snippet shows the implementation of the remote method from the `HelloInterface`.

Code Snippet

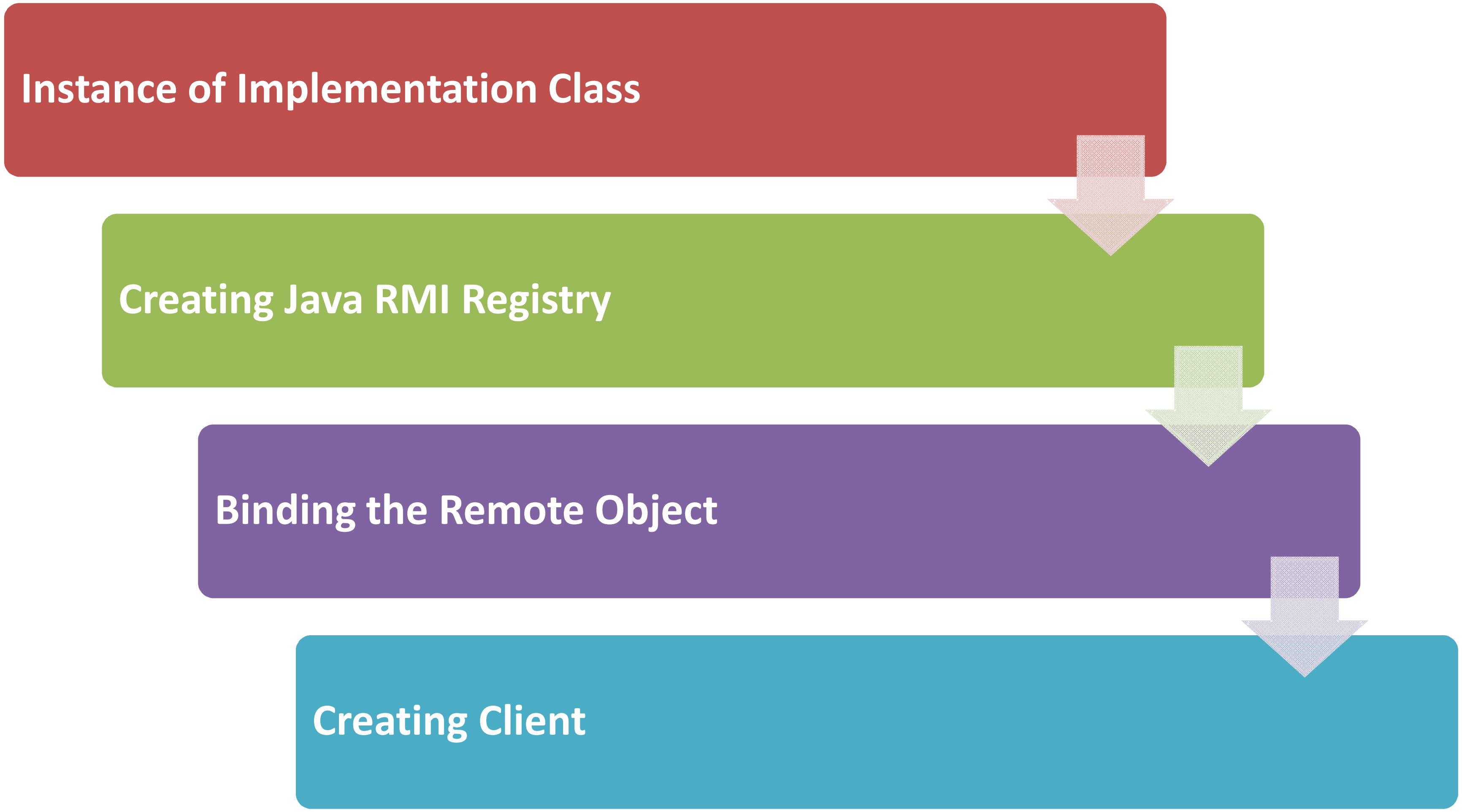
```
public class HelloServerImpl extends UnicastRemoteObject implements
HelloInterface {
public HelloServerImpl() throws RemoteException {
         super(); }
 public String sayHello(String name) throws RemoteException {
       return "Hello " + name;
 }
}
```

Remote Server [1-2]

- The Remote Server is any class which has a `main()` method to execute as an application.
- To make the remote methods available for clients for remote invocations, the server class has to perform a few setup procedures.
- After completion of these setup steps, the server is ready to allow clients to make remote invocations.

