Network Communication Programming

Network programming is the act of using **computer code** to write **programs** or **processes** that can **communicate** with other programs or processes across a **network**.

There are many **issues** that arise when doing **network programming** which do not appear when doing **single** program **applications**. However, JAVA makes networking applications simple due to the easy-to-use libraries. In general, applications that have components running on different machines are known as **distributed applications** and usually they consist of **client/server relationships**.

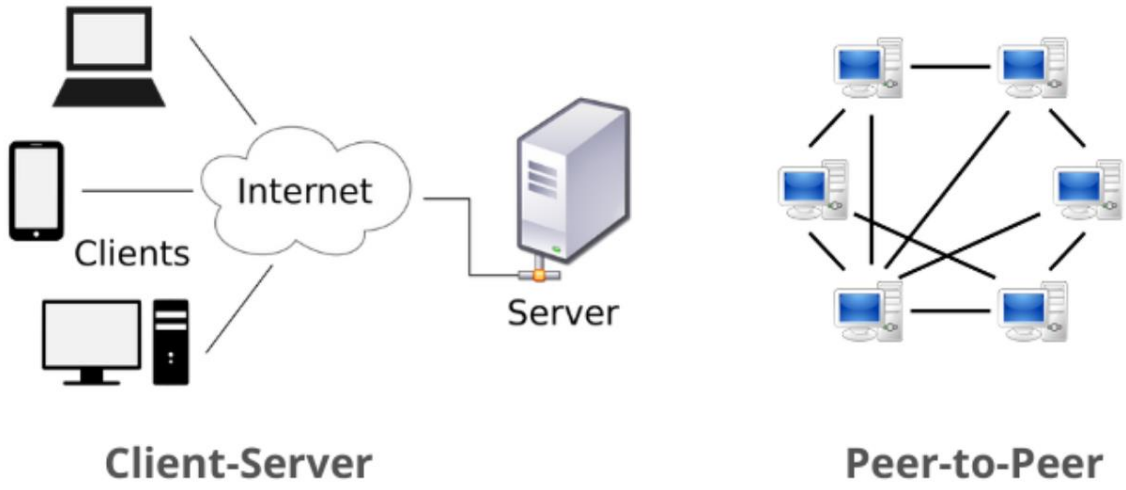
A **server** is an application that provides a "service" to various clients who request the service.

There are many **client/server** scenarios in real life:

- **Bank tellers** (server) provide a service for the **account owners** (client)
- **Waitresses** (server) provide a service for **customers** (client)
- **Travel agents** (server) provide a service for **people** wishing to go **on vacation** (client)

In some cases, servers themselves may become clients at various times. E.g., travel agents will become clients when they phone the airline to make a reservation or contact a hotel to book a room.

In the **general networking scenario**, **everybody** can either be a **client** or a **server** at **any time**. This is known as **peer-to-peer** computing. In terms of writing Java applications, it is similar to having many applications communicating among one another.



There are many **strategies** for allowing **communication** between **applications**.

We will look at the simplest strategy of connecting applications using **sockets**.

A **Protocol** is a standard pattern of exchanging information. It is like a set of rules/steps for communication.

The **Internet Protocol Suite** is a collection of protocols — that's what the word suite stands for — that determines how the Internet should work. The **TCP/IP** alias comes from two of the most important protocols the Internet Protocol Suite contains: the **Transmission Control Protocol (TCP)** and the **Internet Protocol (IP)**. It is often referred to as the **TCP/IP protocol stack** for brevity.

Cables and **computers** that make up the Internet infrastructure understand a very simple **binary language** made of zeroes and ones, and yet we want to be able to move rich data around such as web pages, emails, movies, video calls, … in a reliable, error-free and easy to establish way. This is a **complex problem** that must be **broken down** into smaller pieces to be solved efficiently. For this reason, the TCP/IP protocol stack has been organised into **four layers**.

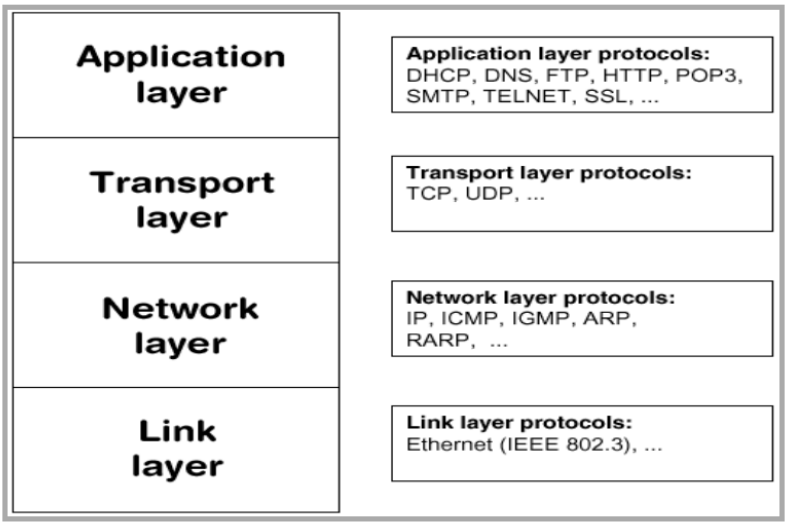
Each layer contains protocols that describe how to route/transmit/receive data according to a different level of abstraction. The lower the layer, the closer you are to the hardware and the more detailed the instructions are. The higher the layer, the closer you are to the human and the more abstract the communication becomes. Let's take a bottom-up look:

Link layer. Also known as the **physical layer**, it contains protocols that operate very close to the metal. Protocols in this layer see the network as a bunch of machines physically linked together that exchange bits of data.

Network layer. Also known as the Internet layer, this is where the communication starts to get fancy. Protocols in this layer think in terms of source networks and destination networks and how to identify them.

Transport layer. Here, the communication becomes even more abstract. Protocols in this layer think in terms of processes that talk to each other through specific channels.

Application layer. The most abstract layer, where protocols think in terms of user services that exchange application data over the network.



