### What is Java Networking?

Networking supplements a lot of power to simple programs. With networks, a single program can regain information stored in millions of computers positioned anywhere in the world.

Java is the leading programming language composed from scratch with networking in mind.

Java Networking is a notion of combining two or more computing devices together to share resources.

All the Java program communications over the network are done at the application layer.

The **java.net**package of the J2SE APIs comprises various classes and interfaces that execute the low-level communication features, enabling the user to formulate programs that focus on resolving the problem.

### Common Network Protocols

As stated earlier,the **java.net**package of the Java programming language includes various classes and interfaces that provide an easy-to-use means to access network resources.

Other than classes and interfaces, the **java.net** package also provides support for the two well-known network protocols. These are:

1. **Transmission Control Protocol (TCP) –**TCP or Transmission Control Protocol allows secure communication between different applications. TCP is a connection-oriented protocol which means that once a connection is established, data can be transmitted in two directions. This protocol is typically used over the Internet Protocol. Therefore, TCP is also referred to as TCP/IP. TCP has built-in methods to examine for errors and ensure the delivery of data in the order it was sent, making it a complete protocol for transporting information like still images, data files, and web pages.
2. **User Datagram Protocol (UDP) –**UDP or User Datagram Protocol is a connection-less protocol that allows data packets to be transmitted between different applications. UDP is a simpler Internet protocol in which error-checking and recovery services are not required. In UDP, there is no overhead for opening a connection, maintaining a connection, or terminating a connection. In UDP, the data is continuously sent to the recipient, whether they receive it or not.

### Java Networking Terminology

In Java Networking, many terminologies are used frequently. These widely used Java Networking Terminologies are given as follows:

1. **IP Address –**An IP address is a unique address that distinguishes a device on the internet or a local network. IP stands for “Internet Protocol.” It comprises a set of rules governing the format of data sent via the internet or local network. IP Address is referred to as a logical address that can be modified. It is composed of octets. The range of each octet varies from 0 to 255.
   * Range of the IP Address – 0.0.0.0  to  255.255.255.255
   * For Example – 192.168.0.1
2. **Port Number –**A port number is a method to recognize a particular process connecting internet or other network information when it reaches a server. The port number is used to identify different applications uniquely. The port number behaves as a communication endpoint among applications. The port number is correlated with the IP address for transmission and communication among two applications. There are 65,535 port numbers, but not all are used every day.
3. **Protocol –**A network protocol is an organized set of commands that define how data is transmitted between different devices in the same network. Network protocols are the reason through which a user can easily communicate with people all over the world and thus play a critical role in modern digital communications. For Example – TCP, FTP, POP, etc.
4. **MAC Address –**MAC address stands for Media Access Control address. It is a bizarre identifier that is allocated to a NIC (Network Interface Controller/ Card). It contains a 48 bit or 64-bit address, which is combined with the network adapter. MAC address can be in hexadecimal composition. In simple words, a MAC address is a unique number that is used to track a device in a network.
5. **Socket –**A socket is one endpoint of a two-way communication connection between the two applications running on the network. The socket mechanism presents a method of inter-process communication (IPC) by setting named contact points between which the communication occurs. A socket is tied to a port number so that the TCP layer can recognize the application to which the data is intended to be sent.
6. **Connection-oriented and connection-less protocol –**In a connection-oriented service, the user must establish a connection before starting the communication. When the connection is established, the user can send the message or the information, and after this, they can release the connection. However, In connectionless protocol, the data is transported in one route from source to destination without verifying that the destination is still there or not or if it is ready to receive the message. Authentication is not needed in the connectionless protocol.
   * Example of Connection-oriented Protocol – Transmission Control Protocol (TCP)
   * Example of Connectionless Protocol – User Datagram Protocol (UDP)

### Java Networking classes

The **java.net**package of the Java programming language includes various classes that provide an easy-to-use means to access network resources. The classes covered in the **java.net** package are given as follows – 

1. [**CacheRequest –**](https://www.geeksforgeeks.org/java-net-cacherequest-class-in-java/)The CacheRequest class is used in java whenever there is a need to store resources in ResponseCache. The objects of this class provide an edge for the OutputStream object to store resource data into the cache.
2. [**CookieHandler**](https://www.geeksforgeeks.org/java-net-cookiehandler-class-in-java/)**–**The CookieHandler class is used in Java to implement a callback mechanism for securing up an HTTP state management policy implementation inside the HTTP protocol handler. The HTTP state management mechanism specifies the mechanism of how to make HTTP requests and responses.
3. [**CookieManager**](https://www.geeksforgeeks.org/java-net-cookiemanager-class-in-java/)**–** The CookieManager class is used to provide a precise implementation of CookieHandler. This class separates the storage of cookies from the policy surrounding accepting and rejecting cookies. A CookieManager comprises a CookieStore and a CookiePolicy.
4. [**DatagramPacket**](https://www.geeksforgeeks.org/java-net-datagrampacket-class-java/)**–**The DatagramPacket class is used to provide a facility for the connectionless transfer of messages from one system to another. This class provides tools for the production of datagram packets for connectionless transmission applying the datagram socket class.
5. [**InetAddress**](https://www.geeksforgeeks.org/java-net-inetaddress-class-in-java/)**–**The InetAddress class is used to provide methods to get the IP address of any hostname. An IP address is expressed by a 32-bit or 128-bit unsigned number. InetAddress can handle both IPv4 and IPv6 addresses.
6. [**Server Socket**](https://www.geeksforgeeks.org/java-net-serversocket-class-in-java/)**–**The ServerSocket class is used for implementing system-independent implementation of the server-side of a client/server Socket Connection. The constructor for ServerSocket class throws an exception if it can’t listen on the specified port. For example – it will throw an exception if the port is already being used.
7. [**Socket**](https://www.geeksforgeeks.org/java-net-socket-class-in-java/)**–**The Socket class is used to create socket objects that help the users in implementing all fundamental socket operations. The users can implement various networking actions such as sending, reading data, and closing connections. Each Socket object built using **java.net.Socket** class has been connected exactly with 1 remote host; for connecting to another host, a user must create a new socket object.
8. [**DatagramSocket**](https://www.geeksforgeeks.org/java-net-datagramsocket-class-java/)**–**The DatagramSocket class is a network socket that provides a connection-less point for sending and receiving packets. Every packet sent from a datagram socket is individually routed and delivered. It can further be practiced for transmitting and accepting broadcast information. Datagram Sockets is Java’s mechanism for providing network communication via UDP instead of TCP.
9. [**Proxy**](https://www.geeksforgeeks.org/java-net-proxy-class-in-java/)**–**A proxy is a changeless object and a kind of tool or method or program or system, which serves to preserve the data of its users and computers. It behaves like a wall between computers and internet users. A Proxy Object represents the Proxy settings to be applied with a connection.
10. [**URL**](https://www.geeksforgeeks.org/url-class-java-examples/)**–**The URL class in Java is the entry point to any available sources on the internet. A Class URL describes a Uniform Resource Locator, which is a signal to a “resource” on the World Wide Web. A source can denote a simple file or directory, or it can indicate a more difficult object, such as a query to a database or a search engine.
11. [**URLConnection**](https://www.geeksforgeeks.org/java-net-urlconnection-class-in-java/)**–**The URLConnection class in Java is an abstract class describing a connection of a resource as defined by a similar URL. The URLConnection class is used for assisting two distinct yet interrelated purposes. Firstly it provides control on interaction with a server(especially an HTTP server) than a URL class. Furthermore, with a URLConnection, a user can verify the header transferred by the server and can react consequently. A user can also configure header fields used in client requests using URLConnection.

### Java Networking Interfaces

The **java.net**package of the Java programming language includes various interfaces also that provide an easy-to-use means to access network resources. The interfaces included in the**java.net** package are as follows:

1. **CookiePolicy –**The CookiePolicy interface in the **java.net**package provides the classes for implementing various networking applications. It decides which cookies should be accepted and which should be rejected. In CookiePolicy, there are three pre-defined policy implementations, namely ACCEPT\_ALL, ACCEPT\_NONE, and ACCEPT\_ORIGINAL\_SERVER.
2. **CookieStore –**A CookieStore is an interface that describes a storage space for cookies. CookieManager combines the cookies to the CookieStore for each HTTP response and recovers cookies from the CookieStore for each HTTP request.
3. **FileNameMap –**The FileNameMap interface is an uncomplicated interface that implements a tool to outline a file name and a MIME type string. FileNameMap charges a filename map ( known as a mimetable) from a data file.
4. **SocketOption –**The SocketOption interface helps the users to control the behavior of sockets. Often, it is essential to develop necessary features in Sockets. SocketOptions allows the user to set various standard options.
5. **SocketImplFactory –**The SocketImplFactory interface defines a factory for SocketImpl instances. It is used by the socket class to create socket implementations that implement various policies.
6. **ProtocolFamily –**This interface represents a family of communication protocols. The ProtocolFamily interface contains a method known as name(), which returns the name of the protocol family.

