Unit 9
Remote Method Invocation(RMI)

RMI - Introduction

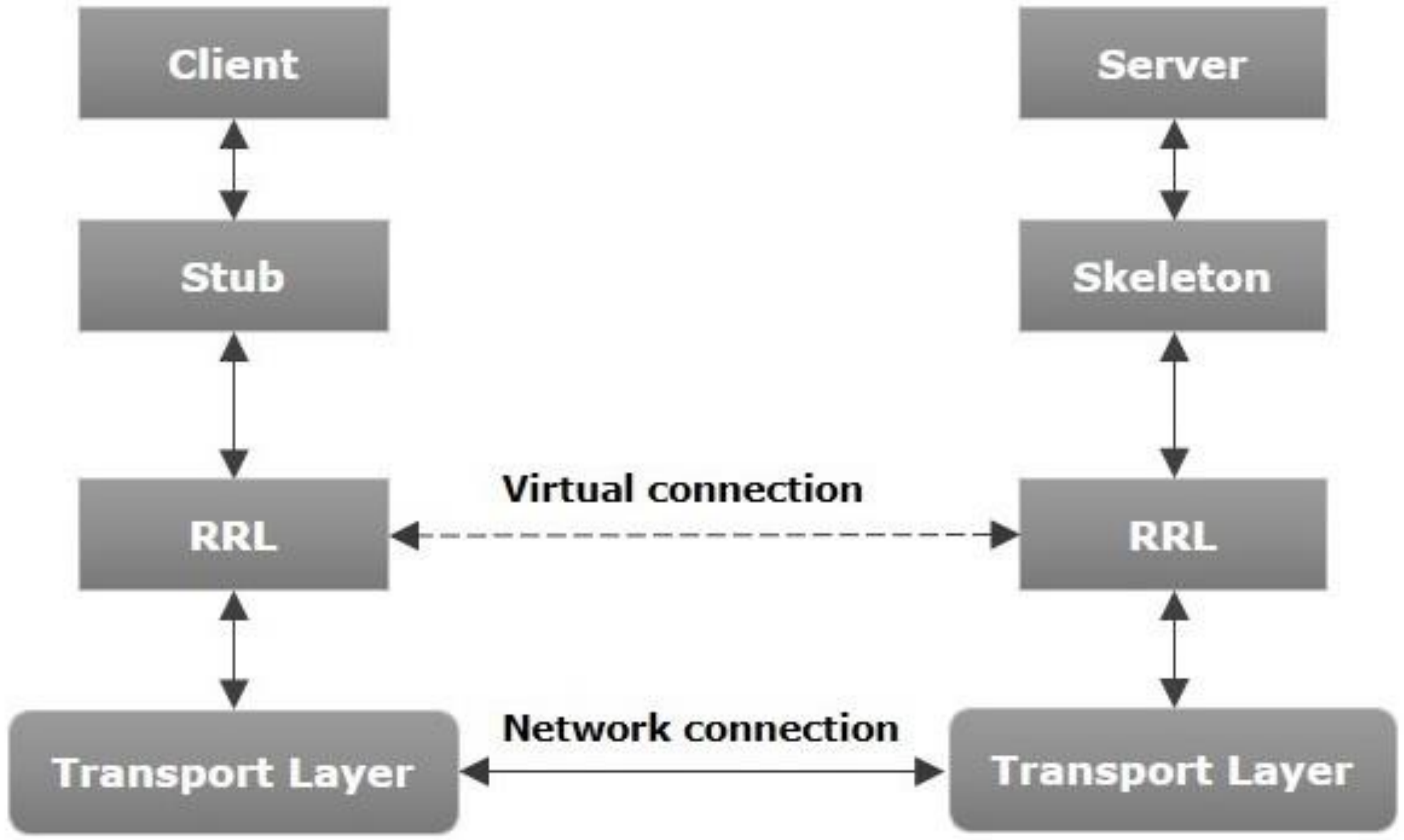
- It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.
- RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

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.