Initially developed by the **United States Department of Defense** and now maintained by the **Internet Engineering Task Force (IETF)**, the TCP/IP protocol stack defines how data should be handled, transmitted, routed, and received over the Internet. Anything that is connected to the Internet or operates with it must comply with the rules defined in the TCP/IP protocol stack. Two machines that want to communicate over the Internet must both implement the TCP/IP protocol stack in order to talk to each other correctly.

For example, the web browser implements the **Hypertext Transfer Protocol (HTTP)**, one of the many protocols in the TCP/IP protocol stack and the foundation of the World Wide Web (WWW). The HTTP protocol determines how the text in a web page should be sent from the web server — the remote computer that stores the information — to a web browser, over the Internet. The protocol also describes how your browser should talk to the web server in order to initiate the data exchange.

Many other software parts must be TCP/IP compliant. For example, the operating system running on your device has to implement several protocols from the TCP/IP suite, in order to provide Internet capabilities to the entire system (web browser included).

On a **TCP/IP network,** every device must have an **IP address**.

The **IP address** identifies the **device**, e.g. computer.

IPv4 is the first version of the internet protocol to be widely used. It stands for **Internet Protocol Version 4**. It was first released in 1983 and is still in use today.

IP addresses using this protocol are groups of 4 **decimal numbers** each ranging from **0 to 255**, separated by periods, called dotted decimal notation. An example of an **IPv4** address would be: 192.168.10.150.

Each number in an IP address stores information that tells data packets where to go. Each group of numbers can store one byte of information, and there are four groups in each IPv4 address, which adds up to 32 bits total of information storage. For that reason, the IPv4 addressing system is known as a 32-bit system.

This system allows for up to **4.3 billion unique addresses**, which sounds like a lot until you realize just how many people and devices are connected to the Internet today. Devices on the Internet of Things (IoT) also need IP addresses. That explosion of use as our lives become increasingly internet-connected means the IPv4 addressing system is running out of space.

IPv6 is a newer version of the Internet Protocol with longer addresses containing both numbers and letters. Though newer than version 4, it's not that new: it was first deployed in 1999.

IPv6 addresses have **128 bits** of information storage. They're written in hexadecimal colon notation, meaning each group of numbers and letters is separated from the next by a **colon** (:) instead of a period.

An example **IPv6** address would look like this: 3002:0bd6:0000:0000:0000:ee00:0033:6778

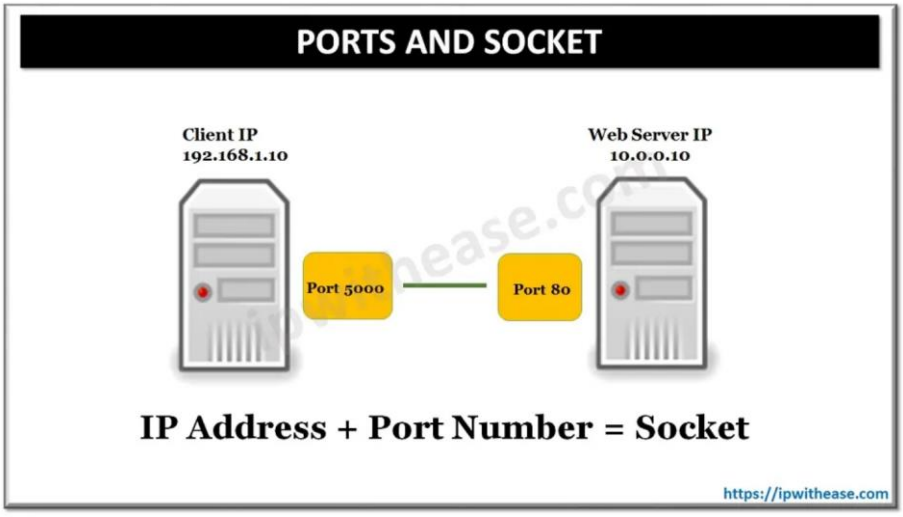
This longer addressing system supports 2^{128} unique addresses, or 10^{28} times the number of IPv4. 4.3 billion multiplied by 10^{28} is enough unique IP addresses that we won't have to worry about running out of them anytime soon.

An IP address alone is not sufficient for running network applications, as a computer can run multiple applications and/or services.

Just as the IP address identifies the computer, the **network port** identifies the a**pplication** or **service running** on the **computer**.

The use of **ports** allows computers/devices to run multiple services/applications.

The diagram below shows a **computer-to-computer** connection and identifies the IP addresses and ports.



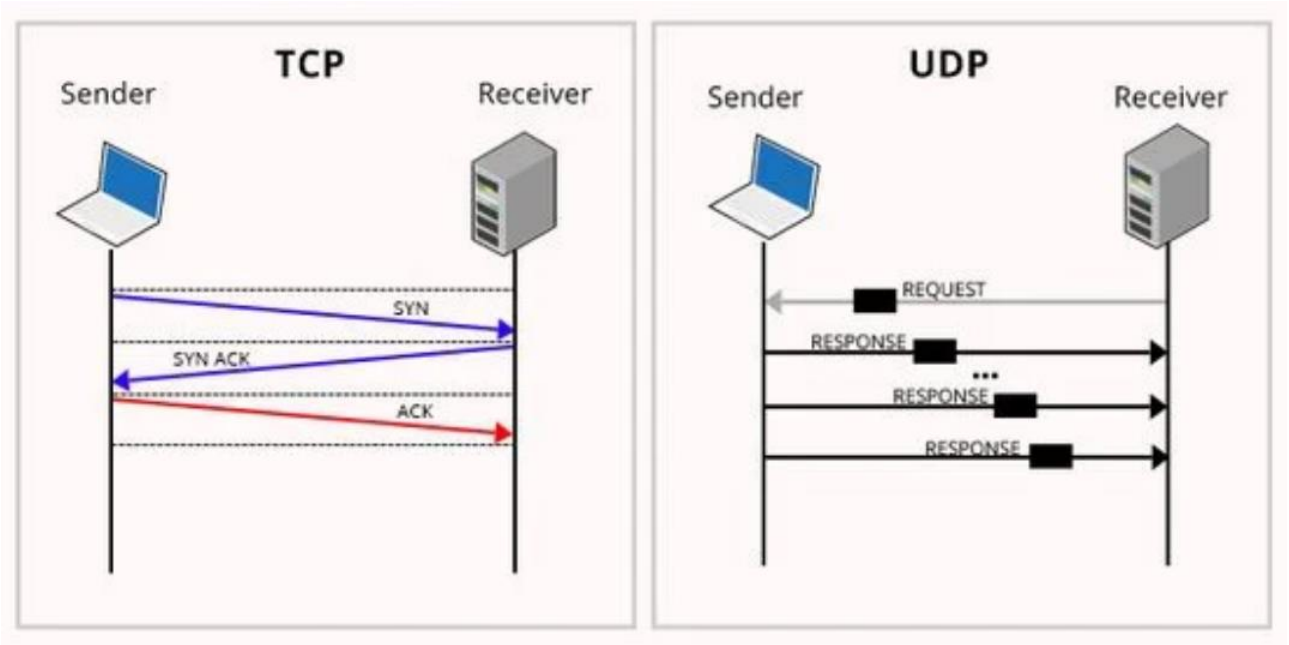
A port number uses **16 bits** and so can therefore have a value from **0** to **65535** decimal port numbers are divided into ranges as follows:

- **Port numbers 0-1023** – **Well-known ports**. These are allocated to server services by the Internet Assigned Numbers Authority (IANA). e.g. Web servers normally use port 80 and SMTP servers use port 25.
- **Ports 1024-49151**- **Registered Port** -These can be registered for services with the IANA and should be treated as semi-reserved. User written programs should not use these ports.
- **Ports 49152-65535**– These are used by client programs and you are free to use these in client programs. When a Web browser connects to a web server, the browser will allocate itself to a port in this range. Also known as ephemeral ports.

When you write **Java applications** that communicate over a **network**, you are programming in the **Application Layer**. JAVA allows two types of communication via two main types of **Transport Layer protocols**:

- **TCP.** A connection-based protocol that provides a reliable flow of data between two computers. It guarantees that data sent from one end of the connection actually gets to the other end and in the same order. TCP provides a **point-to-point channel** for applications that require **reliable communications**. It has an overhead time of setting up an end-to-end connection.
- **UDP**. This is a protocol that sends independent packets of data, called **datagrams**, from one computer to another. **UDP** gives **no guarantees** about **arrival**. UDP is not connection-based like TCP. It provides communication that is not guaranteed between the two ends. Sending packets is like sending a letter through the postal service. The order of delivery is not important and not guaranteed. Each message is independent of any other. UDP is faster since no overhead for setting up end-to-end connection. Many **firewalls** and **routers** have been configured **not** to **allow UDP packets**.

Why would anyone want to use UDP protocol if information may get lost. Well, why do we use email or the post office? We are never guaranteed that our mail will make it to the person that we send it to, yet we still rely on those delivery services. It may still be quicker than trying to contact a person via phone to convey the data (i.e., like a TCP protocol).



Java.net.NetworkInterface class in Java

This class represents the **network interface**, both software as well as hardware, its name, list of IP addresses assigned to it, and all related information. It can be used in cases when we specifically want to use a particular interface to transmit our packet in a system with multiple NICs.

What is a Network Interface?

A **network interface** can be thought of as a point at which your computer connects to the network. It is not necessarily a piece of hardware but can also be implemented in software. For example, a **loopback** interface which is used for **testing purposes**.

Method	**Description**
public String getName()	Returns the name of this network interface.
public Enumeration getInetAddresses()	Returns an enumeration of all Inetaddresses bound to this network interface, if the security manager allows it.
public List getInterfaceAddresses()	Returns a list of all interface addresses on this interface.
public Enumeration getSubInterfaces()	Returns an enumeration of all the sub or virtual interfaces of this network interface. For example, eth0:2 is a sub interface of eth0.
public NetworkInterface getParent()	In case of a subinterface, this method returns the parent interface. If this is not a subinterface, this method will return null.
public int getIndex()	Returns the index assigned to this network interface by the system. Indexes can be used in place of long names to refer to any interface on the device.
public String getDisplayName()	This method returns the name of the network interface in a readable string format.
public static NetworkInterface getByName(String name) throws SocketException	Finds and returns the network interface with the specified name, or null if none exists. **Parameters:** **name:** name of network interface to search for. **Throws:** SocketException: if I/O error occurs.

public static NetworkInterface getByIndex(int index) throws SocketException	Performs a similar function as the previous function with index used as search parameter instead of name. **Parameters:** **index:** index of network interface to search for. **Throws:** SocketException: if I/O error occurs.
public static NetworkInterface getByInetAddress(InetAddress addr) throws SocketException	This method is widely used as it returns the network interface the specified inetaddress is bound to. If an InetAddress is bound to multiple interfaces, any one of the interfaces may be returned. **Parameters:** **addr:** address to search for. **Throws:** SocketException: If IO error occurs.
public static Enumeration getNetworkInterfaces() throws SocketException	Returns all the network interfaces on the system. **Throws:** SocketException: If IO error occurs.
public boolean isUp()	Returns a boolean value indicating if this network interface is up and running.
public boolean isLoopback()	Returns a boolean value indicating if this interface is a loopback interface or not.
public boolean isPointToPoint()	Returns a boolean value indicating if this interface is a point to point interface or not.
public boolean supportsMulticast()	Returns a boolean value indicating if this interface supports multicasting or not.
public byte[] getHardwareAddress()	Returns a byte array containing the hardware address (MAC) address of this interface. The caller must have appropriate permissions before calling this method.
public int getMTU()	Returns the maximum transmission unit of this interface. An MTU is the largest size of the packet or frame that can be sent in packet-based network.

public boolean isVirtual()	Returns a boolean value indicating whether this interface is a virtual interface or not. Virtual interfaces are used in conjunction with physical interfaces to provide additional values such as addresses and MTU.
public boolean equals(Object obj)	This method is used to compare two network interfaces for equality. Two network interfaces are equal if they have the same name and addresses bound to them. **Parameters:** **obj:** Object to compare this network interface for equality
public int hashCode()	Returns the hashcode value for this object.
public String toString()	Returns a textual description of this object.

java.net.InterfaceAddress class in Java

This class represents a **network interface address**. Every device that has an **IP address** has an IP address on the **network interface**. In fact, the ping command doesn't ping a device but the device's interface address. Java provides certain methods to deal with interface addresses which can be used in places where there is a need to know the topology of the network, for fault detection in a network etc.

