

UNIT II

Client/Server Communication

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

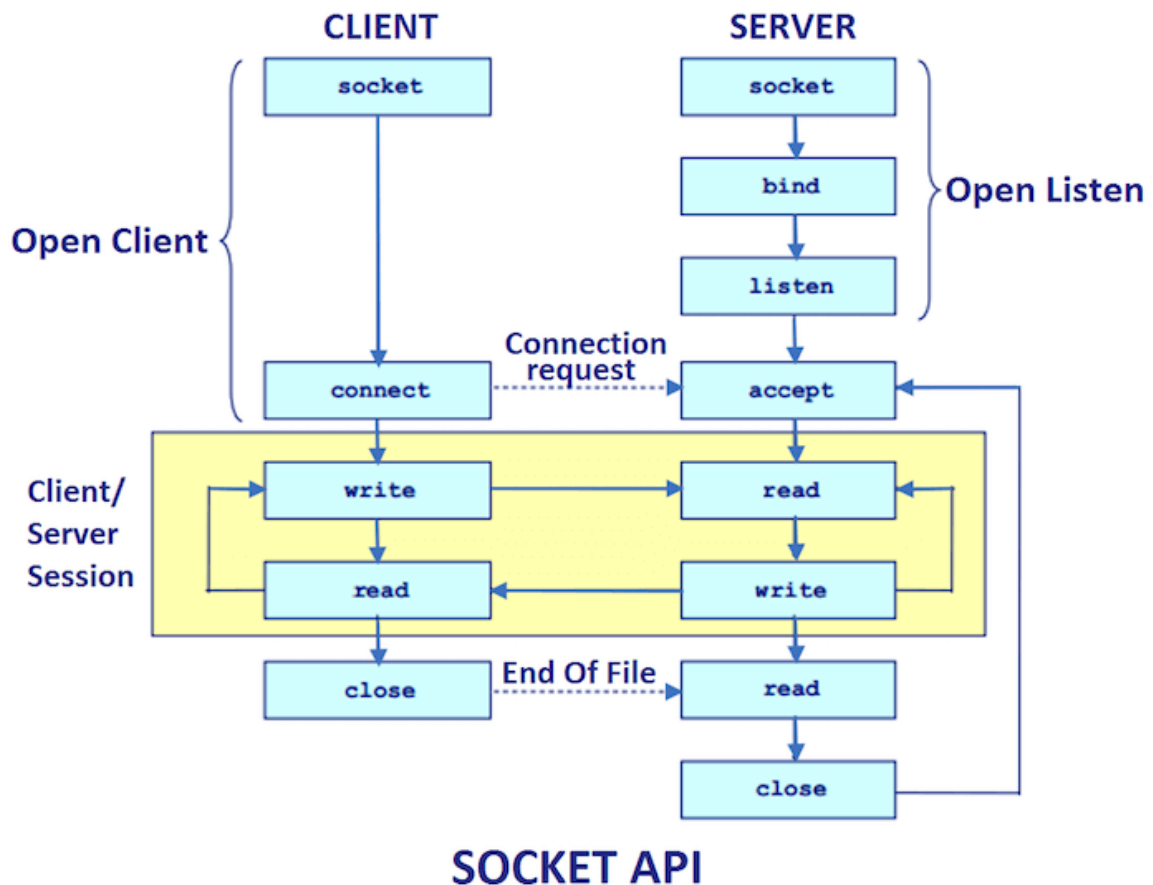
Java Socket programming can be connection-oriented or connection-less.

Socket and Server Socket classes are used for connection-oriented socket programming and Datagram Socket and Datagram Packet 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 Server Socket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The Server Socket class is used at server-side. The accept() method of Server Socket 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.

Important methods

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
-------------------------------------	--------------------

Server Socket class

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

Important methods

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.

Java Socket Programming

Creating Server:

To create the server application, we need to create the instance of Server Socket 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.

1. `ServerSocket ss=new ServerSocket(6666);`
2. `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.

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

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

File: MyServer.java

1. `import java.io.*;`
2. `import java.net.*;`
3. `public class MyServer {`
4. `public static void main(String[] args){`
5. `try{`
6. `ServerSocket ss=new ServerSocket(6666);`
7. `Socket s=ss.accept();` // establishes connection
8. `DataInputStream dis=new DataInputStream(s.getInputStream());`
9. `String str=(String)dis.readUTF();`
10. `System.out.println("message= "+str);`
11. `ss.close();`
12. `}catch(Exception e){System.out.println(e);}`
13. `}`
14. `}`
- 15.