RMI - Architecture

- The following diagram shows the architecture of an RMI application.



RMI - Architecture

Description of components of this architecture.

- **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

RMI Registry

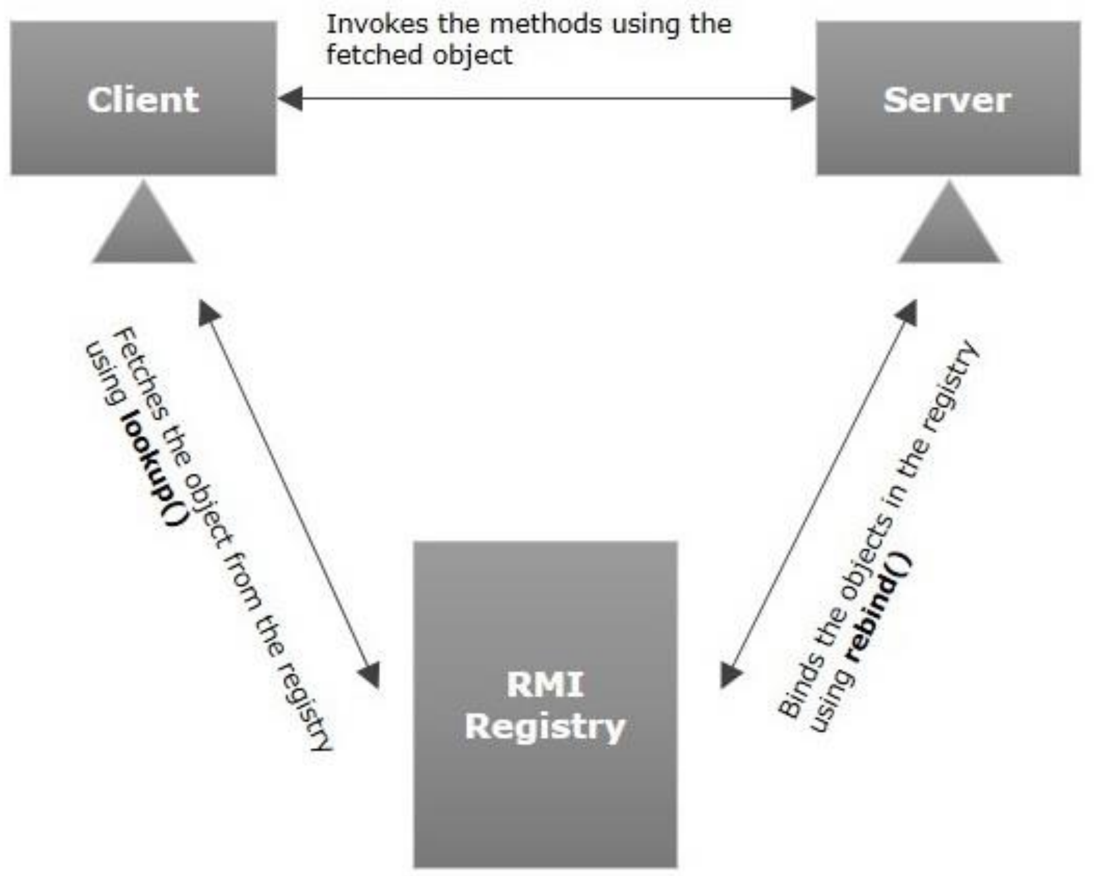
- RMI registry is a namespace on which all server objects are placed.
- Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.
- To invoke a remote object, the client

needs a reference of that object.

At that time, the client fetches the

object from the registry using its bind name

(using lookup() method).