This class represents an **IP address**, a **netmask** and a **broadcast address** (when the address is IPv4). It only represents an IP address and a network prefix length in the case of IPv6 addresses.

Method	**Description**
public InetAddress getAddress()	Returns an InetAddress for this address.
public InetAddress getBroadcast()	Returns the InetAddress for the broadcast address for this interface address. As only IPv4 addresses have broadcast addresses, null would be returned on using an IPv6 address.
public short getNetworkPrefixLength()	getNetworkPrefixLength() : Returns the prefix length for this interface address, i.e. subnet mask for this address.
public boolean equals(Object obj)	Used for comparison of the specified object with this interface address. Returns true only if the given object is not null and represents the same interface address as this object.

	Parameters : **obj**: obj to compare with
public int hashCode()	Returns the hashcode for this interface address.
public String toString()	Returns a string representation of this interface address. The string is of the form: Interface Address/ prefix length.

java.net.InetAddress class in Java

The java.net.InetAddress class provides methods to get the **IP address** of any **hostname**. An **IP address** is represented by **32-bit** or **128-bit** unsigned number. InetAddress can handle both **IPv4** and **IPv6** addresses.

There are 2 types of addresses:

Unicast — An identifier for a single interface.

Multicast — An identifier for a set of interfaces.

The **InetAddress** class is used to encapsulate both the **numerical IP address** and the **domain name** for that address. The **InetAddress** class has no visible constructors. The InetAddress class has the inability to create objects directly, hence **factory methods** are used for the purpose. Factory Methods are **static methods** in a class that return an **object** of that class.

There are **5 factory methods** available in **InetAddress class**.

Method	**Description**
public static InetAddress getLocalHost() throws UnknownHostException	This method returns the instance of InetAddress containing the local hostname and address.
public static InetAddress getByName(String host) throws UnknownHostException	This method returns the instance of InetAddress containing IP and Host name of host represented by host argument. For machines that do not have hostnames, you can always pass a string that contains a hexadecimal form of the IP address to InetAddress.getByName().
public static InetAddress[] getAllByName(String hostName) throws UnknownHostException	This method returns the array of the instance of InetAddress class which contains IP addresses.

public static InetAddress getByAddress(byte IPAddress[]) throws UnknownHostException	This method returns an InetAddress object created from the raw IP address.
public static InetAddress getByAddress(String hostName, byte IPAddress[]) throws UnknownHostException	This method creates and returns an InetAddress based on the provided hostname and IP address.

Example:

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class InetAddressExample {
    public static void main(String[] args) throws UnknownHostException {
        // To get and print InetAddress of Local Host
        InetAddress address1 = InetAddress.getLocalHost();
        System.out.println("InetAddress of Local Host : " + address1);

        // To get and print InetAddress of Named Host
        InetAddress address2 = InetAddress.getByName("www.iessanclemente.net");
        System.out.println("InetAddress of Named Host : " + address2);

        // To get and print ALL InetAddresses of Named Host
        InetAddress address3[] = InetAddress.getAllByName("www.google.com");
        for (int i = 0; i < address3.length; i++) {
            System.out.println("ALL InetAddresses of Named Host : " + address3[i]);
        }

        // To get and print InetAddresses of Host with specified IP Address
        byte IPAddress[] = { 1, 24, 0, 1 };
        InetAddress address4 = InetAddress.getByAddress(IPAddress);
        System.out.println("InetAddresses of Host with specified IP Address : " +
address4);

        // To get and print InetAddresses of Host with specified IP Address and
hostname
        byte[] IPAddress2= { (byte)193, (byte)144, 43, (byte)236 };
        InetAddress address5 = InetAddress.getByAddress("mestre.iessanclemente.net",
IPAddress2);
        System.out.println("InetAddresses of Host with specified IP Address and
hostname : "+ address5);
    }
}
```

InetAddress class has plenty of instance methods that can be called using the object.

Method	**Description**
equals(Object obj)	This function compares this object against the specified object.
getAddress()	This method returns the raw IP address of this InetAddress object, in bytes.

getCanonicalHostName()	This method returns the fully qualified domain name for this IP address.
getHostAddress()	This method gets the IP address in string form.
getHostName()	This method returns the host name for this IP address.
hashCode()	This method gets a hashcode for this IP address.
isAnyLocalAddress()	This method utility routine to check if the InetAddress is an unpredictable address.
isLinkLocalAddress()	This method utility routine to check if the address is not linked to local unicast address.
isLoopbackAddress()	This method used to check if the InetAddress represents a loopback address.
isMCGlobal()	This method utility routine check if this address has a multicast address of global scope.
isMCLinkLocal()	This method utility routine check if this address has a multicast address of link-local scope.
isMCNodeLocal()	This method utility routine check if the multicast address has node scope.
isMCOrgLocal()	This method utility routine check if the multicast address has an organization scope.
isMCSiteLocal()	This method utility routine check if the multicast address has site scope.
isMulticastAddress()	This method checks whether the site has multiple servers.
isReachable(int timeout)	This method tests whether that address is reachable.
isReachable(NetworkInterface netif, int ttl, int timeout)	This method tests whether that address is reachable.

isSiteLocalAddress()	This method utility routine check if the InetAddress is a site-local address.
toString()	This method converts and returns an IP address in string form.

java.net.URL class in Java

The java.net.URL class provides a representation of the Internet's **Uniform Resource Locator**. A resource can be anything from a simple text file to anything else such as images, directory files, etc. A **URL** has the following parts:

- **Protocol**. The protocol is HTTP or HTTPS.
- **Hostname** or **IP**. The hostname represents the address of the machine on which the resource is located.
- **Port number**. This is an optional attribute. If not specified, it returns -1. If not specified, the default port used by the protocol indicated in the first field is used.
- **Resource name**. It is the name of a resource located on the given server. Depending on the server configuration, the file name may have a default value.

Example of URL: https://www.iessanclemente.net/o-centro/historia/

The URL class constructors were:

Method	**Description**
URL(String address) throws MalformedURLException	It creates a URL object from the specified String.
URL(String protocol, String host, String file)	Creates a URL object from the specified protocol, host, and file name.
URL(String protocol, String host, int port, String file)	Creates a URL object from protocol, host, port, and file name.
URL(URL context, String spec)	Creates a URL object by parsing the given spec in the given context.
URL(String protocol, String host, int port, String file, URLStreamHandler handler)	Creates a URL object from the specified protocol, host, port number, file, and handler.
URL(URL context, String spec, URLStreamHandler handler)	Creates a URL by parsing the given spec with the specified handler within a specified context.

The **java.net.URL constructors** are **deprecated**. Developers are encouraged to use **java.net.URI** to parse or construct a **URL**.

Important methods used in URL class are:

Method	Description
getAuthority()	Returns the authority part of URL or null if empty
getDefaultPort()	Returns the default port used
getFile()	Returns the file name
getHost()	Return the hostname of the URL in IPv6 format
getPath()	Returns the path of the URL, or null if empty
getPort()	Returns the port associated with the protocol specified by the URL
getProtocol()	Returns the protocol used by the URL
getQuery()	Returns the query part of URL. A query is a part after the '?' in the URL. Whenever logic is used to display the result, there would be a query field in the URL. It is similar to querying a database.
getRef()	Returns the reference of the URL object. Usually, the reference is the part marked by a '#' in the URL.
toString()	As in any class, toString() returns the string representation of the given URL object.

Let's see an example of Java code using the URL class:

```
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URI;
import java.net.URISyntaxException;
public class URLDemo {
    public static void main(String[] args) throws URISyntaxException,
MalformedURLException {
        URL url=new
URI("https://en.wikipedia.org/wiki/Internet#Terminology").toURL();
        System.out.println("Protocol: "+url.getProtocol());
        System.out.println("Host Name: "+url.getHost());
        System.out.println("Port Number: "+url.getPort());
        System.out.println("Default Port Number: "+url.getDefaultPort());
        System.out.println("Query String: "+url.getQuery());
        System.out.println("Path: "+url.getPath());
        System.out.println("File: "+url.getFile());
    }
}
```

java.net.URLConnection class in Java

URLConnection is an **abstract class** whose subclasses form the link between the user application and any resource on the web. We can use it to read/write from/to any resource referenced by a URL object. There are mainly two subclasses that extend the URLConnection class:

- **HttpURLConnection**. If we are connecting to any URL which uses "http" as its protocol, then HttpURLConnection class is used.
- **JarURLConnection**. If however, we are trying to establish a connection to a jar file on the web, then JarURLConnection is used.

Once the connection is established and we have a URLConnection object, we can use it to read or write or get further information about when was the page/file last modified, content length, etc.

But barely getting the state information is not the true motive of a real-world application. To retrieve the information, process it, and send the results back to the server, or just display the required information retrieved from the server is what we are aiming at.

Some of the **methods** of **URLConnection** class are the following.

Method	**Description**
getContent()	Retrieves the content of the URLConnection
getContentEncoding()	Returns the value of the content-encoding header field
getContentLength()	Returns the length of the content header field
getDate()	Returns the value of date in the header field
getHeaderFields()	Returns the map containing the values of various header fields in the HTTP header
getHeaderField(int i)	Returns the value of the ith index of the header
getHeaderField(String field)	Returns the value of the field named "field" in the header
getInputStream()	Returns the input stream to this open connection
getOutputStream()	Returns the output stream to this connection of OutputStream class
openConnection()	Opens the connection to the specified URL
setAllowUserInteraction()	Setting this true means a user can interact with the page. The default value is true
setDefaultUseCaches()	Sets the default value of useCache field as the given value
setDoInput()	Sets if the user is allowed to take input or not

setDoOutput()	Sets if the user is allowed to write on the page. The default value is false since most of the URLs don't allow to write

Typically, a client program that 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

The steps 3 to 6 are optional, and the steps 5 and 6 are interchangeable.

Next example uses the class URLConnection:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URL;
import java.net.URLConnection;

public class URLExample {
    public static void main(String[] args) throws IOException,
URISyntaxException {
        // Creating an object of URL class
        // Custom input URL is passed as an argument
        URL u = new URI("https://www.simplilearn.com/java-networking-
article").toURL();
        // 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();
        BufferedReader in = new BufferedReader(new InputStreamReader(stream));
        // Till the time URL is being read
        String line;
        while ((line = in.readLine()) != null) {
            // Continue printing the stream
            System.out.println(line);
        }
    }
}
```

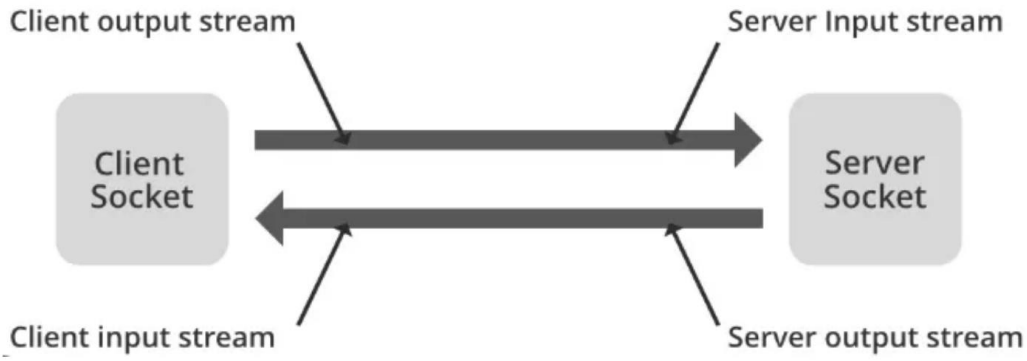
Client/Server Communication via Sockets

Many networking applications are based on the **client/ server** model. According to this model, a **task** is viewed as a **service** that can be **requested** by **clients** and **handled** by **servers**.

