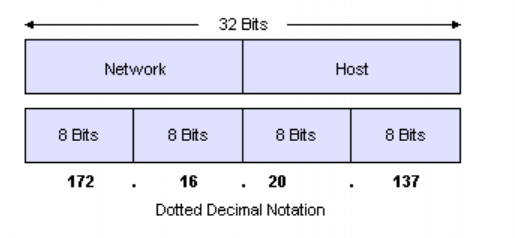
**Networking: Internet Addressing, InetAddress, Factory Methods, Instance Methods, TCP /IP Client Sockets, URL, URL Connection, TCP/IP Server Sockets, Datagrams.**

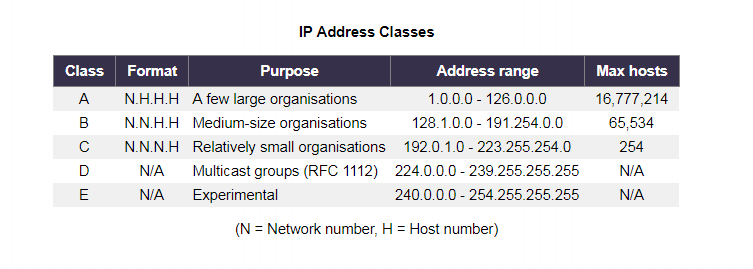
**Internet Addressing**

Each host on a TCP/IP network is assigned a unique 32-bit IP address consisting of a network number and a host number. The network number identifies a specific network, and must be assigned by the Internet Network Information Center (InterNIC) or an accredited registrar. An Internet Service Provider (ISP) can obtain blocks of network addresses from InterNIC and can assign addresses as necessary. The host number identifies a host on a network and is assigned by the network administrator. The IP address is grouped into four binary octets (an octet is a group of eight bits) and is represented using dotted decimal notation. The minimum value for an octet is 0, and the maximum value is 255. The basic format is illustrated below.

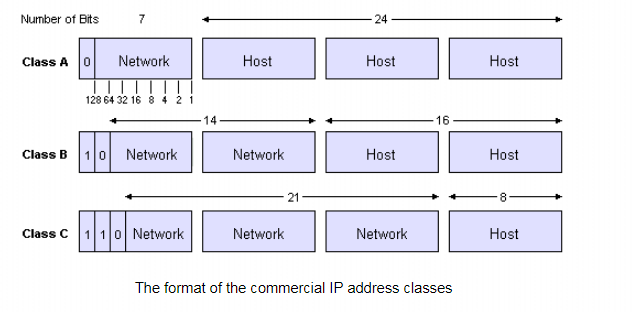


**The basic format of an IP address**

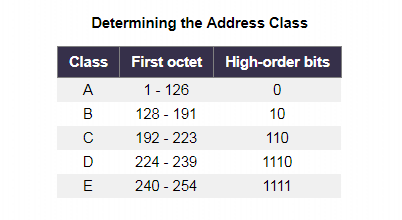
IP addressing supports five address classes - A, B, C, D, and E, of which only classes A, B, and C are available for commercial use. The following table illustrates the IP address classes.

****

The diagram below illustrates the format of the commercial IP address classes.

****

The address class can be determined using the following table. For example, IP address 172.31.1.2 has a first octet of 172, which falls between 128 and 191, so 172.31.1.2 is a Class B address.

****

Note that addresses beginning with 127 are used by some systems for special purposes. Note also that addresses with all zeros in the host part of the address refer to the network itself, and addresses ending in 255 are not used to refer to individual hosts on the network, since they are considered to be broadcast addresses.  
  
**IPv6 addresses** are typically composed of a 64-bit network prefix, and a 64-bit host part. The host part may be automatically generated from the interface's MAC address, or assigned sequentially. IPv6 addresses are normally written as eight groups of four hexadecimal numbers. A group consisting solely of zeros can be omitted. For example, 2001:0db8:85a3:0000:1319:8a2e:0370:1337 can be shortened to 2001:0db8:85a3::1319:8a2e:0370:1337. Note, however, that there must be only one double colon in an address. Leading zeros in a group can also be omitted, so the addresses below are all valid and equivalent to each other:

2001:0db8:0000:0000:0000:0000:1428:57ab  
2001:0db8:0000:0000:0000::1428:57ab  
2001:0db8:0:0:0:0:1428:57ab  
2001:0db8:0:0::1428:57ab  
2001:0db8::1428:57ab  
2001:db8::1428:57ab

**InetAddress**

An Internet Protocol **address** (**IP address**) is a logical numeric **address** that is assigned to every single computer, printer, switch, router or any other device that is part of a TCP/**IP**-based network. ... An **IP address** is a logical **address** that is used to uniquely identify every node in the network.