### Socket Programming

[**Java Socket programming**](https://www.geeksforgeeks.org/socket-programming-in-java/) is practiced for communication between the applications working on different JRE. Sockets implement the communication tool between two computers using TCP. Java Socket programming can either be connection-oriented or connection-less. In Socket Programming, Socket and ServerSocket classes are managed for connection-oriented socket programming. However, DatagramSocket and DatagramPacket classes are utilized for connection-less socket programming.

A client application generates a socket on its end of the communication and strives to combine that socket with a server. When the connection is established, the server generates an object of socket class on its communication end. The client and the server can now communicate by writing to and reading from the socket.

The **java.net.Socket** class describes a socket, and the **java.net.ServerSocket** class implements a tool for the server program to host clients and build connections with them.  
   
**Steps to establishing a TCP connection between two computing devices using Socket Programming**

The following are the steps that occur on establishing a TCP connection between two computers using socket programming are given as follows:

**Step 1 –**The server instantiates a ServerSocket object, indicating at which port number communication will occur.

**Step 2 –**After instantiating the ServerSocket object, the server requests the accept() method of the ServerSocket class. This program pauses until a client connects to the server on the given port.

**Step 3 –**After the server is idling, a client instantiates an object of Socket class, defining the server name and the port number to connect to.

**Step 4 –**After the above step, the constructor of the Socket class strives to connect the client to the designated server and the port number. If communication is authenticated, the client forthwith has a Socket object proficient in interacting with the server.

**Step 5 –**On the server-side, the accept() method returns a reference to a new socket on the server connected to the client’s socket.

After the connections are stabilized, communication can happen using I/O streams. Each object of a socket class has both an OutputStream and an InputStream. The client’s OutputStream is correlated to the server’s InputStream, and the client’s InputStream is combined with the server’s OutputStream. Transmission Control Protocol (TCP) is a two-way communication protocol. Hence information can be transmitted over both streams at the corresponding time.

### Socket Class

The **Socket class** is used to create socket objects that help the users in implementing all fundamental socket operations. The users can implement various networking actions such as sending, reading data, and closing connections. Each Socket object created using **java.net.Socket** class has been correlated specifically with 1 remote host. If a user wants to connect to another host, then he must build a new socket object.

**Methods of Socket Class**

In Socket programming, both the client and the server have a Socket object, so all the methods under the Socket class can be invoked by both the client and the server. There are many methods in the Socket class.

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

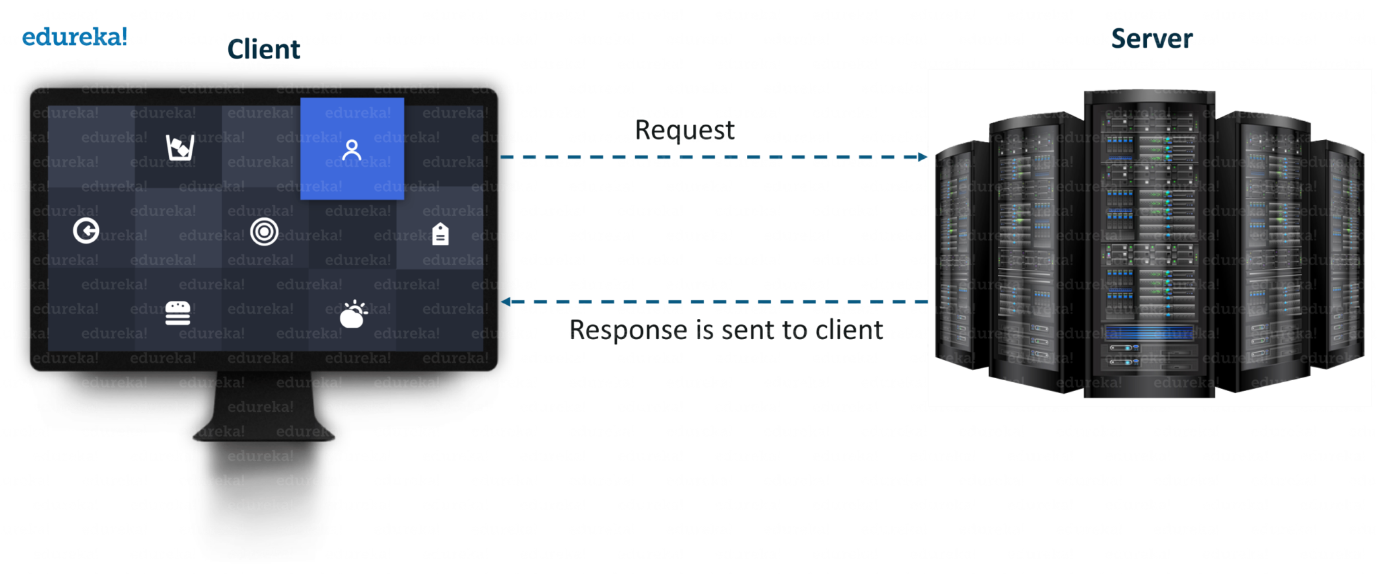
It can be either connection-oriented or connectionless.

On the whole, a socket is a way to establish a connection between a client and a server.

**What is Socket Programming in Java?**

*Socket programming* is a way of connecting two nodes on a network to communicate with each other.

One ***socket***(node) listens on a particular port at an IP, while other *socket*reaches out to the other in order to form a connection.

