Java Networking

1.1 Introduction

Java is practically a synonym for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, crossplatform, portable code. However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the **java.net** package. They provide an easy-to-use means by which programmers of all skill levels can access network resources.

Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a port, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. **Internet Protocol (IP)** is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. **Transmission Control Protocol (TCP)** is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data. A third protocol, **User Datagram Protocol (UDP)**, sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for **FTP**; 23 is for **Telnet**; 25 is for **e-mail**; 80 is for **HTTP** and the list goes on. It is up to each protocol to determine how a client should interact with the port.

A key component of the Internet is the address. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all **Internet addresses** consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, when using Java, you won't normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you.

Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its domain name, describes a machine's location in a name space. For example, www.HerbSchildt.com is in the COM top-level domain (reserved for U.S. commercial sites); it is called HerbSchildt, and www identifies the server for web requests. An Internet domain name is mapped to an IP address by the **Domain Naming Service (DNS)**. This enables users to work with domain names, but the Internet operates on IP addresses.

1.2 Networking Classes & Interfaces

Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the java.net package are:

CacheRequest	**Inet4Address**	SocketAddress
CacheResponse	**Inet6Address**	URI
CookieHandler	**InetAddress**	**URL**
CookieManager	InetSocketAddress	**URLConnection**
DatagramPacket	**ServerSocket**	
DatagramSocket	**Socket**	

The java.net package's **interfaces** are listed here:

ContentHandlerFactory	DatagramSocketImplFactory	SocketImplFactory
CookiePolicy	FileNameMap	SocketOptions
CookieStore	ProtocolFamily	URLStreamHandlerFactory

1.3 InetAddress

The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. InetAddress can handle both IPv4 and IPv6 addresses.

Factory Methods

The InetAddress class has no visible constructors. To create an InetAddress object, you have to use one of the available factory methods. Factory methods are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used InetAddress factory methods are shown here:

- static InetAddress ***getLocalHost***( ) throws **UnknownHostException**
- static InetAddress ***getByName***(String *hostName*) throws **UnknownHostException**
- static InetAddress[ ] ***getAllByName***(String *hostName*) throws **UnknownHostException**

The **getLocalHost( )** method simply returns the InetAddress object that represents the local host. The **getByName( )** method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The **getAllByName( )** factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException if it can't resolve the name to at least one address.

Example:

```
import java.net.*;
public class InetAddressTest {
      public static void main(String args[]) throws UnknownHostException

        {

                InetAddress Address = InetAddress.getLocalHost();
                    System.out.println("Local Host : "+Address);

                 Address = InetAddress.getByName("www.facebook.com");
                    System.out.println("Google Host : "+Address);

                 InetAddress sw[] = InetAddress.getAllByName("www.google.com");

                    for(int i=0;i<sw.length;i++)
                        System.out.println("Google Host : "+sw[i]);

          }
}
```

OUTPUT:

Instance Methods

The InetAddress class has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the more commonly used methods:

boolean equals(Object other)	Returns true if this object has the same Internet address as other.
byte[ ] getAddress( )	Returns a byte array that represents the object's IP address in network byte order.
String getHostAddress( )	Returns a string that represents the host address associated with the InetAddress object.
String getHostName( )	Returns a string that represents the host name associated with the InetAddress object.
boolean isMulticastAddress( )	Returns true if this address is a multicast address. Otherwise, it returns false.
String toString( )	Returns a string that lists the host name and the IP address for convenience

Inet4Address and Inet6Address

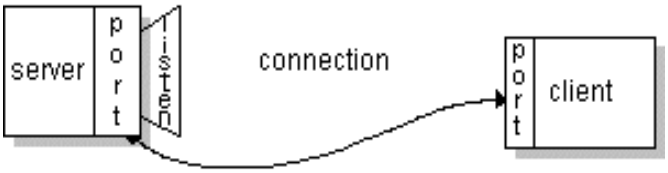
Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of InetAddress were created: Inet4Address and Inet6Address. Inet4Address represents a traditional-style IPv4 address. Inet6Address encapsulates a newer IPv6 address. Because they are subclasses of InetAddress, an InetAddress reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use InetAddress when working with IP addresses because it can accommodate both styles.

Socket Definition

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

Client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can now communicate by writing to or reading from their sockets.

A socket 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. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

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

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

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

1.4 Client/Server Interaction with Stream Socket Connections (TCP/IP):

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, 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.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is for server side. It is designed to be a "listener," which waits for clients to connect before doing anything. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges.

ServerSocket Class Methods:

Java has a different socket class that must be used for creating server applications. The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports. ServerSockets are quite different from normal Sockets. When you create a ServerSocket, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an IOException under adverse conditions. Here are three of its constructors:

SNO	Description
1	**ServerSocket(int port) throws IOException** Creates server socket on the specified port with a queue length of 50.
2	**ServerSocket(int port, int maxQueue) throws IOException** Creates a server socket on the specified port with a maximum queue length of maxQueue.
3	**ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException** Creates a server socket on the specified port with a maximum queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds.