Invokes the methods using the fetched object
Client
Server
Fetches the object from the registry using lookup()
Binds the objects in the registry using rebind()
RMI Registry

Goals of RMI

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

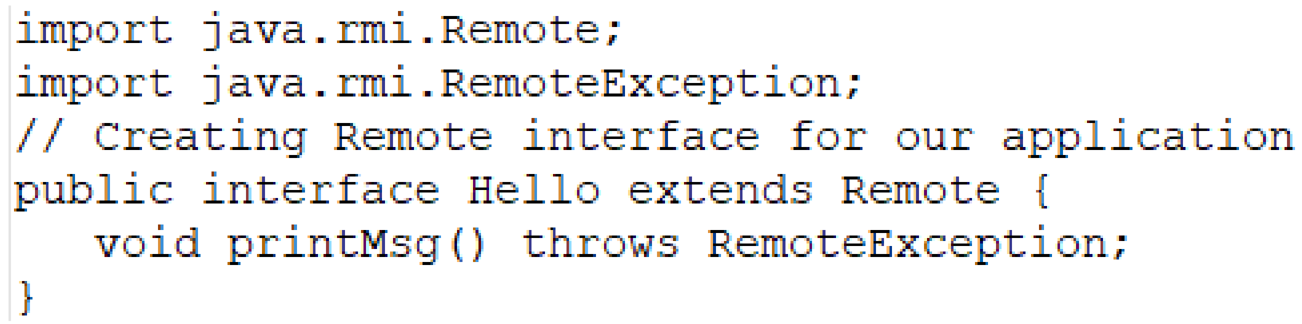
To write an RMI Java application, you would have to follow the steps given below –

- Define the remote interface
- Develop the implementation class
- Develop the client program
- Compile the application
- Execute the application remote object
- Develop the server program

Creating an RMI Server and Client

Defining 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.
- Following is an example of a remote interface. Here we have defined an interface with the name Hello and it has a method called printMsg().



```
import java.rmi.Remote;
import java.rmi.RemoteException;
// Creating Remote interface for our application
public interface Hello extends Remote {
   void printMsg() throws RemoteException;
}
```

Creating an RMI Server and Client

Developing the Implementation Class (Remote Object)

- We need to implement the remote interface created in the earlier step. (We can write an implementation class separately or we can directly make the server program implement this interface.) To develop an implementation class –
  - Implement the interface created in the previous step.
  - Provide implementation to all the abstract methods of the remote interface.
- Following is an implementation class. Here, we have created a class named ImplExample and implemented the interface Hello created in the previous step and provided body for this method which prints a message.