A **socket** is a simple communication **channel** through which two programs communicate over a **network**. A **socket** supports **two-way communication** between a **client** and a **server**, using a well-established protocol. The protocol simply prescribes rules and behaviour that both the server and client must follow in order to establish two-way communication.

According to this protocol, a **server** program creates a **socket** at a certain **port** and waits until a **client requests** a **connection**. A port is a particular address or entry point on the host computer, which typically has hundreds of potential ports. It is usually represented as a simple integer value. For example, the standard port for an HTTP (Web) server is 80. Once the **connection** is **established**, the server **creates input** and **output streams** to the **socket** and begins **sending messages** to and **receiving messages** from the **client**. Either the **client** or the **server** can **close** the **connection**, but it's usually done by the **client**.

From the client's side, the protocol goes as follows: The **client** creates a **socket** and attempts to make a **connection** to the **server**. The client must know the **server's URL** and the **port** at which the **service** exists. Once a **connection** has been **established**, the client creates **input** and **output** streams to the **socket** and begins exchanging messages with the server. The **client** can **close** the **connection** when the service is completed.



Let's now see how a **client/server application** would be coded in **Java**. The first step the server takes is to create a **ServerSocket**. The **first argument** to the **ServerSocket() method** is the **port** at which the **service** will reside. The **second argument** specifies the **number of clients** that can be **backlogged**, waiting on the server, before a **client** will be **refused service**. If more than one client at a time should request service, Java would establish and manage a waiting list, turning away clients when the list is full.

The next step is to wait for a **client request**. The **accept()** method will **block** until a **connection** is **established**. The **Java system** is responsible for **waking** the **server** when a **client request** is **received**. Once a **connection** is **established**, the **server** can begin **communicating** with the **client**. Both the client and server can "talk" back and forth to each other. The two-way conversation is managed by

connecting both an input and an output stream to the socket. Once the conversation between client and server is finished—once the server has delivered the requested service—the server can **close** the connection by calling **close()**. Thus, there are **four steps** involved on the **server side**:

- Create a ServerSocket and establish a port number.
- Listen for and accept a connection from a client.
- Converse with the client.
- Close the socket.

What distinguishes the server from the client is that the server establishes the port and accepts the connection.

```
Socket socket;     // Reference to the socket
ServerSocket port;// The port where the server will listen
try {
    port = new ServerSocket(10001, 5); // Create a port
    socket = port.accept();  // Wait for client to call
    // Communicate with the client

    socket.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
```

The **client protocol** is just as easy to implement. Indeed, on the client side, there are only three steps involved. The **first step** is to **request** a **connection** to the **server**. This is done in the **Socket()** constructor by supplying the **server's URL** and **port number**. Once the connection is established, the client can carry out **two-way communication** with the **server**. Finally, when the **client** is **finished**, it can simply **close()** the connection. Thus, from the client side, the protocol involves just three steps:

- Open a socket connection to the server, given its address.
- Converse with the server.
- Close the connection.

What distinguishes the client from the server is that the client initiates the two-way connection by requesting the service.

```
Socket connection;     // Reference to the socket
try {                  // Request a connection
  connection = new Socket("java.cs.trincoll.edu", 10001);
   // Carry on a two-way communication
    connection.close();   // Close the socket
  } catch (IOException e ) {
    e.printStackTrace();
  }
```

Let's look now at the actual **two-way communication** that takes place. Because this part of the process will be exactly the same for both **client** and **server**, we can develop a single set of methods, **writeToSocket()** and **readFromSocket()**, that may be called by either.

The **writeToSocket()** method takes two parameters, the **Socket** and a **String**, which will be sent to the process on the other end of the socket:

```
public void writeToSocket(Socket sock, String str)
                              throws IOException {
    oStream = sock.getOutputStream();
    for (int k = 0; k < str.length(); k++)
        oStream.write(str.charAt(k));
  }// writeToSocket()
```

If **writeToSocket()** is **called** by the **server**, then the **string** will be **sent** to the **client**. If it is **called** by the **client**, the **string** will be **sent** to the **server**.

Given the reference to the socket's output stream, we simply write each character of the string using the **OutputStream.write()** method. This method writes a **single byte**. Therefore, the input stream on the other side of the socket must read bytes and convert them back into characters.

The **readFromSocket()** method takes a **Socket** parameter and returns a **String**:

```
protected String readFromSocket(Socket sock)
                                 throws IOException {
    iStream = sock.getInputStream();
    String str="";
    char c;
    while (  ( c = (char) iStream.read() ) != '\n')
        str = str + c + "";
    return str;
  }
```

It uses the **Socket.getInputStream()** method to obtain a reference to the socket's input stream, which has already been created. So here again, it is important that you don't close the stream in this method. A socket's input and output streams will be closed automatically when the socket connection itself is closed.

The **InputStream.read()** method reads a single byte at a time from the input stream until an end-of-line character is received. Note the use of the cast operator (char) in the read() statement. Because bytes are being read, they must be converted to char before they can be compared to the end-of-line character or concatenated to the String. When the read loop encounters an end-of-line character, it terminates and returns the String that was input.

Trivial sequential server

This is perhaps the simplest possible server. It **listens** on **port 57777**. When a **client connects**, the **server** sends the **current date-time** to the **client**. The connection socket is created in a try-with-

resources block, so it is automatically closed at the end of the block. Only after serving the date time and closing the connection will the server go back to waiting for the next client.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.util.Date;

public class DateServer {
    public static void main(String[] args) throws IOException {
        try (var listener = new ServerSocket(57777)) {
            System.out.println("The date server is running...");
            while (true) {
                try (var socket = listener.accept()) {
                    var out = new PrintWriter(socket.getOutputStream(), true);
                    out.println(new Date().toString());
                }
            }
        }
    }
}
```

This code does not handle multiple clients well; each client must wait until the previous client is completely served before it even gets accepted. The **ServerSocket.accept()** call is a **BLOCKING CALL**.

Socket **communication** is always with **bytes**; therefore, sockets come with input streams and output streams. But by wrapping the socket's output stream with a PrintWriter, we can specify strings to write, which Java will automatically convert (decode) to bytes.

Communication through sockets is always **buffered**. This means nothing is sent or received until the buffers fill up, or you explicitly **flush** the buffer. The second argument to the **PrintWriter**, in this case true, tells Java to **flush automatically** after every println.

We defined all sockets in a **try-with-resources** block so they will automatically close at the end of their block. No explicit close call is required.

After sending the date-time to the client, the try-block ends and the communication socket is closed, so in this case, closing the connection is initiated by the server.

Let's see how to write our own client in Java:

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

public class DateClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Pass the server IP as the sole command line
argument");
            return;
        }
```

```
        var socket = new Socket(args[0], 57777);
        var in = new Scanner(socket.getInputStream());
        System.out.println("Server response: " + in.nextLine());
    }
}
```

On the client side, the Socket constructor takes the IP address and port on the server. If the connect request is accepted, we get a socket object to communicate.