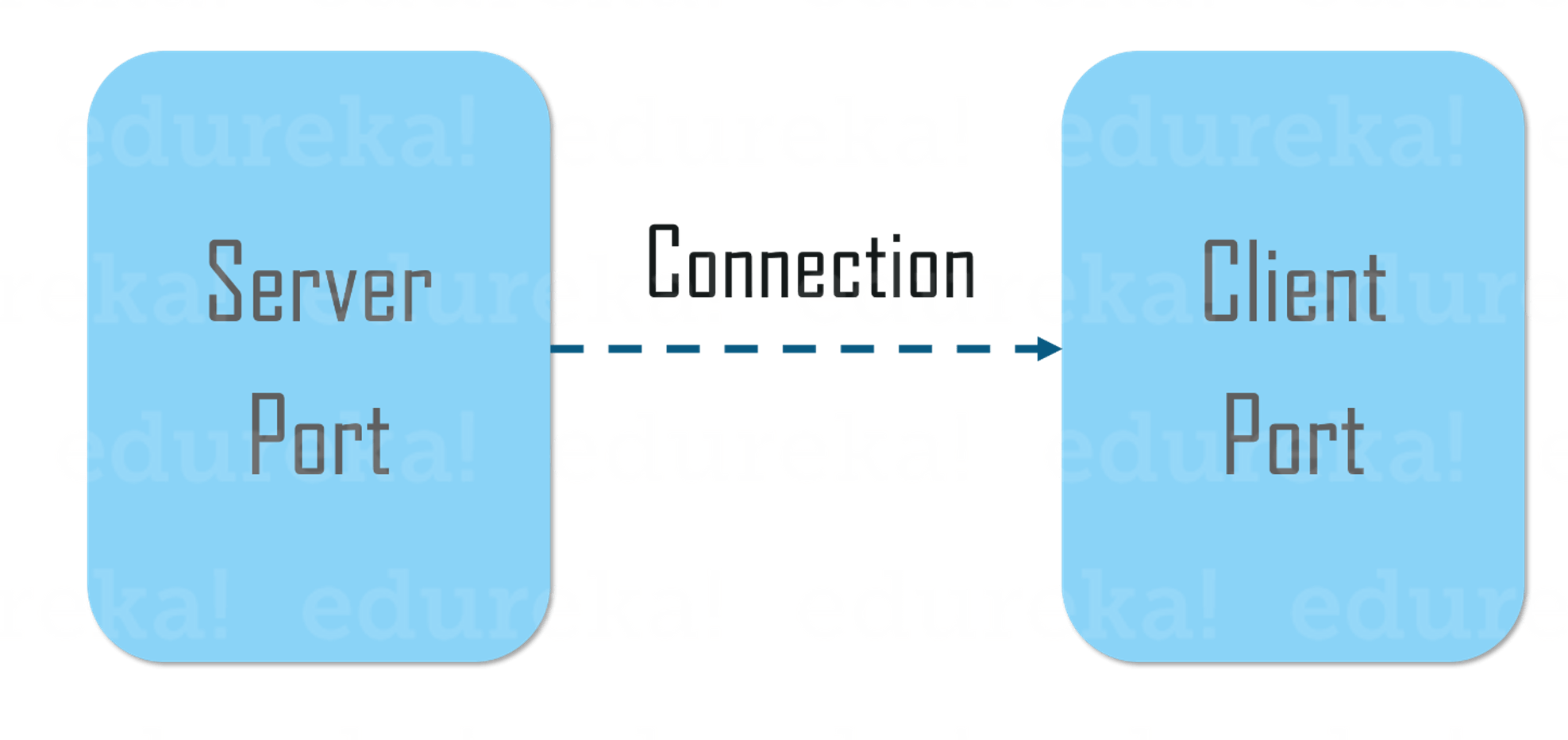


The server forms the listener *socket while* the client reaches out to the server. Socket and Server Socket [classes](https://www.edureka.co/blog/java-tutorial/#obj) are used for connection-oriented socket programming.

**What is a Socket in Java?**

A **socket**in [Java](https://www.edureka.co/blog/java-tutorial/) is one endpoint of a two-way communication link between two programs running on the network.

A **socket** is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.



An endpoint is a combination of an IP address and a port number.

The package in the Java platform provides a class, Socket that implements one side of a two-way connection between your Java program and another program on the network.

The class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program.

By using the class instead of relying on native code, your [Java programs](https://www.edureka.co/blog/java-programs/) can communicate over the network in a platform-independent fashion.

Now that you know, what is Socket in Java, let’s move further and understand how does client communicates with the server and how the server responds back.

**Client Side Programming**

In the case of client-side programming, the client will first wait for the server to start.

Once the server is up and running, it will send the requests to the server. After that, the client will wait for the response from the server.

So, this is the whole logic of client and server communication.

Now let’s understand the client side and server side programming in detail.

In order to initiate a clients request, you need to follow the below-mentioned steps:

**1. Establish a Connection**

[[](https://www.edureka.co/java-j2ee-training-course?utm_source=blogbanner&utm_campaign=curriculum)](https://www.edureka.co/java-j2ee-training-course?utm_source=blogbanner&utm_campaign=curriculum" \t "_blank)

**[Java Certification Training Course](https://www.edureka.co/java-j2ee-training-course?utm_source=blogbanner&utm_campaign=curriculum" \t "_blank)**

The very first step is to establish a socket connection.

A socket connection implies that the two machines have information about each other’s network location (IP Address) and TCP port.

You can create a Socket with the help of a below statement:

Socket socket = new Socket(“127.0.0.1”, 5000)

* Here, the first argument represents the **IP address of Server**.
* The second argument represents the **TCP Port**. (It is a number that represents which application should run on a server.)

**2. Communication**

In order to communicate over a socket connection, streams are used for both input and output the data.

After establishing a connection and sending the requests, you need to close the connection.

**3. Closing the connection**

The socket connection is closed explicitly once the message to the server is sent.

Now let’s see how to write a Java program to implement socket connection at client side.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | // A Java program for a ClientSide  **import** java.net.\*;  **import** java.io.\*;  **public** **class** ClientProgram  {  // initialize socket and input output streams  **private** Socket socket = **null**;  **private** DataInputStream input = **null**;  **private** DataOutputStream out = **null**;  // constructor to put ip address and port  **public** Client(String address, **int** port)  {  // establish a connection  **try**  {  socket = **new** Socket(address, port);  System.out.println("Connected");  // takes input from terminal  input = **new** DataInputStream(System.in);  // sends output to the socket  out = **new** DataOutputStream(socket.getOutputStream());  }  **catch**(UnknownHostException u)  {  System.out.println(u);  }  **catch**(IOException i)  {  System.out.println(i);  }// string to read message from input  String line = "";  // keep reading until "Over" is input  **while** (!line.equals("Over"))  {  **try**  {  line = input.readLine();  out.writeUTF(line);  }  **catch**(IOException i)  {  System.out.println(i);  }  }  // close the connection  **try**  {  input.close();  out.close();  socket.close();  }  **catch**(IOException i)  {  System.out.println(i);  }  }  **public** **static** **void** main(String args[]) {  Client client = **new** Client("127.0.0.1", 5000);  }  } |

Now, let’s implement server-side programming and then arrive at the output.

**Server Side Programming**

Basically, the server will instantiate its object and wait for the client request. Once the client sends the request, the server will communicate back with the response.

In order to code the server-side application, you need two sockets and they are as follows:

* A **ServerSocket** which waits for the client requests (when a client makes a new Socket())
* A plain old **socket** for communication with the client.

After this, you need to communicate with the client with the response.

**Communication**

**getOutputStream()** method is used to send the output through the socket.

**Close the Connection**

It is important to close the connection by closing the socket as well as input/output streams once everything is done.

Now let’s see how to write a Java program to implement socket connection at server side.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | // A Java program for a Serverside  **import** java.net.\*;  **import** java.io.\*;  **public** **class** ServerSide  {  //initialize socket and input stream  **private** Socket socket = **null**;  **private** ServerSocket server = **null**;  **private** DataInputStream in = **null**;  // constructor with port  **public** Server(**int** port)  {  // starts server and waits for a connection  **try**{  server = **new** ServerSocket(port);  System.out.println("Server started");  System.out.println("Waiting for a client ...");  socket = server.accept();  System.out.println("Client accepted");  // takes input from the client socket  in = **new** DataInputStream(  **new** BufferedInputStream(socket.getInputStream()));  String line = "";  // reads message from client until "Over" is sent  **while** (!line.equals("Over"))  {  **try**  {  line = in.readUTF();  System.out.println(line);        }  **catch**(IOException i)  {  System.out.println(i);  }  }  System.out.println("Closing connection");  // close connection  socket.close();  in.close();  }  **catch**(IOException i){  System.out.println(i);  }  }  **public** **static** **void** main(String args[]){  Server server = **new** Server(5000);  }  } |

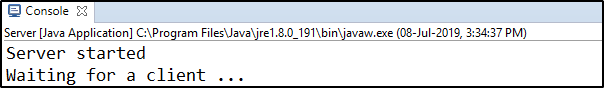
After configuring both client and server end, you can execute  the server side program first.

After that, you need to run client side program and send the request.

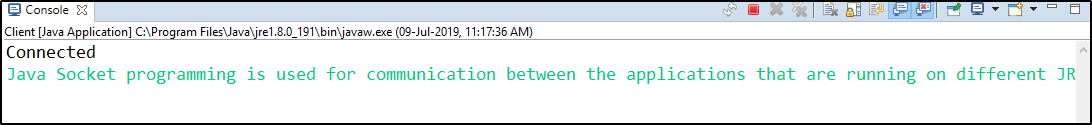
As soon as the request is sent from the client end, server will respond back.

Below snapshot represents the same.

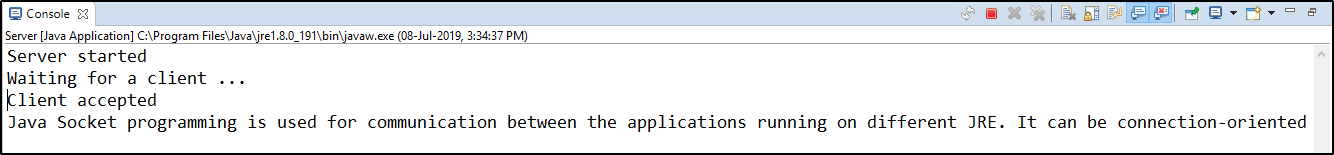
1. When you run the server side script, it will start and wait for the client to get started.



2. Next, the client will get connected and inputs the request in the form of a string.



3. When the client sends the request, the server will respond back.



# What is Proxy Server?

Proxy server refers to a server that acts as an intermediary between the request made by clients, and a particular server for some services or requests for some resources.

There are different types of proxy servers available that are put into use according to the purpose of a request made by the clients to the servers.

The basic purpose of Proxy servers is to protect the direct connection of Internet clients and internet resources.

The proxy server also prevents the identification of the client’s IP address when the client makes any request is made to any other servers.