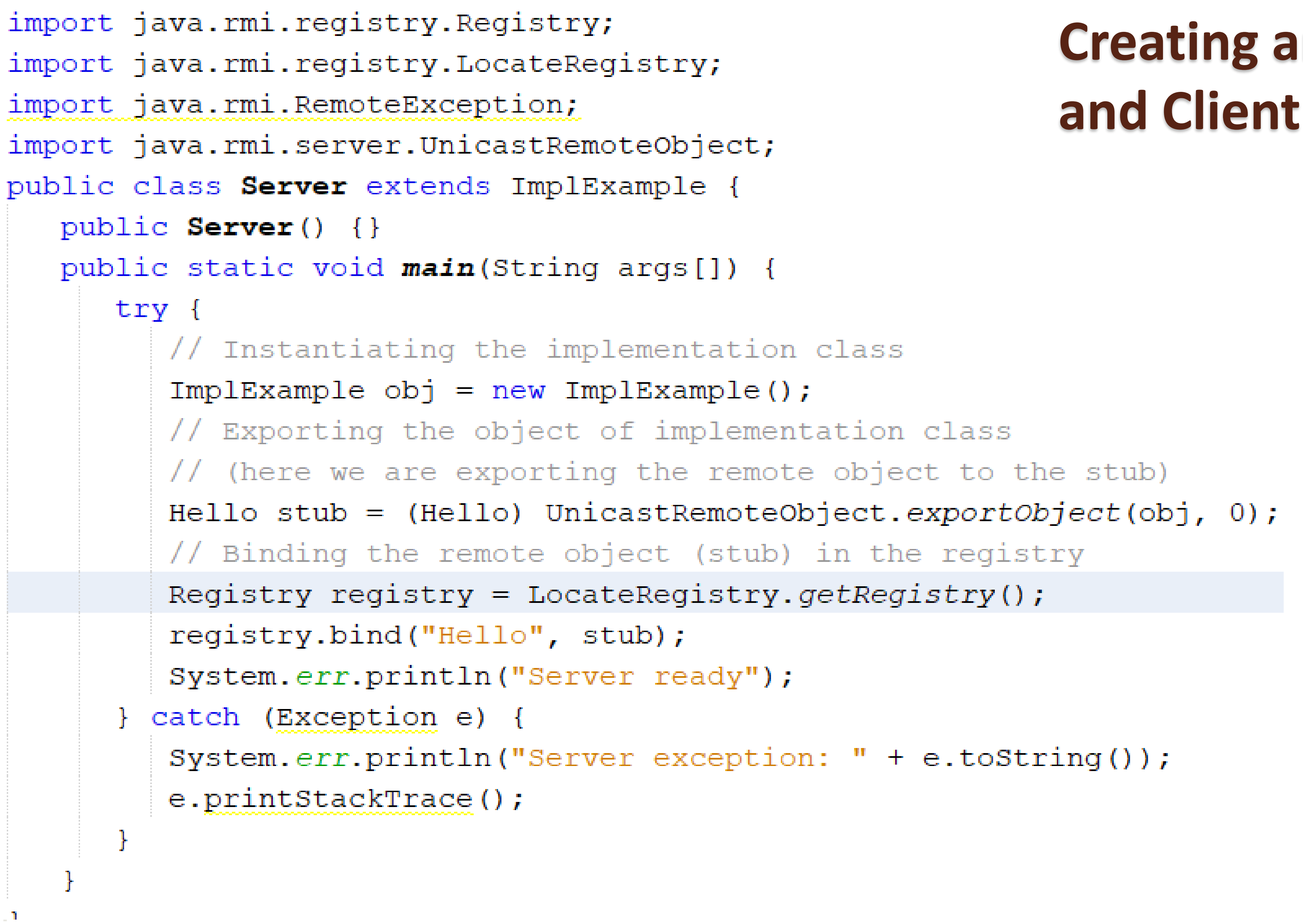
```
// Implementing the remote interface
public class ImplExample implements Hello {
   // Implementing the interface method
   public void printMsg() {
      System.out.println("This is an example RMI program");
   }
}
```

Creating an RMI Server and Client

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 the class named UnicastRemoteObject which belongs to the package java.rmi.server.
  - 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.

Creating an RMI Server and Client



```java
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Server extends ImplExample {
   public Server() {}
   public static void main(String args[]) {
      try {
         // Instantiating the implementation class
         ImplExample obj = new ImplExample();
         // Exporting the object of implementation class
         // (here we are exporting the remote object to the stub)
         Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
         // Binding the remote object (stub) in the registry
         Registry registry = LocateRegistry.getRegistry();
         registry.bind("Hello", stub);
         System.err.println("Server ready");
      } catch (Exception e) {
         System.err.println("Server exception: " + e.toString());
         e.printStackTrace();
      }
   }
}
```