**Factory Method Pattern**

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

Advantage of Factory Design Pattern

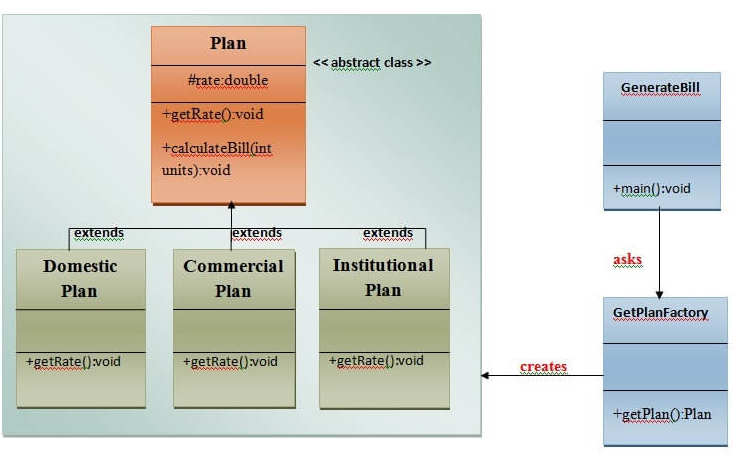
* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Usage of Factory Design Pattern

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

UML for Factory Method Pattern

* We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.
* GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.

****

**Instance method**

Instance method are methods which require an object of its class to be created before it can be called. To invoke a instance method, we have to create an Object of the class in within which it defined.

public void geek(String name)

{

// code to be executed....

}

// Return type can be int, float String or user defined data type.

* Instance method(s) belong to the Object of the class not to the class i.e. they can be called after creating the Object of the class.
* Every individual Object created from the class has its own copy of the instance method(s) of that class.
* They can be overridden since they are resolved using **dynamic binding** at run time.

**TCP /IP Client Sockets**

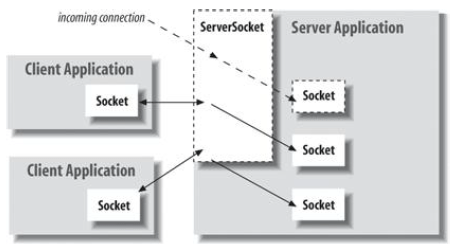
TCP/IP sockets are used to implement t reliable, bidirectional, persistent, point-to-point, and stream -based connections between hosts on the Internet. A socket can be used to connect Java’s I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

## TCP/IP Client and Server Sockets in Java

***Clients and servers, Sockets and Server Sockets***

Applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The Server Socket class is designed to be a listener, which waits for clients to connect before doing anything. The Socket class is designed to connect to server sockets and initiate protocol exchanges.

The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:



Socket Programming in Java

This article describes a very basic one-way Client and Server setup where a Client connects, sends messages to server and the server shows them using socket connection. There’s a lot of low-level stuff that needs to happen for these things to work but the Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers.

**The steps involved in establishing a socket on the server side are as follows:**

1. **Create a socket with the socket() system call.**
2. **Bind the socket to an address using the bind() system call. ...**
3. **Listen for connections with the listen() system call.**
4. **Accept a connection with the accept() system call. ...**
5. **Send and receive data.**

**Datagrams in Java**

**Client Side Programming**

**Establish a Socket Connection**

To connect to other machine we need a socket connection. A socket connection means the two machines have information about each other’s network location (IP Address) and TCP port.The java.net.Socket class represents a Socket. To open a socket:

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

* First argument – **IP address of Server**. ( 127.0.0.1  is the IP address of localhost, where code will run on single stand-alone machine).
* Second argument – **TCP Port**. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

**Communication**

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

**Closing the connection**

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

In the program, Client keeps reading input from user and sends to the server until “Over” is typed.