* **Internet Client and Internet resources:**For internet clients, Proxy servers also act as a shield for an internal network against the request coming from a client to access the data stored on the server.

It makes the original IP address of the node remains hidden while accessing data from that server.

* **Protects true host identity:**In this method, outgoing traffic appears to come from the proxy server rather than internet navigation.

It must be configured to the specific application such as HTTPs or FTP.

For example, organizations can use a proxy to observe the traffic of its employees to get the work efficiently done.

It can also be used to keep a check on any kind of highly confidential data leakage. Some can also use it to increase their websites rank.

**Need Of Private Proxy:**

1. **Defeat Hackers:** To protect organizations data from malicious use, passwords are used and different architects are setup, but still, there may be a possibility that this information can be hacked in case the IP address is accessible easily. To prevent such kind of misuse of Data Proxy servers are set up to prevent tracking of original IP addresses instead data is shown to come from a different IP address.
2. **Filtering of Content:**By caching the content of the websites, Proxy helps in fast access to the data that has been accessed very often.
3. **Examine Packet headers and Payloads:**Payloads and packet headers of the requests made by the user nodes in the internal server to access to social websites can be easily tracked and restricted.
4. **To control internet usage of employees and children:**In this, the Proxy server is used to control and monitor how their employees or kids use the internet.

Organizations use it, to deny access to a specific website and instead redirecting you with a nice note asking you to refrain from looking at said sites on the company network.

1. **Bandwidth savings and improved speeds:**Proxy helps organizations to get better overall network performance with a good proxy server.
2. **Privacy Benefits:**Proxy servers are used to browse the internet more privately. It will change the IP address and identify the information the web request contains.
3. **Security:**Proxy server is used to encrypt your web requests to keep prying eyes from reading your transactions as it provides top-level security.

**Types Of Proxy Server**

1. **Reverse Proxy Server:**The job of a reverse proxy server to listen to the request made by the client and redirect to the particular web server which is present on different servers.  
   Example – Listen for TCP port 80 website connections which are normally placed in a demilitarized zone (DMZ) zone for publicly accessible services but it also protects the true identity of the host. Moreover, it is transparent to external users as external users will not be able to identify the actual number of internal servers.

So, it is the prime duty of reverse proxy to redirect the flow depending upon the configurations of internal servers.

The request that is made to pass through the private network protected by firewalls will need a proxy server that is not abiding by any of the local policies.

Such types of requests from the clients are completed using reverse proxy servers.

This is also used to restrict the access of the clients to the confidential data residing on the particular servers.

1. **Web Proxy Server:**Web Proxy forwards the HTTP requests, only URL is passed instead of a path.

The request is sent to particular the proxy server responds. Examples, Apache, HAP Proxy.

1. **Anonymous Proxy Server:**This type of proxy server does not make an original IP address instead these servers are detectable still provides rational anonymity to the client device.
2. **Highly Anonymity Proxy:**This proxy server does not allow the original IP address and it as a proxy server to be detected.
3. **Transparent Proxy:**This type of proxy server is unable to provide any anonymity to the client, instead, the original IP address can be easily detected using this proxy.

But it is put into use to act as a cache for the websites.

A transparent proxy when combined with gateway results in a proxy server where the connection requests are sent by the client , then IP are redirected.

Redirection will occurs without the client IP address configuration. HTTP headers present on the server-side can easily detect its redirection .

1. **CGI Proxy:**CGI proxy server developed to make the websites more accessible. It accepts the requests to target URLs using a web form and after processing its result will be returned to the web browser.

It is less popular due to some privacy policies like VPNs but it still receives a lot of requests also.

Its usage got reduced due to excessive traffic that can be caused to the website after passing the local filtration and thus leads to damage to the organization.

1. **Suffix Proxy:**Suffix proxy server basically appends the name of the proxy to the URL.

This type of proxy doesn’t preserve any higher level of anonymity. It is used for bypassing the web filters.

It is easy to use and can be easily implemented but is used less due to the more number of web filter present in it.

1. **Distorting Proxy:**Proxy servers are preferred to generate an incorrect original IP address of clients once being detected as a proxy server.

To maintain the confidentiality of the Client IP address HTTP headers are used.

1. **Tor Onion Proxy:**This server aims at online anonymity to the user’s personal information.

It is used to route the traffic through various networks present worldwide to arise difficulty in tracking the users’ address and prevent the attack of any anonymous activities.

It makes it difficult for any person who is trying to track the original address.

In this type of routing, the information is encrypted in a multi-folds layer.

At the destination, each layer is decrypted one by one to prevent the information to scramble and receive original content.

This software is open-source and free of cost to use.

1. **12P Anonymous Proxy:**It uses encryption to hide all the communications at various levels.

This encrypted data is then relayed through various network routers present at different locations and thus I2P is a fully distributed proxy. This software is free of cost and open source to use, It also resists the censorship.

1. **DNS Proxy:**DNS proxy take requests in the form of DNS queries and forward them to the Domain server where it can also be cached, moreover flow of request can also be redirected.

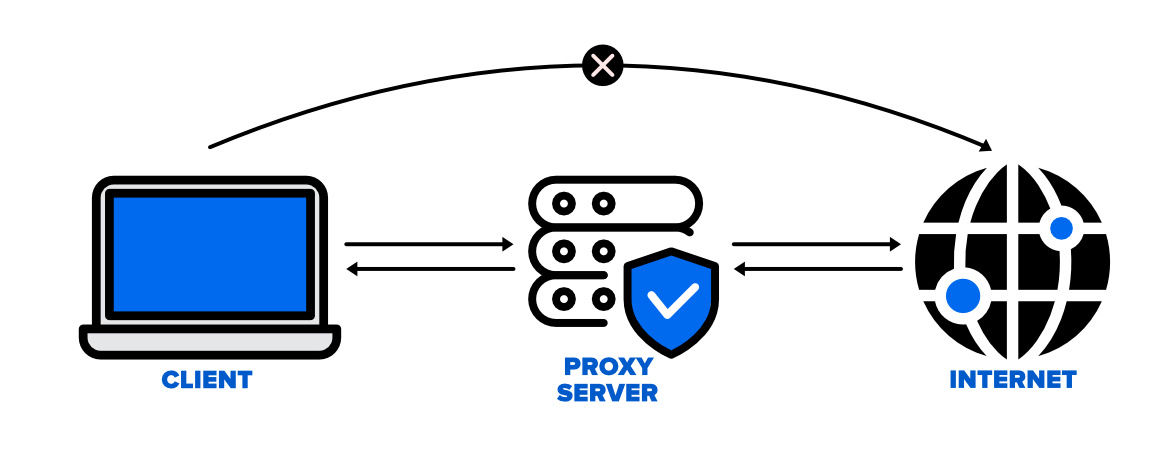
**How Does The Proxy Server Operates?**

Every computer has its unique IP address which it uses to communicate with another node.

Similarly, the proxy server has its IP address that your computer knows. When a web request is sent, your request goes to the proxy server first.

The Proxy sends a request on your behalf to the internet and then collect the data and make it available to you.

A proxy can change your IP address So, the webserver will be unable to fetch your location in the world.

It protects data from getting hacked too. Moreover, it can block some web pages also. 

**Disadvantages of Proxy Server**

1. **Proxy Server Risks:**Free installation does not invest much in backend hardware or encryption.

It will result in performance issues and potential data security issues. If you install a “free” proxy server, treat very carefully, some of those might steal your credit card numbers.

1. **Browsing history log:** The proxy server stores your original IP address and web request information is possibly unencrypted form and saved locally.

Always check if your proxy server logs and saves that data – and what kind of retention or law enforcement cooperation policies they follow while saving data.

1. **No encryption:**No encryption means you are sending your requests as plain text.

Anyone will be able to pull usernames and passwords and account information easily.

Keep a check that proxy provides full encryption whenever you use it.