Creating an RMI Server and Client

Developing the Client Program

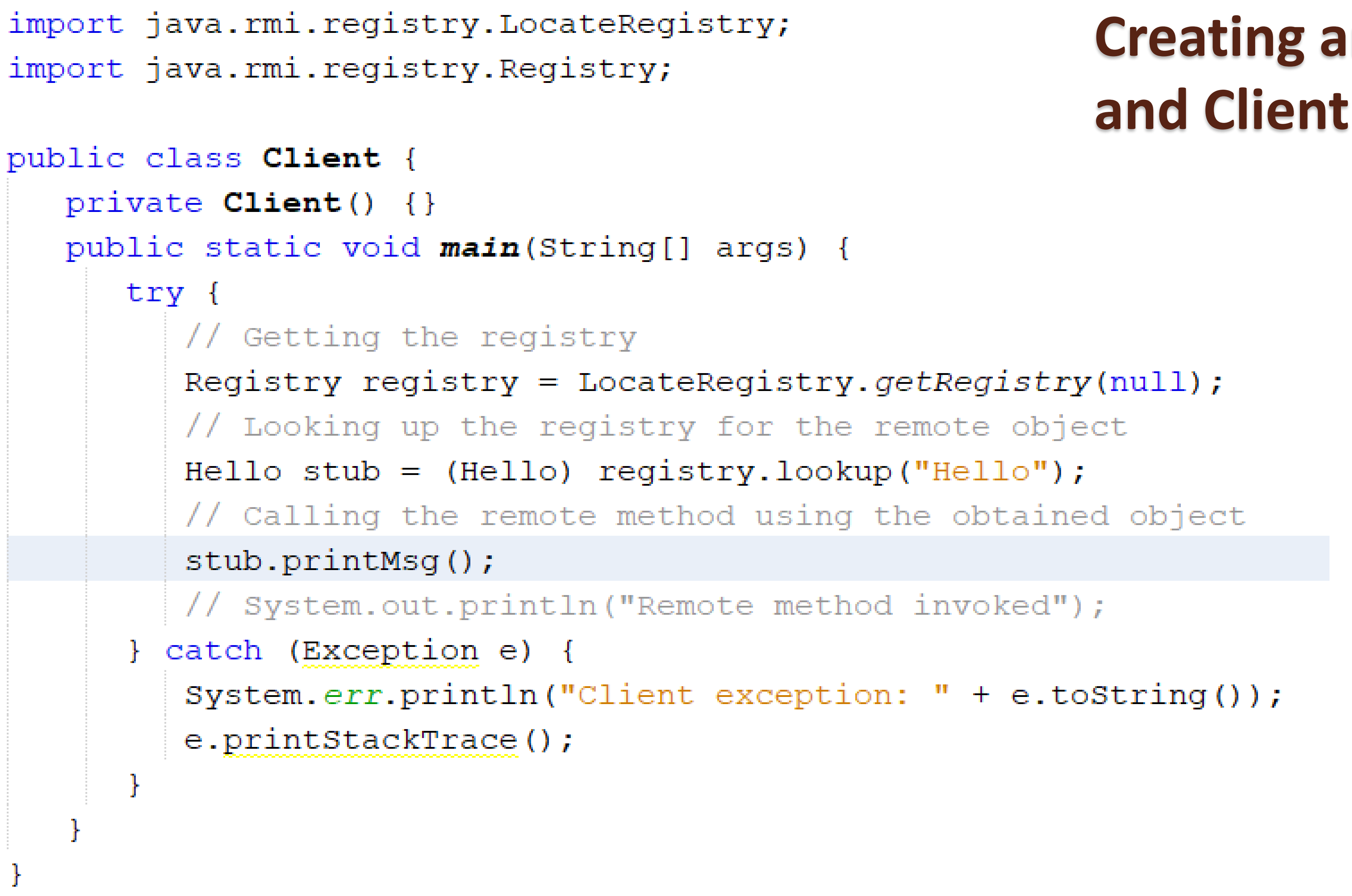
- Write a client program in it, fetch the remote object and invoke the required method using this object. To develop a client program –
- Create a client class from where your intended to invoke the remote object.
- Get the RMI registry using the getRegistry() method of the LocateRegistry class which 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 the type Hello.

Finally invoke the required method using the obtained remote object.