File: MyClient.java

```
1. import java.io.*;
2. import java.net.*;
3. public class MyClient {
4.     public static void main(String[] args) {
5.         try{
6.             Socket s=new Socket("localhost",6666);
7.             DataOutputStream dout=new DataOutputStream(s.getOutputStream());
8.             dout.writeUTF("Hello Server");
9.             dout.flush();
10.            dout.close();
11.            s.close();
12.        }catch(Exception e){System.out.println(e);}
13.    }
14. }
```

HostsIdentification

Every computer on the Internet is identified by a unique, 4-byte IP address . This is typically written in

dotted quad format like 128.250.25.158 where each byte is an unsigned value between 0 and 255. This

representation is clearly not user-friendly because it does not tell us anything about the content and then it is

difficult to remember. Hence, IP addresses are mapped to names

like www.musiclamhe.com or www.Dicoor.com, which are easier to remember.

Internet supports name servers that translate these names to IP addresses.

Service Ports

Ports are logical abstractions that allow one host to communicate simultaneously with many other hosts.

Many services run on well-known ports. For example, http tends to run on port 80.

In general, each computer only has one Internet address. However, computers often need to communicate and provide more than one type of service or to talk to multiple hosts/computers at a time.

For example, there may be multiple ftp sessions, web connections, and chat programs all running at the same time. To distinguish these services, a concept of port's, a logical access point, represented by a 16-bit integer number is used.

That means, each service offered by a computer is uniquely identified by a port number. Each Internet packet contains both the destination host address and the port number on that host to which the message/request has to be delivered. The host computer dispatches the packets it receives to programs by looking at the port numbers specified within the packets. That is, IP address can be thought of as a house address when a letter is sent via post/snail mail and port number as the name of a specific individual to whom the letter has to be delivered.

Choosing the Host and the Port

- You must at least specify the remote host and port to connect to.
- The host may be specified as either a string like "utopia.poly.edu" or as an InetAddress object.
- The port should be an int between 1 and 65535.
`Socket webMetalab = new Socket("musiclamhe.com", 80);`
- You cannot just connect to any port on any host. The remote host must actually be listening for connections on that port.
- You can use the constructors to determine which ports on a host are listening for connections.

SOCKET PROGRAMMING AND JAVA.NET CLASS

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols –

- **TCP** – TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP** – UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects –

- **Socket Programming** – This is the most widely used concept in Networking and it has been explained in very detail.
- **URL Processing** – This would be covered separately. Click here to learn about [URL Processing](#) in Java language.

Socket Programming

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

The `java.net.Socket` class represents a socket, and the `java.net.ServerSocket` class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets –

- The server instantiates a `ServerSocket` object, denoting which port number communication is to occur on.
- The server invokes the `accept()` method of the `ServerSocket` class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a `Socket` object, specifying the server name and the port number to connect to.
- The constructor of the `Socket` class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a `Socket` object capable of communicating with the server.
- On the server side, the `accept()` method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an `OutputStream` and an `InputStream`. The client's `OutputStream` is connected to the server's `InputStream`, and the client's `InputStream` is connected to the server's `OutputStream`.

TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.

ServerSocket Class Methods

The **`java.net.ServerSocket`** class is used by server applications to obtain a port and listen for client requests.

The `ServerSocket` class has four constructors –

Sr.No.	Method & Description
1	public ServerSocket(int port) throws IOException Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	public ServerSocket(int port, int backlog) throws IOException Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.
3	public ServerSocket(int port, int backlog, InetAddress address) throws IOException Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on.
4	public ServerSocket() throws IOException Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket.

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Following are some of the common methods of the ServerSocket class –

Sr.No.	Method & Description
--------	----------------------

1	<p>public int getLocalPort()</p> <p>Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.</p>
2	<p>public Socket accept() throws IOException</p> <p>Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely.</p>
3	<p>public void setSoTimeout(int timeout)</p> <p>Sets the time-out value for how long the server socket waits for a client during the accept().</p>
4	<p>public void bind(SocketAddress host, int backlog)</p> <p>Binds the socket to the specified server and port in the SocketAddress object. Use this method if you have instantiated the ServerSocket using the no-argument constructor.</p>

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and the server, and communication can begin.

Socket Class Methods

The **java.net.Socket** class represents the socket that both the client and the server use to communicate with each other. The client obtains a Socket object by instantiating

one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server –

Sr.No.	Method & Description
1	public Socket(String host, int port) throws UnknownHostException, IOException. This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2	public Socket(InetAddress host, int port) throws IOException This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.
3	public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException. Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException. This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String.
5	public Socket()

	Creates an unconnected socket. Use the connect() method to connect this socket to a server.
--	---

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and the server have a Socket object, so these methods can be invoked by both the client and the server.

Sr.No.	Method & Description
1	public void connect(SocketAddress host, int timeout) throws IOException This method connects the socket to the specified host. This method is needed only when you instantiate the Socket using the no-argument constructor.
2	public InetAddress getInetAddress() This method returns the address of the other computer that this socket is connected to.
3	public int getPort() Returns the port the socket is bound to on the remote machine.
4	public int getLocalPort() Returns the port the socket is bound to on the local machine.
5	public SocketAddress getRemoteSocketAddress() Returns the address of the remote socket.