# Creating an Server-Client Application using the DatagramPacket and DatagramSocket classes

To create an application that uses UDP to establish the connection between a client and server, we need to perform the following steps:

* Create a server program
* Create a client program
* Execute the client and server program

Let’s perform the steps in the following subsections:

**Creating the Server Program**

Let’s create the server class, named UDPServerEx which takes messages from a user and sends the messages (datagrams) to the clients. Listing 1 shows the code of the **UDPServerEx.java** file:

**Filename: UDPServerEx.java**

* Java

|  |
| --- |
| // A server that sends messages to the client    **import** java.net.\*;    **class** UDPServerEx {    **public** **static** DatagramSocket mySocket;  **public** **static** **byte** myBuffer[] = **new** **byte**[2000];    **public** **static** **void** serverMethod() **throws** Exception      {  **int** position = 0;  **while** (**true**) {  **int** charData = System.in.read();  **switch** (charData) {  **case** -1:                  System.out.println(                      "The execution of "                      + "the server has been terminated");  **return**;  **case** '\r':  **break**;  **case** '\n':                  mySocket.send(  **new** DatagramPacket(                          myBuffer,                          position,                          InetAddress.getLocalHost(),                          777));                  position = 0;  **break**;  **default**:                  myBuffer[position++]                      = (**byte**)charData;              }          }      }  **public** **static** **void** main(String args[]) **throws** Exception      {          System.out.println("Please enter some text here");          mySocket = **new** DatagramSocket(888);          serverMethod();      }  } |

**To compile the UDPServerEx.java file:**

D:\UDPExample>javac UDPServerEx.java

***Note: The path may vary according to where you save file.***

**Creating the ClientProgram**

Let’s create a client class, named UDPClient, which accepts the messages sent from the server, UDPServerEx class. The client then displays the messages received in the Command Prompt. Listing 2 shows the code of the **UDPClient.java** file:

**Filename: UDPClient.java**

* Java

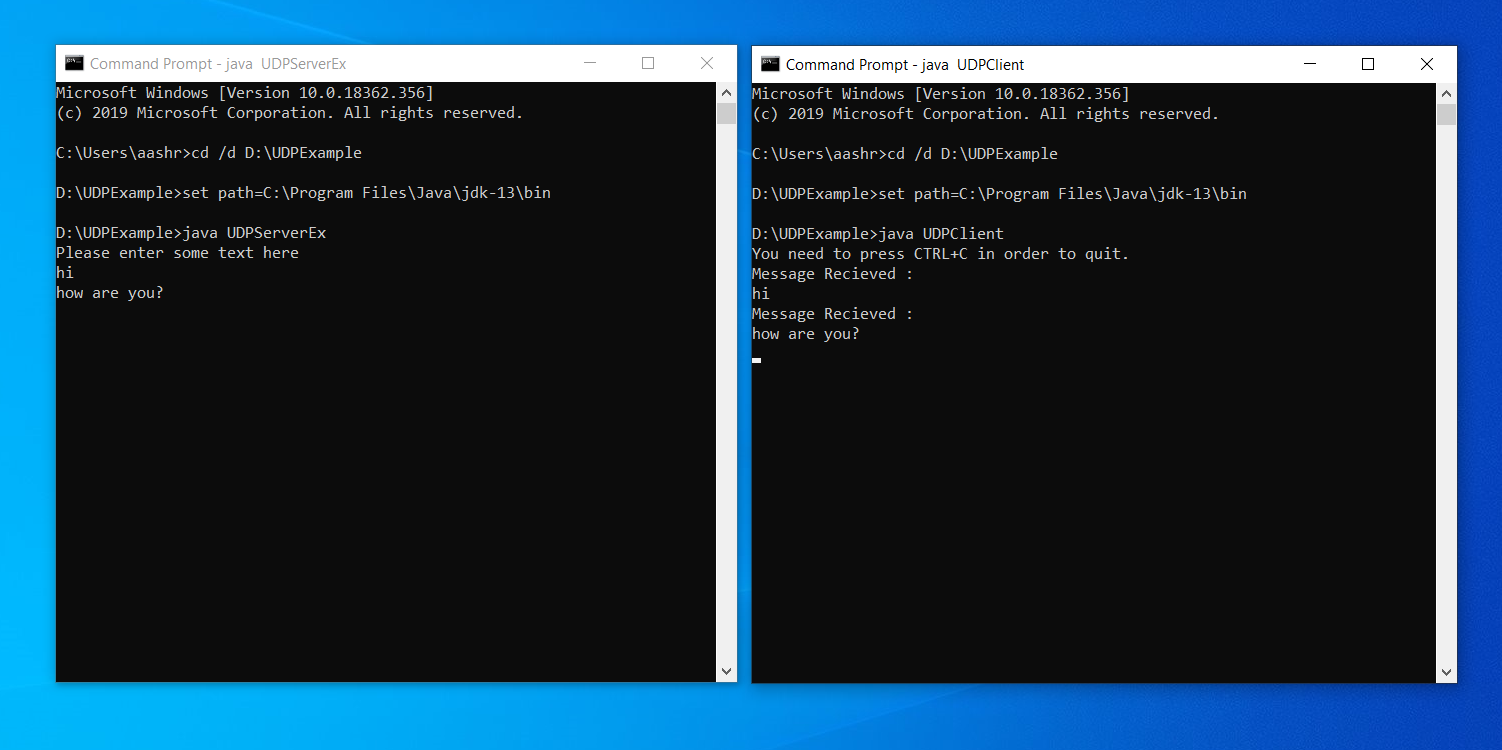
|  |
| --- |
| // UDPClient that receives and  // displays messages sent from the server    **import** java.net.\*;  **class** UDPClient {    **public** **static** DatagramSocket mySocket;  **public** **static** **byte** myBuffer[] = **new** **byte**[2000];    **public** **static** **void** clientMethod() **throws** Exception      {  **while** (**true**) {              DatagramPacket dataPacket                  = **new** DatagramPacket(myBuffer,                                       myBuffer.length);              mySocket.receive(dataPacket);              System.out.println("Message Received :");              System.out.println(  **new** String(                      dataPacket.getData(),                      0,                      dataPacket.getLength()));          }      }  **public** **static** **void** main(String args[]) **throws** Exception      {          System.out.println(              "You need to press CTRL+C"              + " in order to quit.");          mySocket = **new** DatagramSocket(777);          clientMethod();      }  } |

**Use the following command to compile the UDPClient.java file:**

D:\UDPExample>javac UDPClient.java

**Output**

**Note:** To execute the UDPServerEx and UDPClient classes, run the UDPServerEx.java and UDPClient.java in two separate Command Prompt windows. Remember, the UDPServerEx class is executed before the UDPClient class. Figure 1 shows the output of the UDP Server java and UDPClient.java files:

[](https://media.geeksforgeeks.org/wp-content/uploads/20190922165217/blog-1.11.png)

*Showing the Output of the UDPServerEx and UDPClient Classes*

# java.net.URLConnection Class in Java

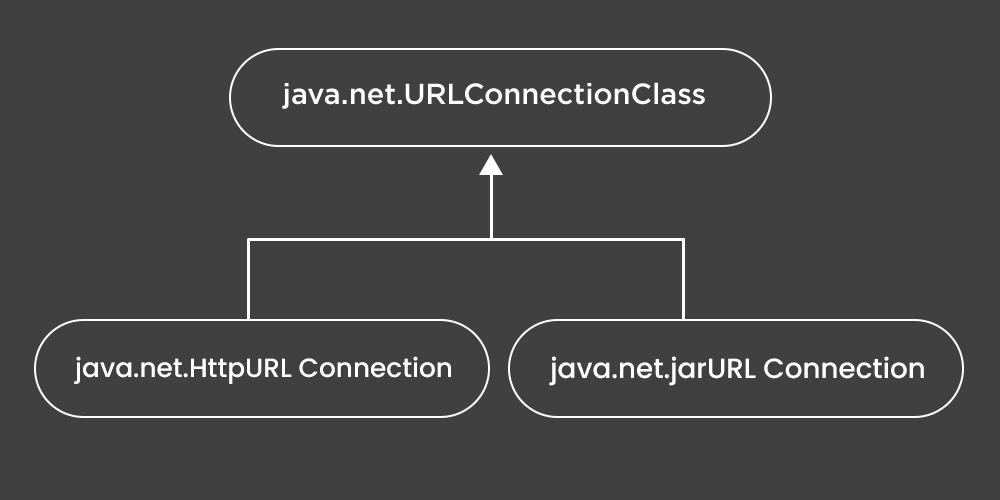
**URLConnection Class** in Java is an abstract class that represents a connection of a resource as specified by the corresponding URL.