Remote Server [2-2]

- Figure shows the steps to create the instances of Implementation class and register it in the registry.


Instance of Implementation Class
Creating Java RMI Registry
Binding the Remote Object
Creating Client

Instance of Implementation Class

- The first step to set up the Remote Server is creating an instance of the Implementation class.
- The server class creates an instance of the implementation class so that the remote methods are available on the server side.
- These remote methods are not yet ready to be invoked remotely by clients.
- Example: `HelloServerImpl remoteObject = new HelloServerImpl();`

Creating Java RMI Registry

- The second step to set up the Remote Server is creating a Java RMI Registry programmatically.
- An RMI Registry is a naming service that can be started manually using the rmiregistry.exe application or created programmatically.
- The RMI Registry allows clients to get a reference (stub) to a remote object.
- The `createRegistry()` method of the `java.rmi.registry.LocalRegistry` class is used to create registry programmatically.

  Syntax:

  ```
  public static Registry createRegistry(int port)
  throws RemoteException
  ```
- Example: `Registry registry = LocateRegistry.createRegistry(1099);`

Binding the Remote Object

- The third step to set up the Remote Server is binding the Remote object with a name in the RMI Registry.
- The server application has to bind the remote object with a name in the registry.
- The clients have to lookup for this name to retrieve the stub which is the remote object reference.
- The `java.rmi.Naming` class has a method `rebind()` to bind the name to the registry.

Syntax:

```
 void rebind(String name, Remote obj) throws
RemoteException, AccessException
```

- Code Snippet shows how to specify a name in a URL and bind the remote object in the RMI registry.

Code Snippet

```
String name = "rmi://localhost:1099/HelloServer";
// Creates an instance if the implementation class
   HelloServerImpl remoteObject = new HelloServerImpl();
// Binds the remote object with a name in the registry
   Naming.rebind(name, remoteObject);
```

Creating the Client [1-2]

- A RMI Client is any application which needs to invoke the remote methods of a RMI Server.
- To be able to invoke the methods remotely, the RMI client has to perform the following steps:
  - **Lookup in the RMIRegistry for the name:**
    The RMI client requires the stub which is the remote object reference to invoke the methods remotely. The RMI client uses the `lookup()` method of `java.rmi.Naming` class to look for the name bound in the registry.

    Syntax:

    ```
    RemoteInterface remoteObject = (RemoteInterface)
    Naming.lookup(String name);
    ```
  - Code Snippet shows how to look up for the name bound in the registry.

Code Snippet

```
// Lookup for the name HelloServer in the rmi registry HelloInterface
objReference = (HelloInterface)
Naming.lookup("rmi://localhost:1099/HelloServer");
```

Creating the Client [2-2]

- **Use the Remote object reference to invoke methods**:

  If the lookup succeeds, the client acquires a stub. The `lookup()` method returns a `java.rmi.Remote` object; hence, it is required to cast it with the appropriate remote interface. The remote object reference can now be used to invoke the remote methods.

  Syntax:

  ```
  Return-type variable = remoteObject.methodName(parameters);
  ```

- Code Snippet shows how to invoke the remote method `sayHello()` with the remote object reference.

Code Snippet

```
// Invoke the remote method, and assign the result to a string

//strResult

    String strResult = objReference.sayHello("Tom");

// Display the result

    System.out.println(strResult);
```

Executing RMI Applications

- An RMI application can be executed in two different ways:
  - **Starting the RMI Registry manually**
    - In this method, the RMI registry has to be executed manually.
  - **Creating the RMI Registry programmatically**
    - In this method, the RMI server starts the RMI Registry programmatically.

list(String name) Method [1-2]

- The `list()` method is used to retrieve all the names bound to a RMI-registry. The names are in URL format.
- The URL format has the form `//host:port/name`. Only the names available in the registry at the point of invocation of this method are returned.

  Syntax:

  ```
  public static String[] list(String name) throws
  RemoteException, MalformedURLException
  ```
- Code Snippet shows how to retrieve all the names bound in the RMI registry.