Creating an RMI Server and Client



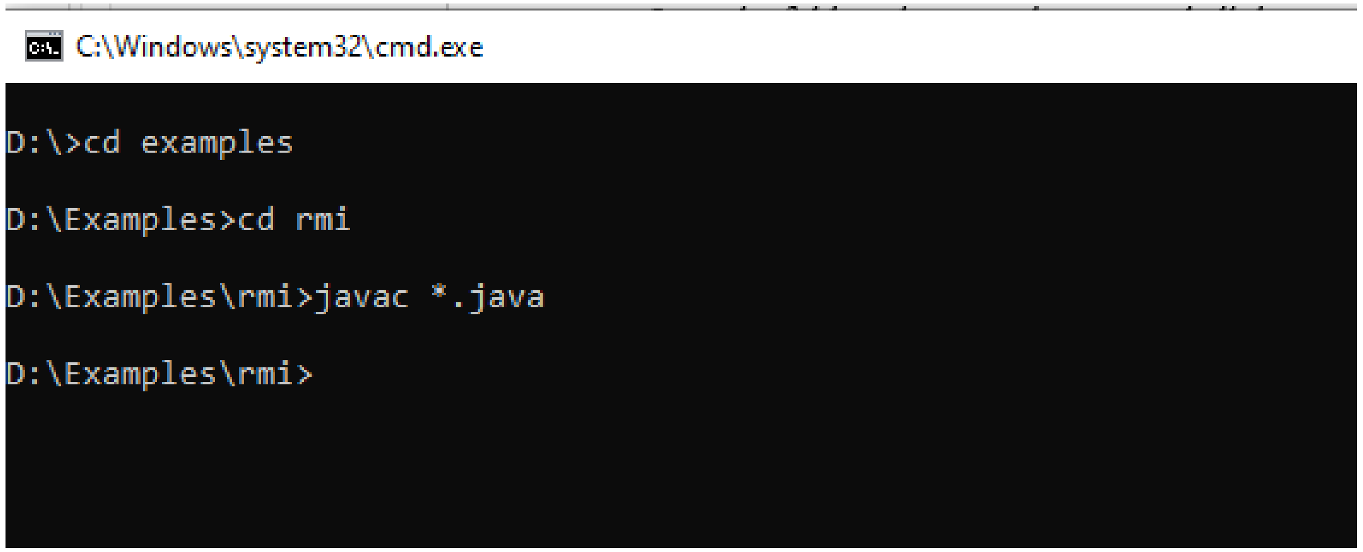
```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
   private Client() {}
   public static void main(String[] args) {
      try {
         // Getting the registry
         Registry registry = LocateRegistry.getRegistry(null);
         // Looking up the registry for the remote object
         Hello stub = (Hello) registry.lookup("Hello");
         // Calling the remote method using the obtained object
         stub.printMsg();
         // System.out.println("Remote method invoked");
      } catch (Exception e) {
         System.err.println("Client exception: " + e.toString());
         e.printStackTrace();
      }
   }
}
```

Running the RMI System

Compiling the Application

To compile the application –

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



C:\Windows\system32\cmd.exe

```
D:\>cd examples

D:\Examples>cd rmi

D:\Examples\rmi>javac *.java

D:\Examples\rmi>
```