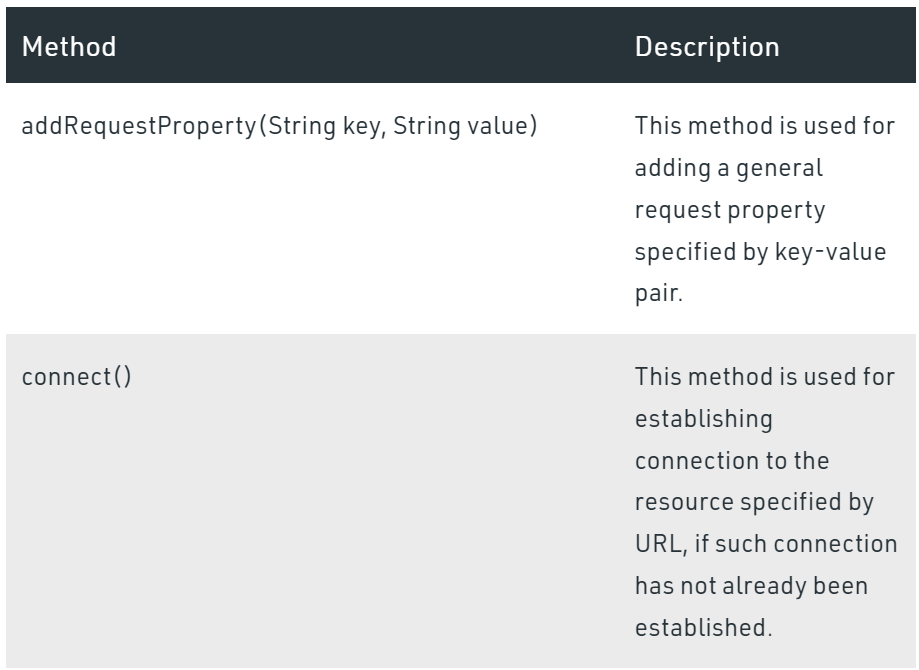
It is imported by the *java.net* package.

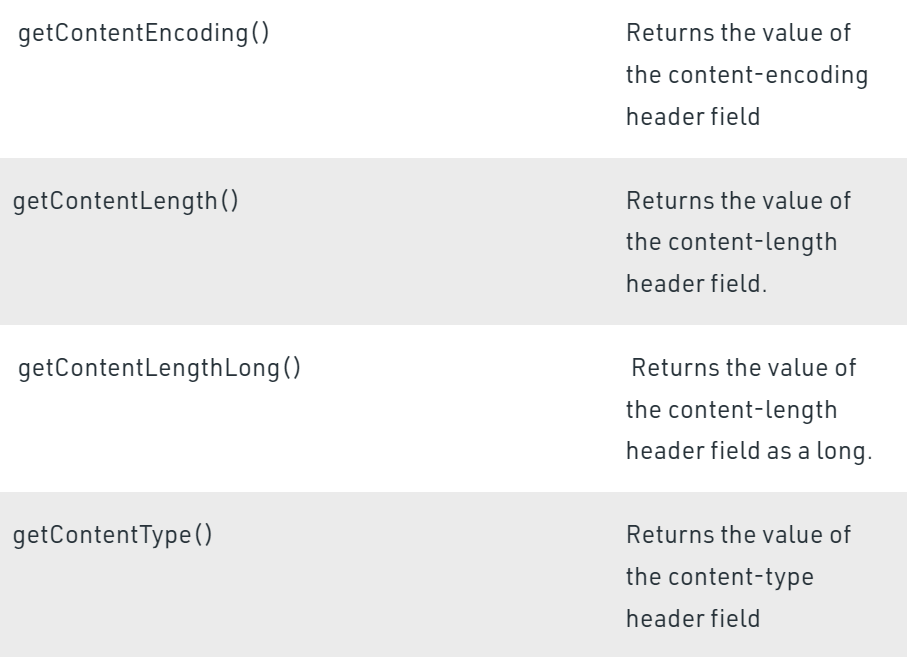
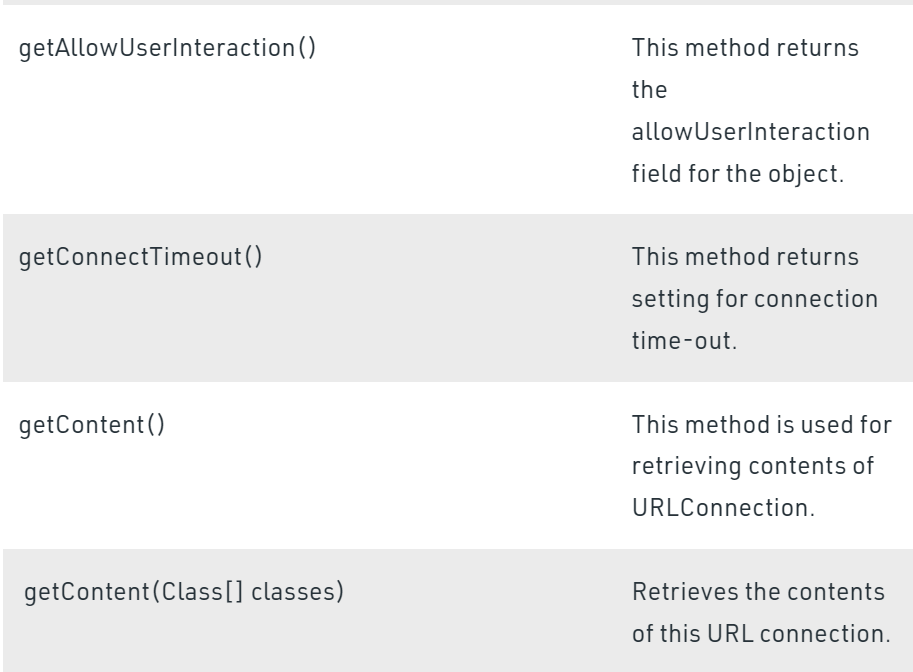
The [URLConnection class](https://www.geeksforgeeks.org/reading-url-using-urlconnection-class/) is utilized for serving two different yet related purposes, Firstly it provides control on interaction with a server(especially an HTTP server) than URL class.

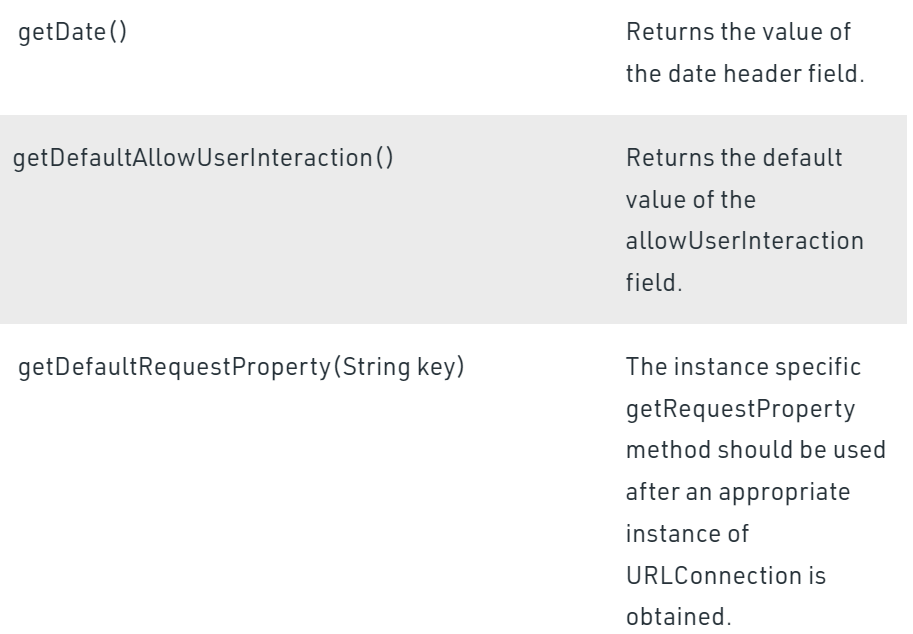
Secondly, with a URLConnection we can check the header sent by the server and respond accordingly, we can configure header fields used in client requests.