Our application is so simple that the client never writes to the server, it only reads. Because we are communicating with text, the simplest thing to do is to wrap the socket's input stream in a Scanner. These are powerful and convenient. In our case, we read a line of text from the server with Scanner.nextLine.

A simple threaded server

This next server receives lines of text from a client and sends back the lines uppercased. It efficiently handles multiple clients at once. When a client connects, the server spawns a thread, dedicated to just that client, to read, uppercase, and reply. The server can listen for and serve other clients at the same time, so we have true concurrency.

```
import java.io.IOException;
import java.net.ServerSocket;

public class CapitalizeMultiServer {
    private ServerSocket serverSocket;
    public static void main(String[] args) {
        CapitalizeMultiServer server = new CapitalizeMultiServer();
        server.begin(55555);
    }
    public void begin(int port) {
        try {
            serverSocket = new ServerSocket(port);
            System.out.println("Server listening");
            while (true) {
                new Thread( new
CapitalizeClientHandler(serverSocket.accept())).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            stop();
        }
    }
    public void stop() {
        try {
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Notice that we call **accept** inside a **while loop**. Every time the while loop is executed, it **blocks** on the accept call until a new **client connects**. Then the handler thread, **CapitalizeClientHandler**, is created for this client.

What happens inside the thread is the same as the **DateServer**, where we handled only a single client. The **CapitalizeMultiServer** delegates this work to **CapitalizeClientHandler** so that it can keep listening for more clients in the while loop.

This time, we can create multiple clients, each sending and receiving multiple messages from the server.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class CapitalizeClientHandler implements Runnable{
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public CapitalizeClientHandler(Socket socket) {
        this.clientSocket = socket;
    }
    public void run() {
        try {
            out = new PrintWriter(clientSocket.getOutputStream(), true);
            in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                if (".".equals(inputLine)) {
                    out.println("bye");
                    break;
                }
                out.println(inputLine.toUpperCase());
            }
            in.close();
            out.close();
            clientSocket.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

The capitalise client could be as follows:´

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
```

```
import java.util.Scanner;

public class CapitalizeClient {
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Pass the server IP as the sole command line
argument");
            return;
        }
        CapitalizeClient capitalizeClient=new CapitalizeClient();
        capitalizeClient.startConnection(args[0],55555);
        capitalizeClient.communicate();
        capitalizeClient.stopConnection();
    }
    public void communicate() throws IOException {
        System.out.println("Enter lines of text. Finish with .");
        Scanner scanner=new Scanner(System.in);
        String inputLine;
        String cap;
        while (scanner.hasNextLine()){
            inputLine=scanner.nextLine();
            cap=this.sendMessage(inputLine);
            System.out.println(cap);
            if (cap.equals("bye")){
                break;
            }
        }
    }
    public void startConnection(String ip, int port) {
        try {
            clientSocket = new Socket(ip, port);
            out = new PrintWriter(clientSocket.getOutputStream(), true);
            in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        } catch (IOException e) {
            //LOG.debug("Error when initializing connection", e);
        }
    }
    public String sendMessage(String msg) {
        try {
            out.println(msg);
            return in.readLine();
        } catch (Exception e) {
            return null;
        }
    }
    public void stopConnection() {
        try {
            in.close();
            out.close();
            clientSocket.close();
        } catch (IOException e) {
```

```
            //LOG.debug("error when closing", e);
        }

    }
}
```

DataInputStream and DataOutputStream in Java

Using DataOutputStream and DataInputStream, it is possible to write primitive data types (byte, short, int, long, float, double, boolean, char) as well as Strings to a binary file and read them out again.

DataInputStream and DataOutputStream are very powerful classes for reading and writing primitive data types from and to a stream. They are used in a wide variety of applications, including networking, database applications, and file processing.

The **DataInputStream** methods are:

- public byte readByte()
- public short readShort()
- public int readInt()
- public long readLong()
- public float readFloat()
- public double readDouble()
- public char readChar()
- public boolean readBoolean()
- public String readUTF()
- public String readLine()

The **DataOutputStream** methods are:

- public void writeByte(byte b)
- public void writeShort(short s)
- public void writeInt(int i)
- public void writeLong(long l)
- public void writeFloat(float f)
- public void writeDouble(double d)
- public void writeChar(char ch)
- public void writeBoolean(boolean b)
- public void writeUTF(String s)
- public void writeBytes(String s)

ObjectInputStream and ObjectOutputStream in Java

Serialization is the process of converting an object into a format that can be easily stored, transmitted, or reconstructed later. Serialization is particularly useful when you need to save the state of an object to disk, send it over a network or pass it between different parts of a program.

Java employs its own **binary serialization stream protocol**. Serialization is achieved through a combination of the **ObjectOutputStream** and **ObjectInputStream** classes.

To enable **serialization**, a class must implement the **Serializable interface**.

In the following example, we can see how the server program creates a Person object, gives it values and sends it to the client program. The client program makes changes to the object and returns it modified to the server.