6	public InputStream getInputStream() throws IOException Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7	public OutputStream getOutputStream() throws IOException Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket.
8	public void close() throws IOException Closes the socket, which makes this Socket object no longer capable of connecting again to any server.

InetAddress Class Methods

This class represents an Internet Protocol (IP) address. Here are following usefull methods which you would need while doing socket programming –

Sr.No.	Method & Description
1	static InetAddress getByAddress(byte[] addr) Returns an InetAddress object given the raw IP address.
2	static InetAddress getByAddress(String host, byte[] addr) Creates an InetAddress based on the provided host name and IP address.
3	static InetAddress getByName(String host) Determines the IP address of a host, given the host's name.

4	String getHostAddress() Returns the IP address string in textual presentation.
5	String getHostName() Gets the host name for this IP address.
6	static InetAddress InetAddress getLocalHost() Returns the local host.
7	String toString() Converts this IP address to a String.

Socket Client Example

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

Example

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;

public class GreetingClient {

    public static void main(String [] args) {
        String serverName = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            System.out.println("Connecting to " + serverName + " on port " + port);
```

```

Socket client = new Socket(serverName, port);

System.out.println("Just connected to " + client.getRemoteSocketAddress());
OutputStream outToServer = client.getOutputStream();
DataOutputStream out = new DataOutputStream(outToServer);

out.writeUTF("Hello from " + client.getLocalSocketAddress());
InputStream inFromServer = client.getInputStream();
DataInputStream in = new DataInputStream(inFromServer);

System.out.println("Server says " + in.readUTF());
client.close();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

Socket Server Example

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument –

Example

```

// File Name GreetingServer.java
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
    private ServerSocket serverSocket;

```

```
public GreetingServer(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    serverSocket.setSoTimeout(10000);
}

public void run() {
    while(true) {
        try {
            System.out.println("Waiting for client on port " +
                serverSocket.getLocalPort() + "...");
            Socket server = serverSocket.accept();

            System.out.println("Just connected to " + server.getRemoteSocketAddress());
            DataInputStream in = new DataInputStream(server.getInputStream());

            System.out.println(in.readUTF());
            DataOutputStream out = new DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress()
                + "\nGoodbye!");
            server.close();

        } catch (SocketTimeoutException s) {
            System.out.println("Socket timed out!");
            break;
        } catch (IOException e) {
            e.printStackTrace();
            break;
        }
    }
}
```



```
public static void main(String [] args) {  
    int port = Integer.parseInt(args[0]);  
    try {  
        Thread t = new GreetingServer(port);  
        t.start();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Compile the client and the server and then start the server as follows –

```
$ java GreetingServer 6066  
Waiting for client on port 6066...
```

Check the client program as follows –

Output