Or,

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

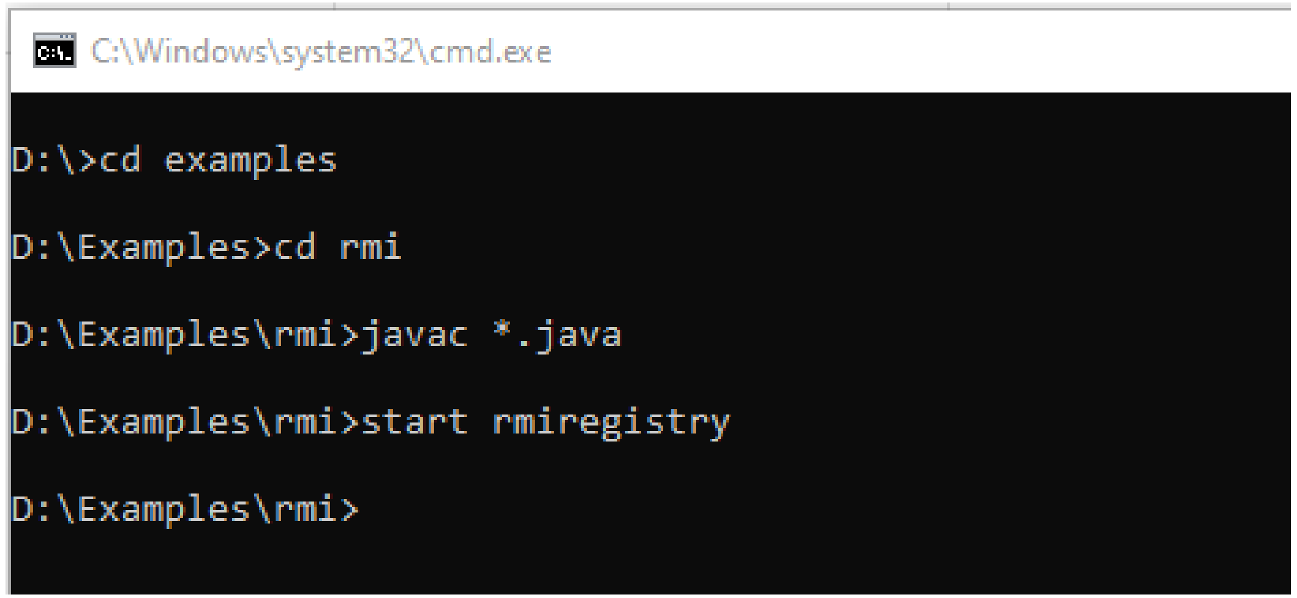
```
javac *.java
```

Running the RMI System

Executing the Application

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

start rmiregistry



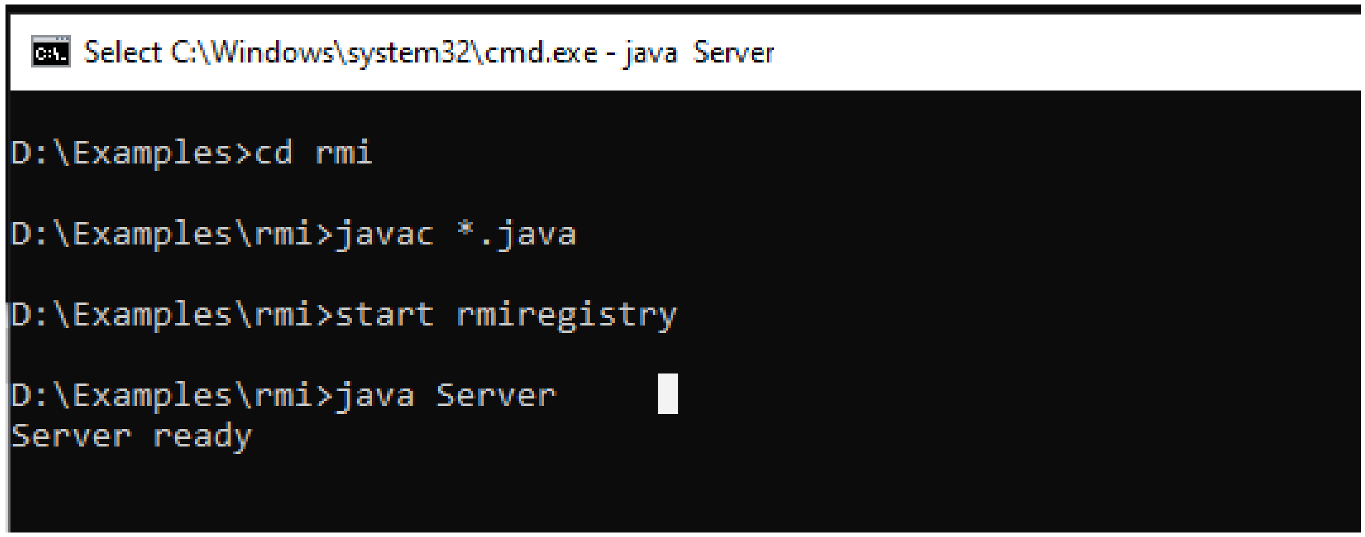
```
D:\>cd examples

D:\Examples>cd rmi

D:\Examples\rmi>javac *.java

D:\Examples\rmi>start rmiregistry

D:\Examples\rmi>
```