**Server Programming**

**Establish a Socket Connection**

To write a server application two sockets are needed.

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

**Communication**

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

**Close the Connection**

After finishing,  it is important to close the connection by closing the socket as well as input/output streams.

// A Java program for a Client

import java.net.\*;

import java.io.\*;

public class Client

{

    // 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);

    }

}

// A Java program for a Server

import java.net.\*;

import java.io.\*;

public class Server

{

    //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);

    }

}

**Important Points**

* Server application makes a ServerSocket on a specific port which is 5000. This starts our Server listening for client requests coming in for port 5000.
* Then Server makes a new Socket to communicate with the client.

**socket = server.accept()**

* The accept() method blocks(just sits there) until a client connects to the server.
* Then we take input from the socket using getInputStream() method. Our Server keeps receiving messages until the Client sends “Over”.
* After we’re done we close the connection by closing the socket and the input stream.
* To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.

**What Is a URL?**

If you've been surfing the Web, you have undoubtedly heard the term URL and have used URLs to access HTML pages from the Web.

It's often easiest, although not entirely accurate, to think of a URL as the name of a file on the World Wide Web because most URLs refer to a file on some machine on the network. However, remember that URLs also can point to other resources on the network, such as database queries and command output.

**Definition:**

URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.

A URL has two main components:

* Protocol identifier: For the URL http://example.com, the protocol identifier is http.
* Resource name: For the URL http://example.com, the resource name is example.com.

Note that the protocol identifier and the resource name are separated by a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:

**Host Name**

The name of the machine on which the resource lives.

**Filename**

The pathname to the file on the machine.

**Port Number**

The port number to which to connect (typically optional).

**Reference**

A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

# Java URLConnection class

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

## How to get the object of URLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

1. **public** URLConnection openConnection()**throws** IOException{}

## Displaying source code of a webpage by URLConnecton class

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

**Typically, a client program communicates with a server via a URL follows this sequence of steps:**

1. Create a **URL** object.
2. Obtain a **URLConnection** object from the **URL**.
3. Configure the **URLConnection**.
4. Read the header fields.
5. Get an input stream and read data.
6. Get an output stream and write data.
7. Close the **connection**.

TCP/IP-style networking provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to inefficient way to transport data.  
Clients and servers that communicate via a reliable channel, such as a TCP socket, have a dedicated point-to-point channel between themselves. To communicate, they establish a connection, transmit the data, and then close the connection. All data sent over the channel is received in the same order in which it was sent. This is guaranteed by the channel.  
In contrast, applications that communicate via datagrams send and receive completely independent packets of information. These clients and servers do not have and do not need a dedicated point-to-point channel. The delivery of datagrams to their destinations is not guaranteed. Nor is the order of their arrival.

**Datagram**

A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

* Datagrams plays a vital role as an alternative.
* Datagrams are bundles of information passed between machines. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it.
* Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response and it is crucial point to note.

Java implements datagrams on top of the UDP (User Datagram Protocol) protocol by using two classes:

1. **DatagramPacket** object is the data container.
2. **DatagramSocket** is the mechanism used to send or receive the DatagramPackets.

[**DatagramSocket Class**](https://www.geeksforgeeks.org/java-net-datagramsocket-class-java/)

DatagramSocket defines four public constructors. They are shown here:

**DatagramSocket( ) throws SocketException :** It creates a DatagramSocket bound to any unused port on the local computer.

**DatagramSocket(int port) throws SocketException :** It creates a DatagramSocket bound to the port specified by port.

**DatagramSocket(int port, InetAddress ipAddress) throws SocketException :** It constructs a DatagramSocket bound to the specified port and InetAddress.

**DatagramSocket(SocketAddress address) throws SocketException :** It constructs a DatagramSocket bound to the specified SocketAddress.