```
$ java GreetingClient localhost 6066  
Connecting to localhost on port 6066  
Just connected to localhost/127.0.0.1:6066  
Server says Thank you for connecting to /127.0.0.1:6066  
Goodbye!
```

UDP SOCKET PROGRAMMING

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming using the UDP instead of TCP.

Datagram

Datagrams are collection of information sent from one device to another device via the established network. When the datagram is sent to the targeted device, there is no assurance that it will reach to the target device safely and completely. It may get damaged or lost in between. Likewise, the receiving device also never know if the datagram received is damaged or not. The UDP protocol is used to implement the datagrams in Java.

Java DatagramSocket class

Java DatagramSocket class represents a connection-less socket for sending and receiving datagram packets. It is a mechanism used for transmitting datagram packets over network.`

A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.

Commonly used Constructors of DatagramSocket class

- **DatagramSocket()** throws **SocketEeption**: it creates a datagram socket and binds it with the available Port Number on the localhost machine.
- **DatagramSocket(int port)** throws **SocketEeption**: it creates a datagram socket and binds it with the given Port Number.
- **DatagramSocket(int port, InetAddress address)** throws **SocketEeption**: it creates a datagram socket and binds it with the specified port number and host address.

Java DatagramSocket Class

Method	Description
void bind(SocketAddress addr)	It binds the DatagramSocket to a specific address and port.
void close()	It closes the datagram socket.
void connect(InetAddress address, int port)	It connects the socket to a remote address for the socket.
void disconnect()	It disconnects the socket.
boolean getBroadcast()	It tests if SO_BROADCAST is enabled.
DatagramChannel getChannel()	It returns the unique DatagramChannel object associated with the datagram socket.
InetAddress getInetAddress()	It returns the address to where the socket is connected.
InetAddress getLocalAddress()	It gets the local address to which the socket is connected.
int getLocalPort()	It returns the port number on the local host to which the socket is bound.
SocketAddress getLocalSocketAddress()	It returns the address of the endpoint the socket is bound to.
int getPort()	It returns the port number to which the socket is connected.
int getReceiverBufferSize()	It gets the value of the SO_RCVBUF option for this DatagramSocket that is the buffer size used by the platform for input on the DatagramSocket.

<code>boolean isClosed()</code>	It returns the status of socket i.e. closed or not.
<code>boolean isConnected()</code>	It returns the connection state of the socket.
<code>void send(DatagramPacket p)</code>	It sends the datagram packet from the socket.
<code>void receive(DatagramPacket p)</code>	It receives the datagram packet from the socket.

Java DatagramPacket Class

Java DatagramPacket is a message that can be sent or received. It is a data container. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

Commonly used Constructors of DatagramPacket class

- **DatagramPacket(byte[] barr, int length):** it creates a datagram packet. This constructor is used to receive the packets.
- **DatagramPacket(byte[] barr, int length, InetAddress address, int port):** it creates a datagram packet. This constructor is used to send the packets.

Java DatagramPacket Class Methods

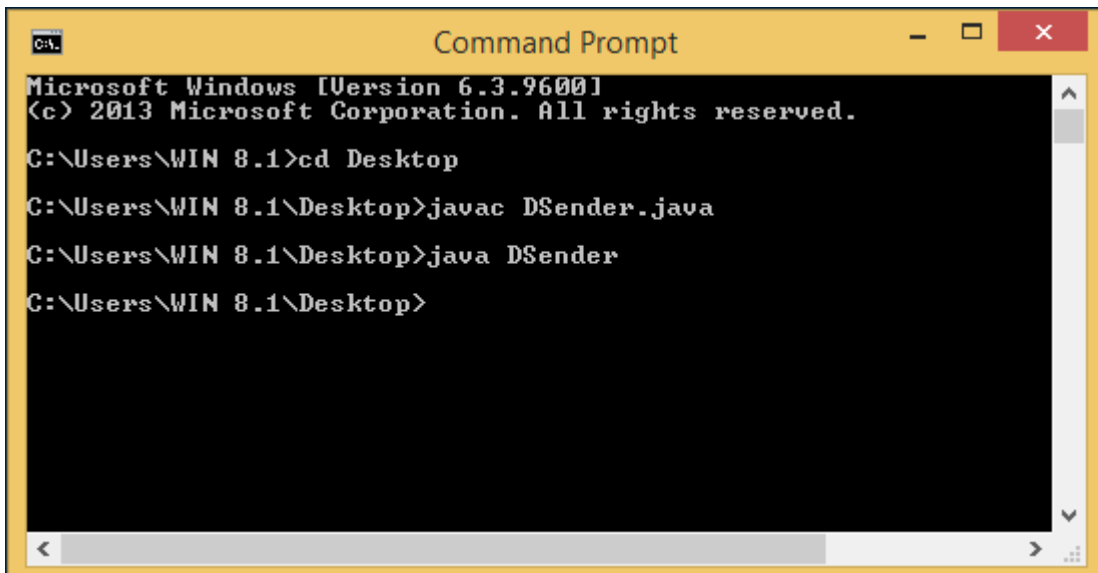
Method	Description
1) <code>InetAddress getAddress()</code>	It returns the IP address of the machine to which the datagram is being sent or from which the datagram was received.
2) <code>byte[] getData()</code>	It returns the data buffer.

3) int getLength()	It returns the length of the data to be sent or the length of the data received.
4) int getOffset()	It returns the offset of the data to be sent or the offset of the data received.
5) int getPort()	It returns the port number on the remote host to which the datagram is being sent or from which the datagram was received.
6) SocketAddress getSocketAddress()	It gets the SocketAddress (IP address + port number) of the remote host that the packet is being sent to or is coming from.
7) void setAddress(InetAddress iaddr)	It sets the IP address of the machine to which the datagram is being sent.
8) void setData(byte[] buff)	It sets the data buffer for the packet.
9) void setLength(int length)	It sets the length of the packet.
10) void setPort(int iport)	It sets the port number on the remote host to which the datagram is being sent.
11) void setSocketAddress(SocketAddress addr)	It sets the SocketAddress (IP address + port number) of the remote host to which the datagram is being sent.

Example of Sending DatagramPacket by DatagramSocket

