




CORE JAVA

By : Kalpesh Chauhan



RMI (REMOTE METHOD INVOCATION)



The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.


UNDERSTANDING STUB AND SKELETON

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:



STUB




The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.



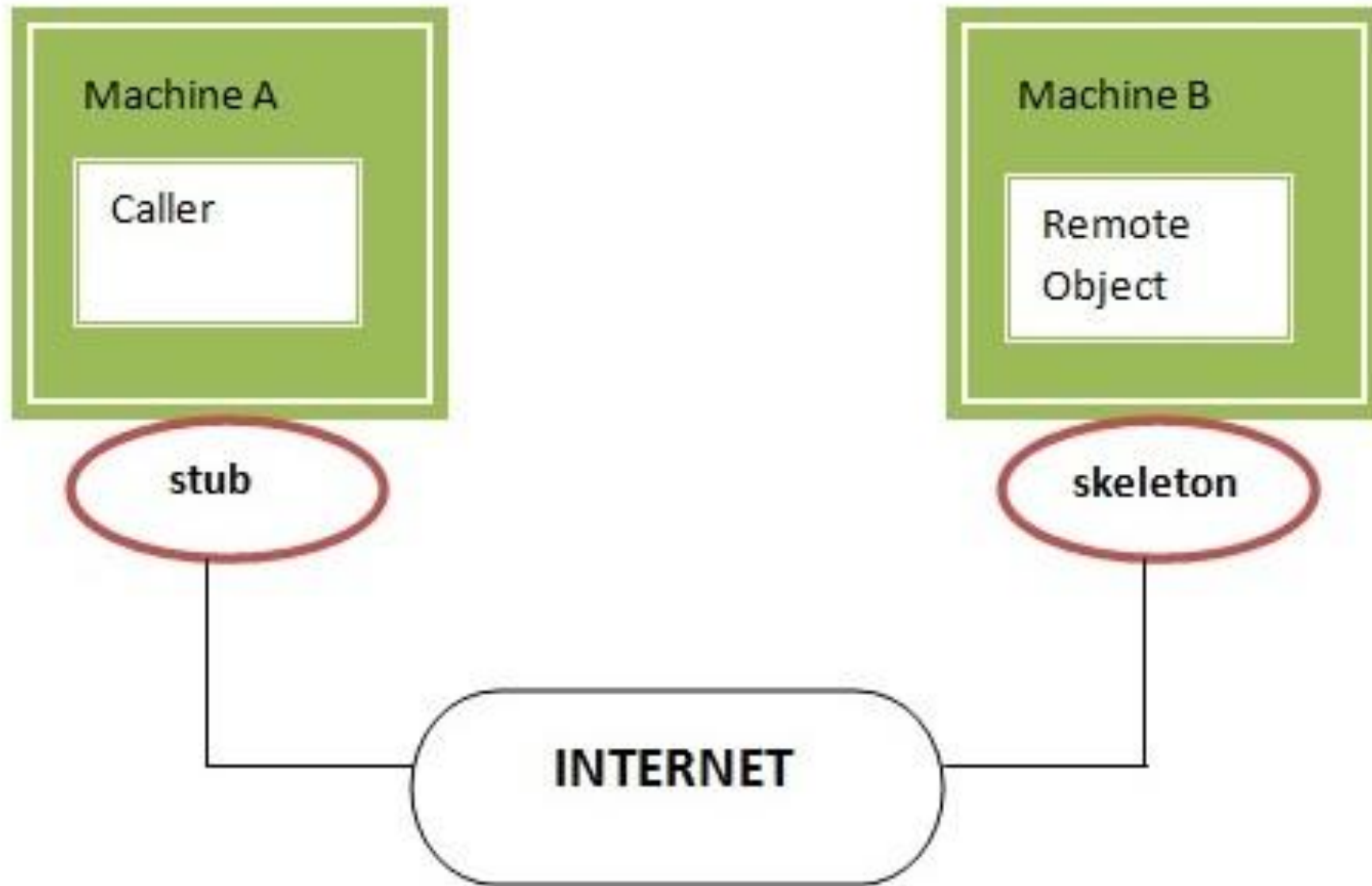
SKELETON



The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.





UNDERSTANDING REQUIREMENTS FOR THE DISTRIBUTED APPLICATIONS




If any application performs these tasks, it can be distributed application.

1. The application need to locate the remote method
2. It need to provide the communication with the remote objects, and
3. The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.