The example code for the server is the following:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class PersonServer {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        int numeroPuerto = 60000;// Puerto
        ServerSocket server = new ServerSocket(numeroPuerto);
        System.out.println("Waiting for the client.....");
        Socket client = server.accept();
        // the output stream is prepared for objects
        ObjectOutputStream outObject = new ObjectOutputStream(
client.getOutputStream());

        // the object is prepared and sent
        Person per = new Person("Teresa", 20);
        outObject.writeObject(per);

        //sending object
        System.out.println("Send: " + per.getName() +"*"+ per.getAge());
        // getting a stream for reading object
        ObjectInputStream inObject = new ObjectInputStream(
client.getInputStream());
        Person person = (Person) inObject.readObject();
        System.out.println("Receive: " + person.getName() + "*" +
person.getAge());
        // Close streams and sockets
        outObject.close();
        inObject.close();
        client.close();
        server.close();
    }
}
```

The example code for the client is:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class PersonClient {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        String host = "localhost";
        int port = 60000;//remote port

        System.out.println("Client started....");
        Socket client = new Socket(host,port);

        //input stream for objects
        ObjectInputStream objIn = new
ObjectInputStream(client.getInputStream());

        //receive an object
        Person person = (Person) objIn.readObject();

        //show the object data
        System.out.println("Receive: " + person.getName() + "*" +
person.getAge());

        //Modifying the object
        person.setName("Teresa Rivas");
        person.setAge(22);

        //output stream for object
        ObjectOutputStream outPerson = new ObjectOutputStream(
client.getOutputStream());

        // sending the object
        outPerson.writeObject(person);
        System.out.println("Send: " + person.getName() + "*" +
person.getAge());

        // close streams and sockets
        objIn.close();
        outPerson.close();
        client.close();
    }
}
```

The Person class is:

```
import java.io.Serializable;

public class Person implements Serializable {
    String name;
    int age;
```

```
    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public Person() { super(); }

    public String getName() {  return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) {this.age = age; }
}
```

UDP - User Datagram Protocol sockets

UDP is an alternative protocol to the more commonly used TCP protocol. It is a **connection-less** protocol where you directly send packets without having to establish a proper connection.

UDP packets have smaller headers compared to TCP headers. Also, data communication is faster since no acknowledgement is exchanged for reliable packet delivery.

UDP is used for **time-critical** data transmissions such as DNS lookups, online gaming, and video streaming. This communication protocol boosts transfer speeds by removing the need for a formal two-way connection before the data transmission begins.

In a UDP-enabled network connection, data transmission begins without the receiving party communicating an agreement to connect. Therefore, the network connections established using UDP are low latency. Additionally, this makes it imperative for applications to be tolerant of data loss. UDP is often used in situations where speed and efficiency are more important than reliability.

A UDP client begins by sending a datagram to a server that is passively waiting to be contacted. The typical **UDP client** goes through **three steps**:

- Construct an instance of DatagramSocket.
- Communicate by sending and receiving instances of DatagramPacket using the send() and receive() methods of DatagramSocket.
- When finished, deallocate the socket using the close() method of DatagramSocket.

Unlike a Socket, a **DatagramSocket** is **not** constructed with a specific **destination address**.

Like a TCP server, a **UDP** server's job is to set up a **communication endpoint** and **passively wait** for the **client** to initiate the **communication.** However, since UDP is connectionless, UDP communication is initiated by a datagram from the client, without going through a connection setup as in TCP. The typical UDP server goes through four steps:

- Construct an instance of DatagramSocket, specifying the local port and, optionally, the local address. The server is now ready to receive datagrams from any client.

- Receive an instance of DatagramPacket using the receive() method of DatagramSocket. When receive() returns, the datagram contains the client's address so we know where to send the reply.
- Communicate by sending and receiving DatagramPackets using the send() and receive() methods of DatagramSocket.
- When finished, deallocate the socket using the close() method of DatagramSocket.

In the following example, a client sends a greeting message to a server and the server answers to the client with another greeting message.

The code for the client is:

```
import java.io.IOException;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ClientUDP {
    public static void main(String[] args) {
        //server port
        final int SERVER_PORT = 50000;
        //buffer to store the messages
        byte[] buffer = new byte[1024];

        try {
            //We get the location of the server
            InetAddress serverAddress = InetAddress.getByName("localhost");

            //Creates the UDP socket
            DatagramSocket socketUDP = new DatagramSocket();

            String message = "Hello World from the client!";

            //We convert the message to bytes
            buffer = message.getBytes();

            //We create the datagram
            DatagramPacket question = new DatagramPacket(buffer, buffer.length,
serverAddress, SERVER_PORT);

            //We send the datagram
            System.out.println("Sending the datagram");
            socketUDP.send(question);

            //Preparing the packet for the answer
            DatagramPacket response = new DatagramPacket(buffer,
buffer.length);

            //Receiving the answer
            socketUDP.receive(response);
            System.out.println("Receiving the response");
```

```
            //Taking data and showing it
            message = new
String(response.getData(),0,response.getLength(),"UTF-8");

            System.out.println(message);

            //Closing socket
            socketUDP.close();

        } catch (SocketException ex) {
            Logger.getLogger(ClientUDP.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (UnknownHostException ex) {
            Logger.getLogger(ClientUDP.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (IOException ex) {
            Logger.getLogger(ClientUDP.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
}
```

The code for the server is:

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ServerUDP {
    public static void main(String[] args) {
        final int PORT = 50000;
        byte[] buffer = new byte[1024];

        try {
            System.out.println("UDP server started");
            //Creation of socket
            DatagramSocket socketUDP = new DatagramSocket(PORT);

            //It will always respond to requests
            while (true) {

                //We prepare the response
                DatagramPacket request = new DatagramPacket(buffer,
buffer.length);

                //We receive the datagram
                socketUDP.receive(request);
                System.out.println("Client information received");

                //We convert the received data and show the message
                String message = new
String(request.getData(),0,request.getLength(),"UTF-8");
                System.out.println(message);
```

```
                //We get the port and the source address
                int clientPort = request.getPort();
                InetAddress address = request.getAddress();

                message = "Hello World from the server!";
                buffer = message.getBytes();

                //We create the datagram
                DatagramPacket answer = new DatagramPacket(buffer,
buffer.length, address, clientPort);

                //We send the information
                System.out.println("Sending information to the client");
                socketUDP.send(answer);
            }
        } catch (SocketException ex) {
            Logger.getLogger(ServerUDP.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (IOException ex) {
            Logger.getLogger(ServerUDP.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
}
```