```
1. //DSender.java
2. import java.net.*;
3. public class DSender{
4.     public static void main(String[] args) throws Exception {
5.         DatagramSocket ds = new DatagramSocket();
6.         String str = "Welcome java";
7.         InetAddress ip = InetAddress.getByName("127.0.0.1");
8.
9.         DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3
000);
10.        ds.send(dp);
11.        ds.close();
12.    }
13. }
```

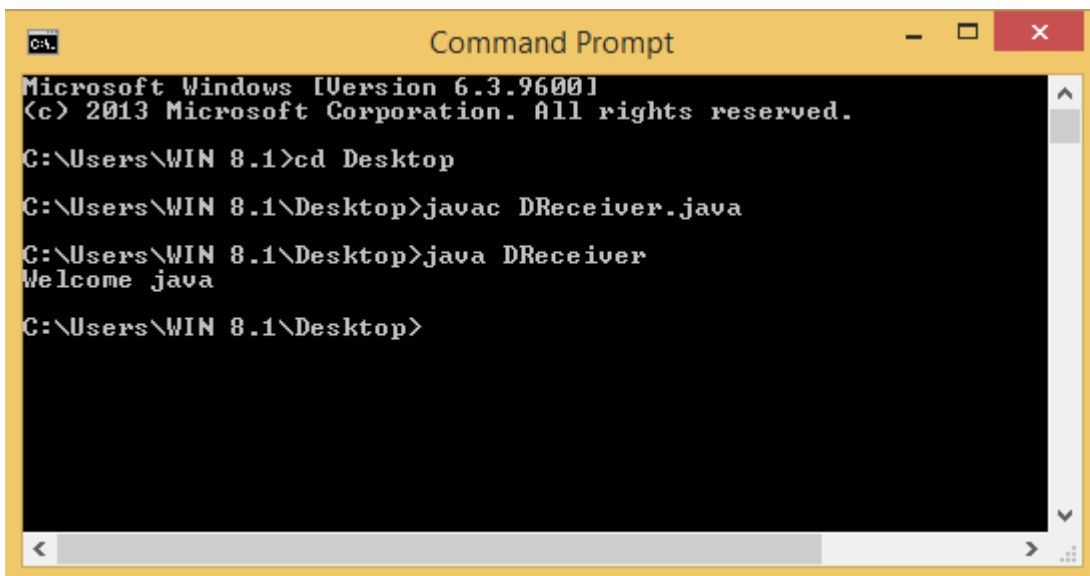
Output:

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window has a yellow title bar with standard Windows window controls (minimize, maximize, close). The background is black with white text. The text shows the following commands and their execution paths:
Microsoft Windows [Version 6.3.9600]
<C> 2013 Microsoft Corporation. All rights reserved.
C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac DSender.java
C:\Users\WIN 8.1\Desktop>java DSender
C:\Users\WIN 8.1\Desktop>
The window includes a scroll bar on the right side.

Example of Receiving DatagramPacket by DatagramSocket

```
1. //DReceiver.java
2. import java.net.*;
3. public class DReceiver{
4.     public static void main(String[] args) throws Exception {
5.         DatagramSocket ds = new DatagramSocket(3000);
6.         byte[] buf = new byte[1024];
7.         DatagramPacket dp = new DatagramPacket(buf, 1024);
8.         ds.receive(dp);
9.         String str = new String(dp.getData(), 0, dp.getLength());
10.        System.out.println(str);
11.        ds.close();
12.    }
13. }
```

Output:



```
Command Prompt
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac DReceiver.java
C:\Users\WIN 8.1\Desktop>java DReceiver
Welcome java
C:\Users\WIN 8.1\Desktop>
```

URL ENCODING

URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory.

This section shows you how to write Java programs that communicate with a URL. A URL can be broken down into parts, as follows –

```
protocol://host:port/path?query#ref
```

Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority.

The following is a URL to a web page whose protocol is HTTP –

```
https://www.amrood.com/index.htm?language=en#j2se
```

Notice that this URL does not specify a port, in which case the default port for the protocol is used. With HTTP, the default port is 80.

Constructors

The **java.net.URL** class represents a URL and has a complete set of methods to manipulate URL in Java.

The URL class has several constructors for creating URLs, including the following –

Sr.No.	Constructors & Description
1	public URL(String protocol, String host, int port, String file) throws MalformedURLException Creates a URL by putting together the given parts.
2	public URL(String protocol, String host, String file) throws MalformedURLException

	Identical to the previous constructor, except that the default port for the given protocol is used.
3	public URL(String url) throws MalformedURLException Creates a URL from the given String.
4	public URL(URL context, String url) throws MalformedURLException Creates a URL by parsing together the URL and String arguments.

The URL class contains many methods for accessing the various parts of the URL being represented. Some of the methods in the URL class include the following –

Sr.No.	Method & Description
1	public String getPath() Returns the path of the URL.
2	public String getQuery() Returns the query part of the URL.
3	public String getAuthority() Returns the authority of the URL.
4	public int getPort() Returns the port of the URL.

5	public int getDefaultPort() Returns the default port for the protocol of the URL.
6	public String getProtocol() Returns the protocol of the URL.
7	public String getHost() Returns the host of the URL.
8	public String getHost() Returns the host of the URL.
9	public String getFile() Returns the filename of the URL.
10	public String getRef() Returns the reference part of the URL.
11	public URLConnection openConnection() throws IOException Opens a connection to the URL, allowing a client to communicate with the resource.

Example

The following URLLDemo program demonstrates the various parts of a URL. A URL is entered on the command line, and the URLLDemo program outputs each part of the given URL.

// File Name : URLEDemo.java

```
import java.net.*;
import java.io.*;

public class URLEDemo {

    public static void main(String [] args) {
        try {
            URL url = new
URL("https://www.amrood.com/index.htm?language=en#j2se");