SocketAddress is an abstract class that is implemented by the concrete class InetSocketAddress. InetSocketAddress encapsulates an IP address with a port number. All can throw a SocketException if an error occurs while creating the socket. DatagramSocket defines many methods. Two of the most important are send( ) and receive( ), which are shown here:

void send(DatagramPacket packet) throws IOException

void receive(DatagramPacket packet) throws IOException

The send( ) method sends packet to the port specified by packet. The receive method waits for a packet to be received from the port specified by packet and returns the result.  
Other methods give you access to various attributes associated with a DatagramSocket. Here is a sampling:

| **FUNCTION** | **USAGE** |
| --- | --- |
| InetAddress getInetAddress( ) | If the socket is connected, then the address is returned. Otherwise, null is returned. |
| int getLocalPort( ) | Returns the number of the local port. |
| int getPort( ) | Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port. |
| boolean isBound( ) | Returns true if the socket is bound to an address. Returns false otherwise. |
| boolean isConnected( ) | Returns true if the socket is connected to a server. Returns false otherwise. |
| void setSoTimeout(int millis) throws SocketException | Sets the time-out period to the number of milliseconds passed in millis. |

**[DatagramPacket Class](https://www.geeksforgeeks.org/java-net-datagrampacket-class-java/)**

DatagramPacket defines several constructors. Four are shown here:

* **DatagramPacket(byte data[ ], int size) :** It specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a DatagramSocket
* **DatagramPacket(byte data[ ], int offset, int size) :**It allows you to specify an offset into the buffer at which data will be stored.
* **DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port) :** It specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent.
* **DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port) :** It transmits packets beginning at the specified offset into the data.

Think of the first two forms as building an “in box, ” and the second two forms as stuffing and addressing an envelope.

DatagramPacket defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length. In general, the get methods are used on packets that are received and the set methods are used on packets that will be sent.

| **FUNCTION** | **USAGE** |
| --- | --- |
| InetAddress getAddress( ) | Returns the address of the source (for datagrams being received) or destination (for datagrams being sent). |
| byte[ ] getData( ) | Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received. |
| int getLength( ) | Returns the length of the valid data contained in the byte array that would be returned from the getData( ) method. This may not equal the length of the whole byte array. |
| int getOffset( ) | Returns the starting index of the data. |
| int getPort( ) | Returns the port number. |
| void setAddress(InetAddress ipAddress) | Sets the address to which a packet will be sent. The address is specified by ipAddress. |
| void setData(byte[ ] data) | Sets the data to data, the offset to zero, and the length to number of bytes in data |
| void setData(byte[ ] data, int idx, int size) | Sets the data to data, the offset to idx, and the length to size. |
| void setLength(int size) | Sets the length of the packet to size. |
| void setPort(int port) | Sets the port to port. |

**A Datagram Example**

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to illustrate datagrams  import java.net.\*;  class WriteServer {        // Specified server port      public static int serverPort = 998;        // Specified client port      public static int clientPort = 999;        public static int buffer\_size = 1024;      public static DatagramSocket ds;        // an array of buffer\_size      public static byte buffer[] = new byte[buffer\_size];        // Function for server      public static void TheServer() throws Exception      {          int pos = 0;          while (true) {              int c = System.in.read();              switch (c) {              case -1:                    // -1 is given then server quits and returns                  System.out.println("Server Quits.");                  return;              case '\r':                  break; // loop broken              case '\n':                  // send the data to client                  ds.send(new DatagramPacket(buffer, pos,                                             InetAddress.getLocalHost(), clientPort));                  pos = 0;                  break;              default:                  // otherwise put the input in buffer array                  buffer[pos++] = (byte)c;              }          }      }        // Function for client      public static void TheClient() throws Exception      {          while (true) {                // first one is array and later is its size              DatagramPacket p = new DatagramPacket(buffer, buffer.length);                ds.receive(p);                // printing the data which has been sent by the server              System.out.println(new String(p.getData(), 0, p.getLength()));          }      }        // main driver function      public static void main(String args[]) throws Exception      {            // if WriteServer 1 passed then this will run the server function          // otherwise client function will run          if (args.length == 1) {              ds = new DatagramSocket(serverPort);              TheServer();          }          else {              ds = new DatagramSocket(clientPort);              TheClient();          }      }  } |

This sample program is restricted by the DatagramSocket constructor to running between two ports on local machine. To use the program, run

java WriteServer

in one window; this will be the client. Then run

java WriteServer 1

This will be the server. Anything that is typed in the server window will be sent to the client window after a newline is received.