A SIMPLE CLIENT/SERVER APPLICATION USING RMI




This section provides step-by-step directions for building a simple client/server application by using RMI. The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.



STEP ONE: ENTER AND COMPILE THE SOURCE CODE



This application uses four source files. The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException**.



```
import java.rmi.*;

public interface AddServerIntf extends Remote {

    double add(double d1, double d2) throws RemoteException;

}
```


The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add()** method is straightforward. Remote objects typically extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.


```
import java.rmi.*;
import java.rmi.server.*;

public class AddServerImpl extends UnicastRemoteObject
    implements AddServerIntf {
    public AddServerImpl() throws RemoteException {
    }

    public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
    }
}
```

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the **rebind()** method of the **Naming** class (found in **java.rmi**). That method associates a name with an object reference. The first argument to the **rebind()** method is a string that names the server as "AddServer". Its second argument is a reference to an instance of **AddServerImpl**.

```
import java.net.*;
import java.rmi.*;
public class AddServer {
    public static void main(String args[]) {
        try {
            AddServerImpl addServerImpl = new AddServerImpl();
            Naming.rebind("AddServer", addServerImpl);
        }
        catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```



The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string "AddServer".

The program then invokes the **lookup()** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object. The program continues by displaying its arguments and then invokes the remote **add()** method. The sum is returned from this method and is then printed.

```
import java.rmi.*;

public class AddClient {

    public static void main(String args[]) {

        try {

            String addServerURL = "rmi://" + args[0] + "/AddServer";

            AddServerIntf addServerIntf =

                (AddServerIntf)Naming.lookup(addServerURL);

            System.out.println("The first number is: " + args[1]);

            double d1 = Double.valueOf(args[1]).doubleValue();

            System.out.println("The second number is: " + args[2]);
```


```
double d2 = Double.valueOf(args[2]).doubleValue();
System.out.println("The sum is: " + addServerIntf.add(d1, d2));
}
catch(Exception e) {
System.out.println("Exception: " + e);
}
}
}
```





After you enter all the code, use **javac** to compile the four source files that you created.




STEP TWO: MANUALLY GENERATE A STUB IF REQUIRE



In the context of RMI, a *stub* is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.



A remote method may accept arguments that are simple types or objects. In the latter case, the object may have references to other objects. All of this information must be sent to the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. the serialization facilities also recursively process all referenced objects.



If a response must be returned to the client, the process works in reverse. Note that the serialization and deserialization facilities are also used if objects are returned to a client. Prior to Java 5, stubs needed to be built manually by using **rmic**. This step is not required for modern versions of Java. However, if you are working in a legacy environment, then you can use the **rmic** compiler, as shown here, to build a stub:

rmic AddServerImpl



This command generates the file **AddServerImpl_Stub.class**. When using **rmic**, be sure that **CLASSPATH** is set to include the current directory.




STEP THREE: INSTALL FILES ON THE CLIENT AND SERVER MACHINES

Copy **AddClient.class**, **AddServerImpl_Stub.class** (if needed), and **AddServerIntf.class** to a directory on the client machine. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Stub.class** (if needed), and **AddServer.class** to a directory on the server machine.



STEP FOUR: START THE RMI REGISTRY ON THE SERVER MACHINE



The JDK provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. First, check that the **CLASSPATH** environment variable includes the directory in which your files are located. Then, start the RMI Registry from the command line, as shown here:

start rmiregistry

When this command returns, you should see that a new window has been created. You need to leave this window open until you are done experimenting with the RMI example.



STEP FIVE: START THE SERVER




The server code is started from the command line, as shown here:

java AddServer

Recall that the **AddServer** code instantiates **AddServerImpl** and registers that object with the name "AddServer".



STEP SIX: START THE CLIENT



The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