            System.out.println("URL is " + url.toString());
            System.out.println("protocol is " + url.getProtocol());
            System.out.println("authority is " + url.getAuthority());
            System.out.println("file name is " + url.getFile());
            System.out.println("host is " + url.getHost());
            System.out.println("path is " + url.getPath());
            System.out.println("port is " + url.getPort());
            System.out.println("default port is " + url.getDefaultPort());
            System.out.println("query is " + url.getQuery());
            System.out.println("ref is " + url.getRef());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

A sample run of the this program will produce the following result –

Output

```
URL is https://www.amrood.com/index.htm?language=en#j2se
```

protocol is http
authority is www.amrood.com
file name is /index.htm?language=en
host is www.amrood.com
path is /index.htm
port is -1
default port is 80
query is language=en
ref is j2se

URLConnections Class Methods

The `openConnection()` method returns a **java.net.URLConnection**, an abstract class whose subclasses represent the various types of URL connections.

For example –

- If you connect to a URL whose protocol is HTTP, the `openConnection()` method returns an `HttpURLConnection` object.
- If you connect to a URL that represents a JAR file, the `openConnection()` method returns a `JarURLConnection` object, etc.

The `URLConnection` class has many methods for setting or determining information about the connection, including the following –

Sr.No.	Method & Description
1	Object getContent() Retrieves the contents of this URL connection.
2	Object getContent(Class[] classes) Retrieves the contents of this URL connection.

3	String getContentEncoding() Returns the value of the content-encoding header field.
4	int getContentLength() Returns the value of the content-length header field.
5	String getContentType() Returns the value of the content-type header field.
6	int getLastModified() Returns the value of the last-modified header field.
7	long getExpiration() Returns the value of the expired header field.
8	long getIfModifiedSince() Returns the value of this object's ifModifiedSince field.
9	public void setDoInput(boolean input) Passes in true to denote that the connection will be used for input. The default value is true because clients typically read from a URLConnection.
10	public void setDoOutput(boolean output)

	<p>Passes in true to denote that the connection will be used for output. The default value is false because many types of URLs do not support being written to.</p>
11	<p>public InputStream getInputStream() throws IOException</p> <p>Returns the input stream of the URL connection for reading from the resource.</p>
12	<p>public OutputStream getOutputStream() throws IOException</p> <p>Returns the output stream of the URL connection for writing to the resource.</p>
13	<p>public URL getURL()</p> <p>Returns the URL that this URLConnection object is connected to.</p>

Example

The following URLConnectionDemo program connects to a URL entered from the command line.

If the URL represents an HTTP resource, the connection is cast to HttpURLConnection, and the data in the resource is read one line at a time.

```
// File Name : URLConnDemo.java
import java.net.*;
import java.io.*;

public class URLConnDemo {

    public static void main(String [] args) {
        try {
            URL url = new URL("https://www.amrood.com");
```

```

URLConnection urlConnection = url.openConnection();
URLConnection connection = null;
if(urlConnection instanceof HttpURLConnection) {
    connection = (URLConnection) urlConnection;
}else {
    System.out.println("Please enter an HTTP URL.");
    return;
}

BufferedReader in = new BufferedReader(
    new InputStreamReader(connection.getInputStream()));
String urlString = "";
String current;

while((current = in.readLine()) != null) {
    urlString += current;
}

System.out.println(urlString);
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

A sample run of this program will produce the following result –

Output

```
$ java URLConnDemo
```

```
.....a complete HTML content of home page of amrood.com.....
```

Writing and Reading Data Via URL Connections

The **URL** class of the java.net package represents a Uniform Resource Locator which is used to point a resource(file or, directory or a reference) in the world wide web.

This class provides various constructors one of them accepts a String parameter and constructs an object of the URL class.

The *openStream()* method of this class opens a connection to the URL represented by the current object and returns an InputStream object using which you can read data from the URL.

Therefore, to read data from web page (using the URL class) –

- Instantiate the java.net.URL class by passing the URL of the desired web page as a parameter to its constructor.
- Invoke the openStream() method and retrieve the InputStream object.
- Instantiate the Scanner class by passing the above retrieved InputStream object as a parameter.

Example

```
import java.io.IOException;
import java.net.URL;
import java.util.Scanner;

public class ReadingWebPage {

    public static void main(String args[]) throws IOException {

        //Instantiating the URL class
        URL url = new URL("http://www.something.com/");

        //Retrieving the contents of the specified page
        Scanner sc = new Scanner(url.openStream());

        //Instantiating the StringBuffer class to hold the result
        StringBuffer sb = new StringBuffer();
    }
}
```



```

while(sc.hasNext()) {
    sb.append(sc.next());

    //System.out.println(sc.next());
}

//Retrieving the String from the String Buffer object
String result = sb.toString();
System.out.println(result);

//Removing the HTML tags
result = result.replaceAll("<[^>]*>", "");