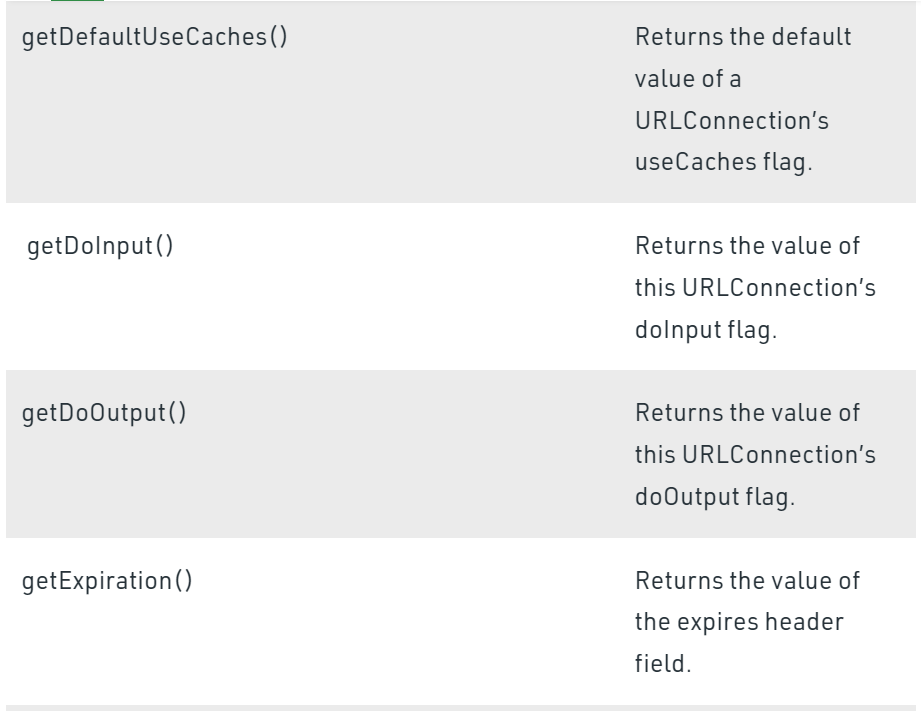
We can also download binary files by using URLConnection.











**Example**

* Java

|  |
| --- |
| // Java Program to demonstrate URLConnection class    // Importing input output classes  **import** java.io.\*;  // Importing java.net package  // consisting of all network classes  **import** java.net.\*;    // Main class  // URLConnectionExample  **public** **class** demo {        // Main driver method  **public** **static** **void** main(String[] args) **throws** Exception      {          // Try block to check for exceptions  **try** {                // Creating an object of URL class                // Custom input URL is passed as an argument              URL u = **new** URL("www.mitwpu.com");                // Creating an object of URLConnection class to              // communicate between application and URL              URLConnection urlconnect = u.openConnection();                // Creating an object of InputStream class              // for our application streams to be read              InputStream stream                  = urlconnect.getInputStream();                // Declaring an integer variable  **int** i;                // Till the time URL is being read  **while** ((i = stream.read()) != -1) {                    // Continue printing the stream                  System.out.print((**char**)i);              }          }            // Catch block to handle the exception  **catch** (Exception e) {                // Print the exception on the console              System.out.println(e);          }      }  } |

**Output**

java.net.MalformedURLException: no protocol: www.mitwpu.com

# Introducing Threads in Socket Programming in Java

**Why to use threads in network programming?**

The reason is simple, we don’t want only a single client to connect to server at a particular time but many clients simultaneously.

We want our architecture to **support multiple clients at the same time**.

For this reason, we must use threads on server side so that whenever a client request comes, a separate thread can be assigned for handling each request.

Let us take an example, suppose a Date-Time server is located at a place, say X.

Being a generic server, it does not serve any particular client, rather to a whole set of generic clients.

Also suppose at a particular time, two requests arrives at the server. With our basic server-client program, the request which comes even a nano-second first would be able to connect to the server and the other request would be rejected as no mechanism is provided for handling multiple requests simultaneously.

To overcome this problem, we use threading in network programming.

We shall see  **creating a simple Date-Time server for handling multiple client requests** at the same time.

**Quick Overview**

As normal, we will create two java files,**Server.java** and **Client.java**. Server file contains two classes namely **Server** (public class for creating server) and **ClientHandler** (for handling any client using multithreading).

Client file contain only one public class **Client** (for creating a client). Below is the flow diagram of how these three classes interact with each other.

Date-time-server-1

**Server Side Programming(Server.java)**

* **Server class :** The steps involved on server side are similar to the article [Socket Programming in Java](https://www.geeksforgeeks.org/socket-programming-in-java/) with a slight change to create the thread object after obtaining the streams and port number.
  1. **Establishing the Connection:** Server socket object is initialized and inside a while loop a socket object continuously accepts incoming connection.
  2. **Obtaining the Streams:** The inputstream object and outputstream object is extracted from the current requests’ socket object.
  3. **Creating a handler object:** After obtaining the streams and port number, a new clientHandler object (the above class) is created with these parameters.
  4. **Invoking the**[start()](https://www.geeksforgeeks.org/start-function-multithreading-java/)**method :** The start() method is invoked on this newly created thread object.
* **ClientHandler class :** As we will be using separate threads for each request, lets understand the working and implementation of the ClientHandler class extending Threads. An object of this class will be instantiated each time a request comes.
  1. First of all this class extends [Thread](https://www.geeksforgeeks.org/java-lang-thread-class-java/) so that its objects assumes all properties of Threads.
  2. Secondly, the constructor of this class takes three parameters, which can uniquely identify any incoming request, i.e. a **Socket**, a [DataInputStream](https://www.geeksforgeeks.org/java-io-datainputstream-class-java-set-1/) to read from and a [DataOutputStream](https://www.geeksforgeeks.org/dataoutputstream-in-java/)to write to. Whenever we receive any request of client, the server extracts its port number, the DataInputStream object and DataOutputStream object and creates a new thread object of this class and invokes [start()](https://www.geeksforgeeks.org/start-function-multithreading-java/) method on it.  
     Note : Every request will always have a triplet of socket, input stream and output stream. This ensures that each object of this class writes on one specific stream rather than on multiple streams.
  3. Inside the **run()** method of this class, it performs three operations: request the user to specify whether time or date needed, read the answer from input stream object and accordingly write the output on the output stream object.

**Client Side Programming (Client.java)**

Client side programming is similar as in general socket programming program with the following steps-

1. **Establish a Socket Connection**
2. **Communication**

**How these programs works together?**

1. When a client, say client1 sends a request to connect to server, the server assigns a new thread to handle this request. The newly assigned thread is given the access to streams for communicating with the client.
2. After assigning the new thread, the server via its while loop, again comes into accepting state.
3. When a second request comes while first is still in process, the server accepts this requests and again assigns a new thread for processing it. In this way, multiple requests can be handled even when some requests are in process.

**How to test the above program on your system?**

Save the two programs in same package or anywhere. Then first run the Server.java followed by the Client.java. You can either copy the client program in two three separate files and run them individually, or if you have an IDE like eclipse, run multiple instances from the same program. The output shown above is from a single client program, the similar results will be achieved if multiple clients are used.

# Multi-threaded chat Application in Java

A simple date time server was created which handled multiple user requests at the same time using threading. It explains the basic concepts of threading in network programming. The same concepts can be used with very slight modification to extend the above idea and create a chatting application similar to facebook messenger, whatsapp, etc.

**Server Side Programming(Server.java)**

**1. Server class :**The main server implementation is easy and similar to the previous article. The following points will help understand Server implementation :

1. The server runs an infinite loop to keep accepting incoming requests.
2. When a request comes, it assigns a new thread to handle the communication part.
3. The server also stores the client name into a vector, to keep a track of connected devices. The vector stores the thread object corresponding to the current request. The helper class uses this [vector](https://www.geeksforgeeks.org/java-util-vector-class-java/) to find the name of recipient to which message is to be delivered. As this vector holds all the streams, handler class can use it to successfully deliver messages to specific clients.
4. Invoke the [start()](https://www.geeksforgeeks.org/start-function-multithreading-java/) method.

**2. ClientHandler class :**Similar to previous article, we create a helper class for handling various requests. This time, along with the socket and streams, we introduce a name variable. This will hold the name of the client that is connected to the server. The following points will help understand ClientHandler implementation :

* Whenever the handler receives any string, it breaks it into the message and recipient part. It uses Stringtokenizer for this purpose with ‘#’ as the delimiter. Here it is assumed that the string is always of the format:

message **#** recipient

* It then searches for the name of recipient in the connected clients list, stored as a vector in the server. If it finds the recipients name in the clients list, it forwards the message on its output stream with the name of the sender prefixed to the message.

**Limitations:**  
Although the above implementation of server manages to handle most of the scenarios, there are some shortcomings in the approach defined above.

* One clear observation from above programs is that **if the number of clients grew large, the searching time would increase** in the handler class. To avoid this increase, two hash maps can be used. One with name as the key, and index in active list as the value. Another with index as key, and associated handler object as value. This way, we can quickly look up the two hashmaps for matching recipient. It is left to the readers to implement this hack to increase efficiency of the implementation.
* Another thing to notice is that this implementation **doesn’t work well when users disconnect from the server**. A lot of errors would be thrown because disconnection is not handled in this implementation. It can easily be implemented as in previous basic TCP examples. It is also left for the reader to implement this feature in the program.