```
java AddClient <SERVER IP ADDRESS> 8 9
```



JAVA NETWORKING



Java Networking is a concept of connecting two or more computing devices together so that we can share resources.


Java socket programming provides facility to share data between different computing devices.

ADVANTAGE OF JAVA NETWORKING

1. sharing resources
2. centralize software management



JAVA NETWORKING TERMINOLOGY

- 
1. IP Address
 2. Protocol
 3. Port Number
 4. MAC Address
 5. Connection-oriented and connection-less protocol
 6. Socket

IP ADDRESS

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

It is a logical address that can be changed.

PROTOCOL

A protocol is a set of rules basically that is followed for communication. For example:

1. TCP
2. FTP
3. Telnet
4. SMTP
5. POP etc.

PORT NUMBER

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

MAC ADDRESS

MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.

CONNECTION-ORIENTED AND CONNECTION-LESS PROTOCOL

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.

SOCKET

A socket is an endpoint between two-way communication.



JAVA SOCKET PROGRAMMING




Java Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less.

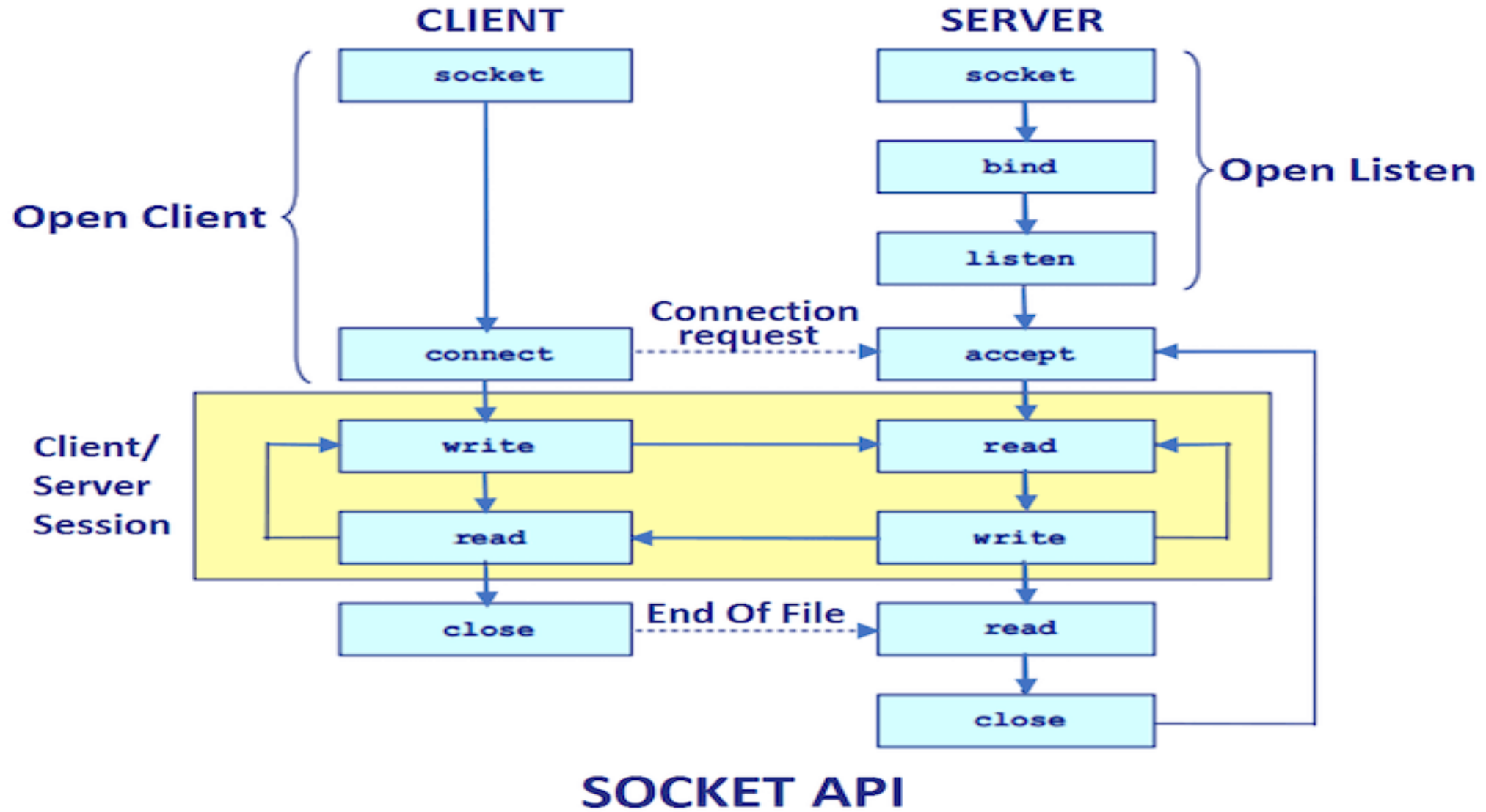
Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.



Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



SOCKET CLASS

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

SERVERSOCKET CLASS

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

EXAMPLE OF JAVA SOCKET PROGRAMMING

Creating Server:

To create the server application, we need to create the instance of `ServerSocket` class. Here, we are using 6666 port number for the communication between the client and server. You may also choose any other port number. The `accept()` method waits for the client. If clients connects with the given port number, it returns an instance of `Socket`.