Code Snippet

```
String[] boundedNames;
String name = "rmi://localhost:1099/HelloServer";
try {
// Create an instance of the implementation class
    HelloServerImpl remoteObject = new HelloServerImpl();
// Rebind the remote object with a new name in the registry
    Naming.rebind(name, remoteObject);
   . . .
  }
```

list(String name) Method [2-2]

```
// Retrieve all the names bound in the registry
    boundedNames =   Naming.list(name);
// Display all the names bound in the registry
    for (int i = 0; i< boundedNames.length; ++i) {
      System.out.println(boundedNames[i])
    }
  } catch (RemoteException ex) { System.out.println("Exception : " +
                                   ex.getMessage());
  } catch (MalformedURLException ex) { System.out.println("Exception : " +
                                          ex.getMessage());
```

unbind() Method [1-2]

- The `unbind()` method destroys the binding for the specified name which is associated with the remote object in the registry.

  Syntax:

  ```
  public static void unbind(String name) throws
  RemoteException, NotBoundException, MalformedURLException
  ```

- Code Snippet shows how to unbind the name associated with a remote object.

Code Snippet

```
String name = "rmi://localhost:1099/HelloServer";
try {
// Create an instance of the implementation class
    HelloServerImpl remoteObject = new HelloServerImpl();
// Bind the remote object with a name in the registry
     Naming.rebind(name, remoteObject);
. . .
. . .
```

unbind() Method [2-2]

```
// Unbind the name bound in the registry
    Naming.unbind(name);
    System.out.println(name + “ has been successfully unbound”);
} catch (RemoteException ex) { System.out.println(“Exception : “ +
                                 ex.getMessage());
} catch (NotBoundException ex) { System.out.println(“Exception : “ +
                                ex.getMessage());
} catch (MalformedURLException ex) { System.out.println(“Exception :
                             “ + ex.getMessage());
}
```

getRegistry() Method [1-2]

- The `getRegistry()` method is used to retrieve a reference of the remote object registry on the specified host name and the specified port.

  Syntax:

  ```
   public static Registry getRegistry(String host)
  throws RemoteException
  ```

- Code Snippet shows how to retrieve a reference of the remote object registry.

  Code Snippet

```
Registry registry;
String name = "rmi://localhost:1099";
try {
// Create an instance of the implementation class
   HelloServerImpl remoteObject = new HelloServerImpl();
// Retrieve the registry
   registry = LocateRegistry.getRegistry(name);
```

getRegistry() Method [2-2]

```
// Bind the remote object with a name in the registry
registry.rebind("HelloServer", remoteObject);
System.out.println("Remote object bound successfully");
} catch (RemoteException ex) {
System.out.println("Exception : " + ex.getMessage());
} catch (MalformedURLException ex) {
System.out.println("Exception : " + ex.getMessage());
}
```

- Similarly the method, `public static Registry getRegistry(String host, int port) throws RemoteException` retrieve a reference of the remote object registry on the specified host name and the specified port.

exportObject() Method [1-2]

- The `exportObject()` method is used to export the remote object and make it available to receive incoming RMI calls using an anonymous port.
- This method is used if the implementation class does not extend the `UnicastRemoteObject` class.

Syntax:

```
public static RemoteStub exportObject(Remote obj)
throws RemoteException
```

- Code Snippet shows how to export a remote object.

Code Snippet

```
Registry registry;
String name = "rmi://localhost:1099";
try {
// Create an instance of the implementation class
// This class does not extend the UnicastRemoteObject class
// The user should explicitly export the remote object
    HelloServerImpl_ remoteObject = new HelloServerImpl_();
```

exportObject() Method [2-2]

```
// Retrieve the registry
registry = LocateRegistry.getRegistry(name);
// Export the remote object
UnicastRemoteObject.exportObject(remoteObject);
System.out.println("Remote object successfully exported.");
} catch (RemoteException ex) { System.out.println("Exception : " +
                                ex.getMessage());
} catch (NoSuchObjectException ex) { System.out.println("Exception : " +
                               ex.getMessage());
}
```

- Similarly, the `unexportObject()` method is used to remove the remote object from the runtime.

  Syntax:

  ```
  public static boolean unexportObject(Remote obj,
  boolean force) throws NoSuchObjectException
  ```

Summary

- RMI allows a Java program running inside a JVM to execute a method of a class available in another JVM.
- Remote Method Invocation allows Java applications to support the distributed computing architecture through the use of client-server technology.
- The Remote Method Invocation (RMI) is based on a three layered architecture. The three layers of the RMI architecture are Stub and Skeleton Layer, Remote Reference Layer, and Transport Layer.
- To implement RMI, you should first identify all the methods which are meant to be remotely invoked by remote clients so that you can create a remote interface.
- The remote class is the implementation class which implements the remote interface.
- The getRegistry() method is used to retrieve a reference of the remote object registry on the specified host name and the default registry port 1099.