ServerSocket has a method called **accept( )**, which is a blocking call that will wait for a client to initiate communications and then return with a normal Socket that is then used for communication with the client.

Here are some of the common methods of the ServerSocket class:

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

Example: Connection oriented Server application using Sockets (TCP)

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

public class TCPServer {
        public static void main(String args[]) {
            try{
                ServerSocket server = new ServerSocket(1234);
                System.out.println("Server Started.....!\n waiting for Client.....");
                Socket soc = server.accept();
                System.out.println("Connection Established");
                System.out.println("Host: "+soc.getLocalPort());
BufferedReader dis = new BufferedReader(new InputStreamReader(soc.getInputStream()));

                String msg = dis.readLine();
                System.out.println("Client >> " +msg);
                dis.close();
                soc.close();
                server.close();
              }
            catch(IOException e) {
                System.out.println("Server Exception: " +e);
               }
        }
}
```

TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, 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.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, ServerSocket is for servers. The Socket class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications.

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:

SNO	Constructor with Description
1	**public Socket(String host, int port) throws UnknownHostException, IOException.** This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2	**public Socket(InetAddress host, int port) throws IOException** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.
3	**public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException** Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String
5	**public Socket()** Creates an unconnected socket. Use the connect() method to connect this socket to a server.

Socket defines several instance methods. For example, a Socket can be examined at any time for the address and port information associated with it, by use of the following methods:

InetAddress getInetAddress( )	Returns the InetAddress associated with the Socket object. It returns null if the socket is not connected.
int getPort( )	Returns the remote port to which the invoking Socket object is connected. It returns 0 if the socket is not connected.
int getLocalPort( )	Returns the local port to which the invoking Socket object is bound. It returns –1 if the socket is not bound.

You can gain access to the input and output streams associated with a Socket by use of the getInputStream( ) and getOuptutStream( ) methods, as shown here. Each can throw an IOException if the socket has been invalidated by a loss of connection.

InputStream getInputStream( ) throws IOException	Returns the InputStream associated with the invoking socket.
OutputStream getOutputStream( ) throws IOException	Returns the OutputStream associated with the invoking socket.

Several other methods are available, including connect( ), which allows you to specify a new connection; isConnected( ), which returns true if the socket is connected to a server; isBound( ), which returns true if the socket is bound to an address; and isClosed( ), which returns true if the socket is closed. To close a socket, call close( ). Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, Socket also implements AutoCloseable, which means that you can use a try-with-resources block to manage a socket.

Example: Connection oriented Client application using Sockets (TCP)

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

public class TCPClient{
        public static void main(String args[]) {
           try{
                Socket client =new Socket(InetAddress.getLocalHost(),1234);
                System.out.println("Message from Client to server:");
                BufferedReader dis = new BufferedReader(new InputStreamReader(System.in));
                String msg = dis.readLine();

                PrintWriter pw = new PrintWriter(client.getOutputStream(),true);
                pw.println(msg);
```

```
                dis.close();
                client.close();
                }
        catch(IOException e){
                System.out.println("Server Exception: " +e);
                }
  }
}
```

1.5 URL:

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. Within Java's network class library, the URL class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples:

http://www.MHProfessional.com/
http://www.MHProfessional.com:80/index.htm.

A URL specification is based on four components.

- ✓ The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are **HTTP, FTP, gopher, and file**, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the "http://" from your URL specification).
- ✓ The second component is the **host name or IP address** of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:).
- ✓ The third component, the **port number**, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, ":80" is redundant.)
- ✓ The fourth part is the actual **file path**. Most HTTP servers will append a file named index.html or index.htm to URLs that refer directly to a directory resource. Thus, http://www.MHProfessional.com/ is the same as http://www.MHProfessional.com /index.htm.

Java's URL class has several constructors; each can throw a MalformedURLException. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

URL(String urlSpecifier) throws MalformedURLException

The next two forms of the constructor allow you to break up the URL into its component parts:

URL (String protocolName, String hostName, int port, String path) throws **MalformedURLException**

URL(String protocolName, String hostName, String path) throws **MalformedURLException**

Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

URL(URL urlObj, String urlSpecifier) throws MalformedURLException

The following example creates a URL to a page on HerbSchildt.com and then examines its properties:

// Demonstrate URL.

```
import java.net.*;

class URLDemo {
public static void main(String args[]) throws MalformedURLException {
URL hp = new URL(http://www.HerbSchildt.com/WhatsNew");
                System.out.println("Protocol: " + hp.getProtocol());
                System.out.println("Port: " + hp.getPort());
                System.out.println("Host: " + hp.getHost());
                System.out.println("File: " + hp.getFile());
                System.out.println("Ext:" + hp.toExternalForm());
        } }
```

When you run this, you will get the following output:

Protocol: http
Port: -1
Host: www.HerbSchildt.com
File: /WhatsNew
Ext: http://www.HerbSchildt.com/WhatsNew

Notice that the port is –1; this means that a port was not explicitly set. Given a URL object, you can retrieve the data associated with it.