System.out.println("Contents of the web page: "+result);
}
}

```

Output

```
<html><body><h1>Itworks!</h1></body></html>
```

```
Contents of the web page: Itworks!
```

RMI

RMI stands for Remote Method Invocation. 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 stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

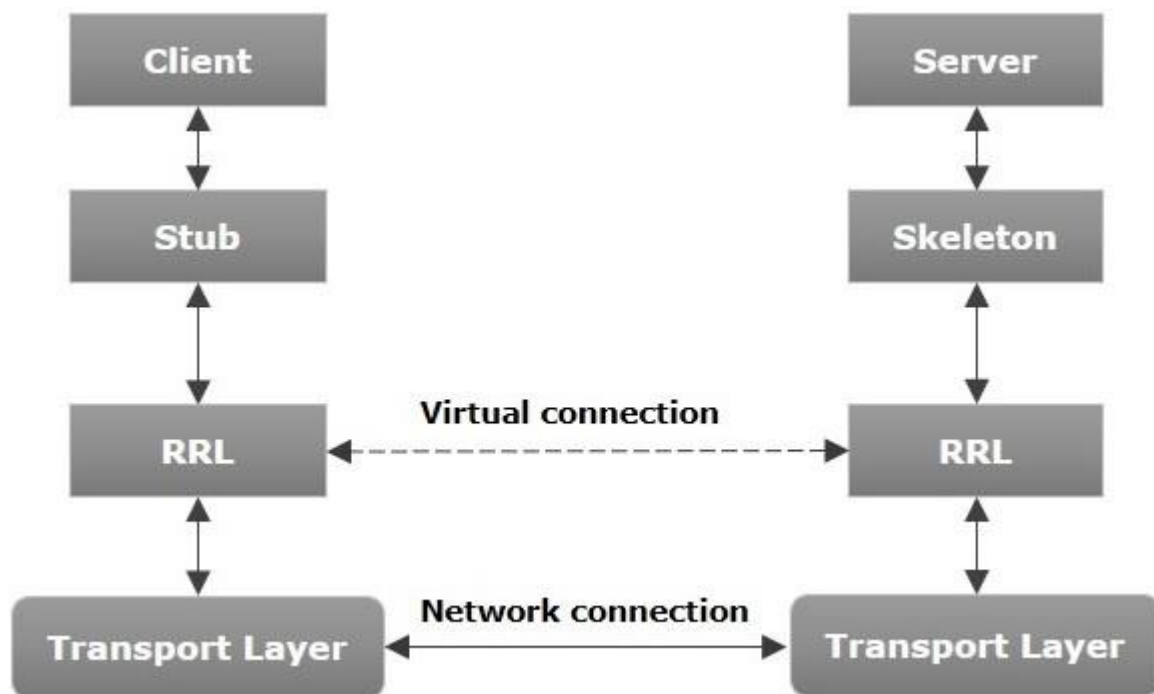
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **`java.rmi`**.

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.

The following diagram shows the architecture of an RMI application.



Let us now discuss the 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

The following points summarize how an RMI application works –

- 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.

Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

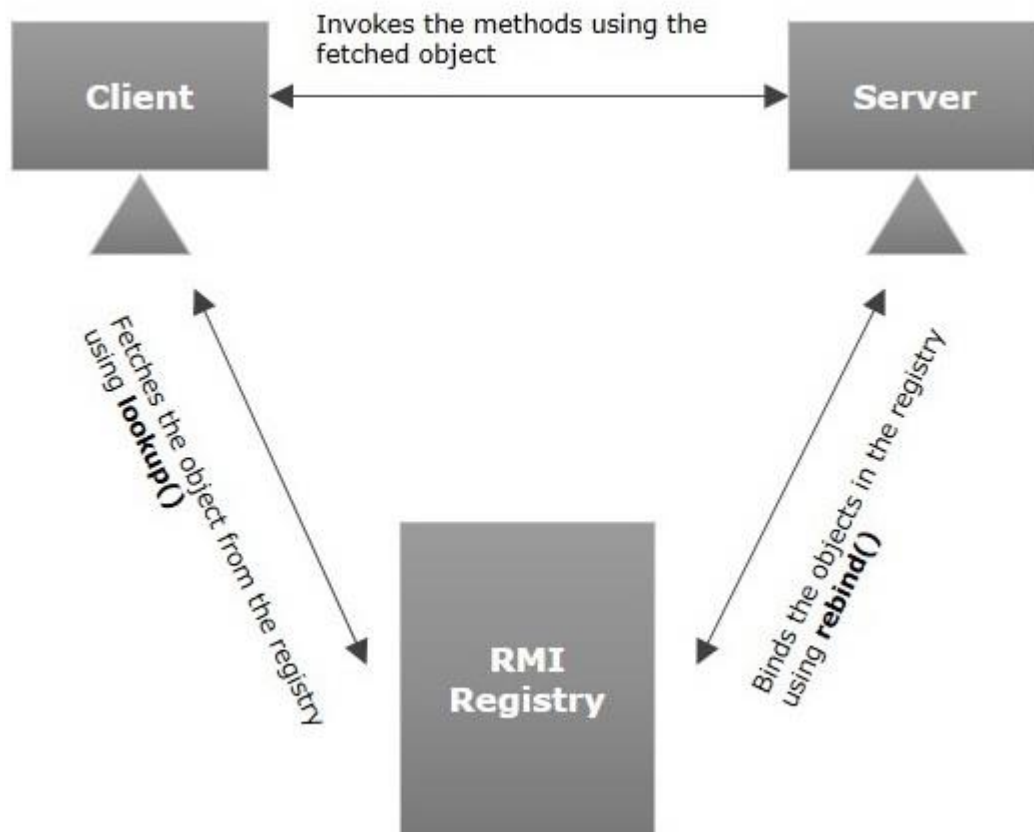
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

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).

The following illustration explains the entire process –



Goals of RMI

Following are the 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

Implementation an RMI Server

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 RMI registry. Create a client class from where you want invoke the remote object.

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.

Following is an example of an RMI server program.