```
ServerSocket ss=new ServerSocket(6666);
```

```
Socket s=ss.accept();//establishes connection and waits for the client
```

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

```
Socket s=new Socket("localhost",6666);
```

```
import java.io.*;
import java.net.*;
public class MyServer {
    public static void main(String[] args){
        try{
            ServerSocket ss=new ServerSocket(6666);
            Socket s=ss.accept();//establishes connection
            DataInputStream dis=new DataInputStream(s.getInputStream());
            String str=(String)dis.readUTF();
            System.out.println("message= "+str);
            ss.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

```
import java.io.*;
import java.net.*;
public class MyClient {
    public static void main(String[] args) {
        try{
            Socket s=new Socket("localhost",6666);
            DataOutputStream dout=new DataOutputStream(s.getOutputStream());
            dout.writeUTF("Hello Server");
            dout.flush();
            dout.close();
            s.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```



To execute this program open two command prompts and execute each program at each command prompt.

EXAMPLE OF JAVA SOCKET PROGRAMMING (READ-WRITE BOTH SIDE)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

Server Example

Client Example



JAVA URL



The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

<https://www.w3schools.com/php>



Protocol: In this case, http is the protocol.

Server name or IP Address: In this case, www.w3schools.com is the server name.

Port Number: It is an optional attribute. If we write `http://www.w3schools.com:80/sonoojaiswal/` , 80 is the port number. If port number is not mentioned in the URL, it returns -1.

File Name or directory name: In this case, index.php is the file name.

EXAMPLE OF JAVA URL CLASS

```
//URLDemo.java
import java.net.*;
public class URLDemo{
public static void main(String[] args){
try{
    URL url=new URL("http://www.w3schools.com/php");
    System.out.println("Protocol: "+url.getProtocol());
    System.out.println("Host Name: "+url.getHost());
    System.out.println("Port Number: "+url.getPort());
    System.out.println("File Name: "+url.getFile());
}catch(Exception e){System.out.println(e);}
}
}
```

JAVA URLCONNECTION CLASS

The **Java URLConnection** class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

DISPLAYING SOURCE CODE OF A WEBPAGE BY URLCONNECTON CLASS

The `URLConnection` class provides many methods, we can display all the data of a webpage by using the `getInputStream()` method. The `getInputStream()` method returns all the data of the specified URL in the stream that can be read and displayed.

```
import java.io.*;
import java.net.*;
public class URLConnectionExample {
    public static void main(String[] args){
        try{
            URL url=new URL("http://www.w3schools.com/php");
            URLConnection urlcon=url.openConnection();
            InputStream stream=urlcon.getInputStream();
            int i;
            while((i=stream.read())!=-1){ System.out.print((char)i); }
        }catch(Exception e){System.out.println(e);}
    }
}
```

JAVA HTTPURLConnection CLASS

The **Java HttpURLConnection** class is http specific URLConnection. It works for HTTP protocol only.

By the help of HttpURLConnection class, you can information of any HTTP URL such as header information, status code, response code etc.

The `java.net.HttpURLConnection` is subclass of URLConnection class.

```
import java.io.*;
import java.net.*;
public class HttpURLConnectionDemo{
    public static void main(String[] args){
        try{
            URL url=new URL("http://www.w3schools.com/php");
            HttpURLConnection huc=(HttpURLConnection)url.openConnection();
            for(int i=1;i<=8;i++){
                System.out.println(huc.getHeaderFieldKey(i)+" = "+huc.getHeaderField(i));
            }
            huc.disconnect();
        }catch(Exception e){System.out.println(e);}
    }
}
```

JAVA INETADDRESS CLASS

Java InetAddress class represents an IP address. The `java.net.InetAddress` class provides methods to get the IP of any host name *for example* `www.google.com`, `www.facebook.com`, etc.


```
import java.io.*;
import java.net.*;
public class InetDemo{
    public static void main(String[] args){
        try{
            InetAddress ip=InetAddress.getByName("www.google.com");

            System.out.println("Host Name: "+ip.getHostName());
            System.out.println("IP Address: "+ip.getHostAddress());
        }catch(Exception e){System.out.println(e);}
    }
}
```



CONTINUE IN NEXT UNIT.....