// Java implementation of  Server side

// It contains two classes : Server and ClientHandler

// Save file as Server.java

**import** java.io.\*;

**import** java.util.\*;

**import** java.net.\*;

// Server class

**public** **class** Server

{

    // Vector to store active clients

**static** Vector<ClientHandler> ar = **new** Vector<>();

    // counter for clients

**static** **int** i = 0;

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

    {

        // server is listening on port 1234

        ServerSocket ss = **new** ServerSocket(1234);

        Socket s;

        // running infinite loop for getting

        // client request

**while** (**true**)

        {

            // Accept the incoming request

            s = ss.accept();

            System.out.println("New client request received : " + s);

            // obtain input and output streams

            DataInputStream dis = **new** DataInputStream(s.getInputStream());

            DataOutputStream dos = **new** DataOutputStream(s.getOutputStream());

            System.out.println("Creating a new handler for this client...");

            // Create a new handler object for handling this request.

            ClientHandler mtch = **new** ClientHandler(s,"client " + i, dis, dos);

            // Create a new Thread with this object.

            Thread t = **new** Thread(mtch);

            System.out.println("Adding this client to active client list");

            // add this client to active clients list

            ar.add(mtch);

            // start the thread.

            t.start();

            // increment i for new client.

            // i is used for naming only, and can be replaced

            // by any naming scheme

            i++;

        }

    }

}

// ClientHandler class

**class** ClientHandler **implements** Runnable

{

    Scanner scn = **new** Scanner(System.in);

**private** String name;

**final** DataInputStream dis;

**final** DataOutputStream dos;

    Socket s;

**boolean** isloggedin;

    // constructor

**public** ClientHandler(Socket s, String name,

                            DataInputStream dis, DataOutputStream dos) {

**this**.dis = dis;

**this**.dos = dos;

**this**.name = name;

**this**.s = s;

**this**.isloggedin=**true**;

    }

    @Override

**public** **void** run() {

        String received;

**while** (**true**)

        {

**try**

            {

                // receive the string

                received = dis.readUTF();

                System.out.println(received);

**if**(received.equals("logout")){

**this**.isloggedin=**false**;

**this**.s.close();

**break**;

                }

                // break the string into message and recipient part

                StringTokenizer st = **new** StringTokenizer(received, "#");

                String MsgToSend = st.nextToken();

                String recipient = st.nextToken();

                // search for the recipient in the connected devices list.

                // ar is the vector storing client of active users

**for** (ClientHandler mc : Server.ar)

                {

                    // if the recipient is found, write on its

                    // output stream

**if** (mc.name.equals(recipient) && mc.isloggedin==**true**)

                    {

                        mc.dos.writeUTF(**this**.name+" : "+MsgToSend);

**break**;

                    }

                }

            } **catch** (IOException e) {

                e.printStackTrace();

            }

        }

**try**

        {

            // closing resources

**this**.dis.close();

**this**.dos.close();

        }**catch**(IOException e){

            e.printStackTrace();

        }

    }

}

Output:

New client request received : Socket[addr=/127.0.0.1,port=61818,localport=1234]

Creating a new handler for this client...

Adding this client to active client list

New client request received : Socket[addr=/127.0.0.1,port=61819,localport=1234]

Creating a new handler for this client...

Adding this client to active client list

Implementation of client program for the multi-threaded chat application. Till now all examples in socket programming assume that client first sends some information and then server or other clients responds to that information.  
In real world, this might not be the case. It is not required to send someone a message in order to be able to receive one. A client should readily receive a message whenever it is delivered to it i.e sending and receiving must be **implemented as separate activities rather than sequential**.  
There is a very simple solution which uses threads to achieve this functionality. In the client side implementation we will be creating two threads:

1. **SendMessage :** This thread will be used for sending the message to other clients. The working is very simple, it takes input the message to send and the recipient to deliver to. Note that this implementation assumes the message to be of the format **message # recipient**, where recipient is the name of the recipient. It then writes the message on its output stream which is connected to the handler for this client. The handler breaks the message and recipient part and deliver to particular recipient. Lets look at how this thread can be implemented.

|  |
| --- |
| Thread sendMessage = **new** Thread(**new** Runnable() {              @Override  **public** **void** run() {  **while** (**true**) {                        // read the message to deliver.                      String msg = sc.nextLine();  **try** {                            // write on the output stream                          dos.writeUTF(msg);                      } **catch** (IOException e) {                          e.printStackTrace();                      }                  }              }          }); |

1. **readMessage :** A similar approach is taken for creating a thread for receiving the messages. When any client tries to write on this clients input stream, we use readUTF() method to read that message. The following snippet of how this thread is implemented is shown below-

|  |
| --- |
| Thread readMessage = **new** Thread(**new** Runnable() {                @Override  **public** **void** run() {    **while** (**true**) {  **try** {                            // read the message sent to this client                          String msg = dis.readUTF();                          System.out.println(msg);                      } **catch** (IOException e) {                            e.printStackTrace();                      }                  }              }          }); |

The remaining steps of client side programming are similar to previous examples. A brief explanation is as follows –

1. **Establish a Socket Connection**
2. **Communication**  
   Communication occurs with the help of the readMessage and sendMessage threads. Separate threads for reading and writing ensures simultaneous sending and receiving of messages.

|  |
| --- |
| // Java implementation for multithreaded chat client  // Save file as Client.java    **import** java.io.\*;  **import** java.net.\*;  **import** java.util.Scanner;    **public** **class** Client  {  **final** **static** **int** ServerPort = 1234;    **public** **static** **void** main(String args[]) **throws** UnknownHostException, IOException      {          Scanner scn = **new** Scanner(System.in);            // getting localhost ip          InetAddress ip = InetAddress.getByName("localhost");            // establish the connection          Socket s = **new** Socket(ip, ServerPort);            // obtaining input and out streams          DataInputStream dis = **new** DataInputStream(s.getInputStream());          DataOutputStream dos = **new** DataOutputStream(s.getOutputStream());            // sendMessage thread          Thread sendMessage = **new** Thread(**new** Runnable()          {              @Override  **public** **void** run() {  **while** (**true**) {                        // read the message to deliver.                      String msg = scn.nextLine();    **try** {                          // write on the output stream                          dos.writeUTF(msg);                      } **catch** (IOException e) {                          e.printStackTrace();                      }                  }              }          });            // readMessage thread          Thread readMessage = **new** Thread(**new** Runnable()          {              @Override  **public** **void** run() {    **while** (**true**) {  **try** {                          // read the message sent to this client                          String msg = dis.readUTF();                          System.out.println(msg);                      } **catch** (IOException e) {                            e.printStackTrace();                      }                  }              }          });            sendMessage.start();          readMessage.start();        }  } |

**Output :**  
**From client 0 :**

hello#client 1

client 1 : heya

how are you#client 1

client 1 : fine..how about you

logout

**From client 1 :**

client 0 : hello

heya#client 0

client 0 : how are you

fine..how about you#client 0

logout

**Important points :**

* To send a message from any client, type the message, followed by a “#” and then the name of the recipient client. Please note that this implementation gives names as “client 0”, “client 1″….”client n” and so carefully names must be appended int the end. After that press Enter key.
* Once a message is sent, the handler for this client will receive the message and it will be delivered to the specified client.
* If any client sends a message to this client, the readMessage thread will automatically print the message on the console.
* Once a client is done with chatting, he can send a “logout” message without any recipient name so that server would know that this client has logged off the system. It is recommended to send a logout message before closing the terminal for the client to avoid any errors.

**How to run the above program ?**

Similar to previous examples, first run the server and then run multiple instances of the client. From each of the client, try sending message to each other. Please make sure you send message to only a valid client, i.e. to the client available on active list.

**Suggested Improvements**

This was only the explanation part as to how threads and socket programming can be used to create powerful programs. There are some suggested improvements to above implementations for the interested readers-

* Create a graphical user interface for clients for sending and receiving messages. A tool such as Netbeans can be used to quickly design an interface
* Currently the names are hard-coded as client 0, client 1. This can be improved to use user given nicknames.
* This implementation can be further enhanced to provide client the list of current active users so that he can know who all of his friends are online. A simple method can be implemented for this purpose which when invoked prints the names in active list.