```
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();
}
}
}

```

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.

Following is an example of an RMI client program.

```
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();
        }
    }
}
```

Compiling the Application

To compile the application –

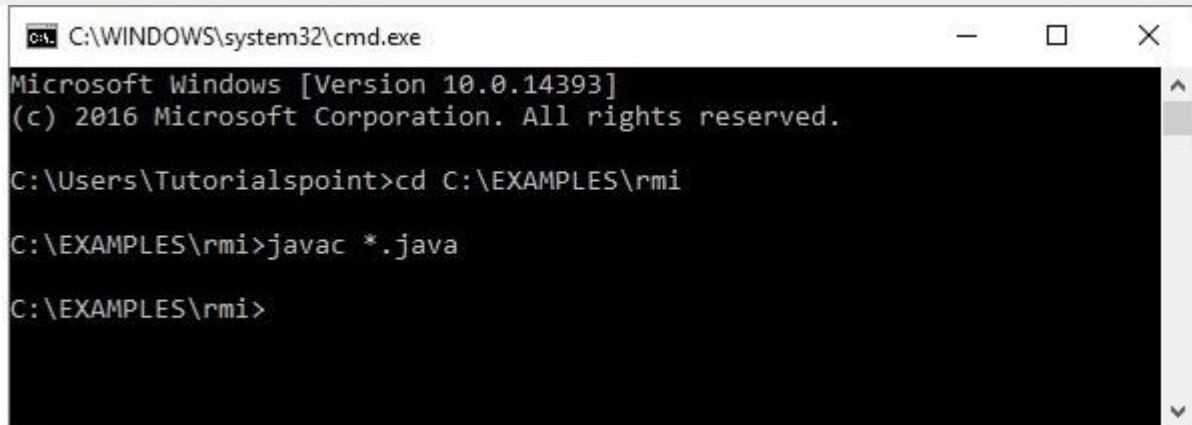
- Compile the Remote interface.
- Compile the implementation class.

- Compile the server program.
- Compile the client program.

Or,

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

Javac *.java



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

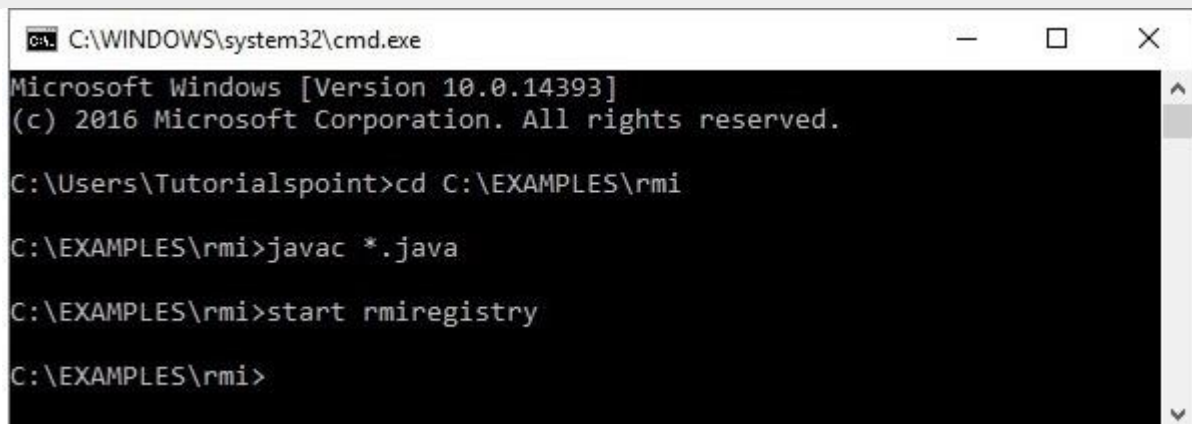
C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>
```

Executing the Application

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

start rmiregistry



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>
```

RMI Application

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

- Define the remote interface

- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

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;  
}
```

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");
    }
}
```