Running the RMI System

Executing the Application

- Step 2 – Run class the server file.

  java Server

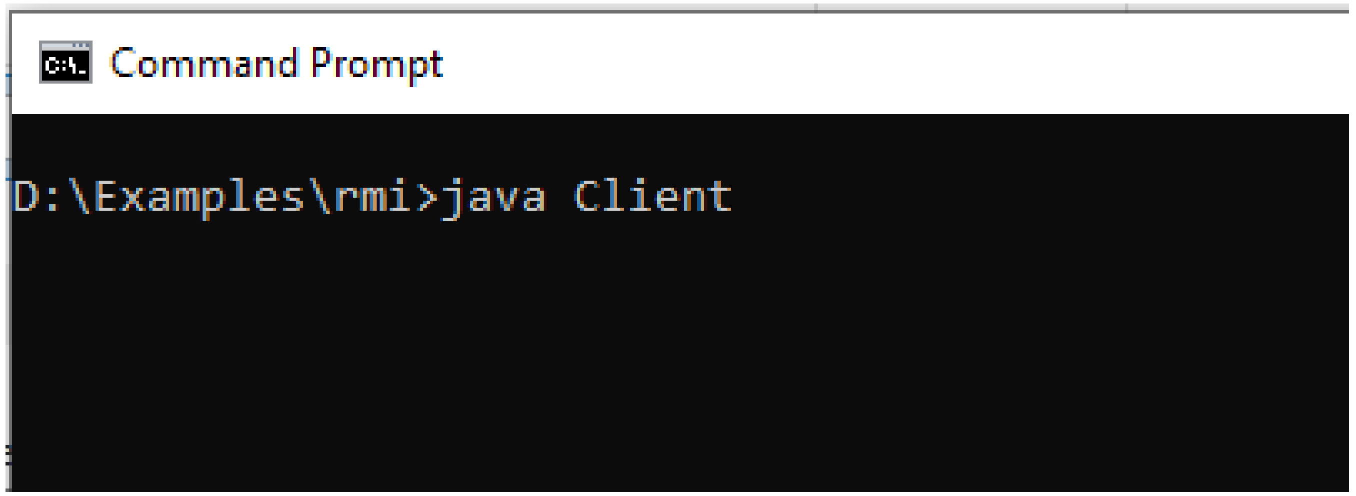


Select C:\Windows\system32\cmd.exe - java Server

```
D:\Examples>cd rmi

D:\Examples\rmi>javac *.java

D:\Examples\rmi>start rmiregistry

D:\Examples\rmi>java Server
Server ready
```

- Step 3 – Run class the client file.

  java Client



Command Prompt

```
D:\Examples\rmi>java Client
```

Buffer Type

Java NIO buffers can be classified in following variants on the basis of data types the buffer deals with −

- ByteBuffer
- MappedByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

Important methods of Buffer

- allocate(int capacity) − use to allocate a new buffer with capacity as parameter. Allocate method throws IllegalArgumentException in case the passed capacity is a negative integer.
- read() and put() − read method of channel is used to write data from channel to buffer while put is a method of buffer which is used to write data in buffer.
- flip() − The flip method switches the mode of Buffer from writing to reading mode. It also sets the position back to 0, and sets the limit to where position was at time of writing.
- write() and get() − write method of channel is used to write data from buffer to channel while get is a method of buffer which is used to read data from buffer.
- rewind() − used when reread is required as it sets the position back to zero and do not alter the value of limit.
- clear() and compact() − clear and compact both methods are used to make buffer from read to write mode. clear() method makes the position to zero and limit equals to capacity, in this method the data in the buffer is not cleared only the markers get re initialized.
- On other hand compact() method is use when there remained some un-read data and still we use write mode of buffer in this case compact method copies all unread data to the beginning of the buffer and sets position to right after the last unread element. The limit property is still set to capacity.
- mark() and reset() − mark is used to mark any particular position in a buffer while reset make position back to marked position.