1.5.1 URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for URL objects that are using the HTTP protocol. URLConnection defines several methods. Here is a sampling:

Method	Purpose
int getContentLength( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
long getContentLengthLong( )	Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned.
String getContentType( )	Returns the type of content found in the resource. This is the value of the content-type header field. Returns null if the content type is not available.
long getDate( )	Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.
long getExpiration( )	Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.
String getHeaderField(String fieldName)	Returns the value of header field whose name is specified by fieldName. Returns null if the specified name is not found.
String getHeaderFieldKey(int idx)	Returns the header field key at index idx. (Header field indexes begin at 0.) Returns null if the value of idx exceeds the number of fields.
Map<String, List<String>> getHeaderFields( )	Returns a map that contains all of the header fields and values.
long getLastModified( )	Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable.
InputStream getInputStream( ) throws IOException	Returns an InputStream that is linked to the resource. This stream can be used to obtain the content of the resource.

Notice that URLConnection defines several methods that handle header information. A header consists of pairs of keys and values represented as strings. By using getHeaderField( ), you can obtain the value associated with a header key. By calling getHeaderFields( ), you can obtain a map that contains all of the headers. Several standard header fields are available directly through methods such as getDate( ) and getContentType( ).

The following example creates a URLConnection using the openConnection( ) method of a URL object and then uses it to examine the document's properties and content:

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

public class UCDemo {
    public static void main(String args[]) throws Exception {
        int c;
        URL hp = new URL(http://www.internic.net");
        URLConnection hpCon = hp.openConnection();

        // get date
        long d = hpCon.getDate();
        if(d==0)
            System.out.println("No date information.");
        else
            System.out.println("Date: " + new Date(d));

        // get content type
        System.out.println("Content-Type: " + hpCon.getContentType());

        // get expiration date
         d = hpCon.getExpiration();

        if(d==0)
            System.out.println("No expiration information.");
        else
            System.out.println("Expires: " + new Date(d));

        // get last-modified date
            d = hpCon.getLastModified();
```

```
        if(d==0)
                System.out.println("No last-modified information.");
        else
                System.out.println("Last-Modified: " + new Date(d));
        // get content length
        long len = hpCon.getContentLengthLong();
        if(len == -1)
                System.out.println("Content length unavailable.");
        else
                System.out.println("Content-Length: " + len);
        if(len != 0) {
                System.out.println("=== Content ===");
                InputStream input = hpCon.getInputStream();
                while (((c = input.read()) != -1)) {
                                System.out.print((char) c);
                }
        input.close();
        } else {
        System.out.println("No content available.");
        }
}
}
```

The program establishes an HTTP connection to www.internic.net over port 80. It then displays several header values and retrieves the content. You might find it interesting to try this example, observing the results, and then for comparison purposes try a different web site of your own choosing.

1.6 Datagrams (UDP)

TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. 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.

Java implements datagrams on top of the UDP protocol by using two classes: the DatagramPacket object is the data container, while the DatagramSocket is the mechanism used to send or receive the DatagramPackets. Each is examined here.

1.6.1 DatagramSocket

DatagramSocket defines four public constructors. They are shown here:

- ✓ DatagramSocket( ) throws SocketException
- ✓ DatagramSocket(int port) throws SocketException
- ✓ DatagramSocket(int port, InetAddress ipAddress) throws SocketException
- ✓ DatagramSocket(SocketAddress address) throws SocketException

The **first** creates a DatagramSocket bound to any unused port on the local computer. The **second** creates a DatagramSocket bound to the port specified by port. The **third** constructs a DatagramSocket bound to the specified port and InetAddress. The **fourth** 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 a packet to the port specified by packet. The **receive( )** method waits for a packet to be received and returns the result. DatagramSocket also defines the **close()** method, which closes the socket.

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.

1.6.2 DatagramPacket:

DatagramPacket defines several constructors. Four are shown here:

- ✓ DatagramPacket(byte data [ ], int size)
- ✓ DatagramPacket(byte data [ ], int offset, int size)
- ✓ DatagramPacket(byte data [ ], int size, InetAddress ipAddress, int port)
- ✓ DatagramPacket(byte data [ ], int offset, int size, InetAddress ipAddress, int port)

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a DatagramSocket. The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent. The fourth form 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.

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.

Example: Connection less Client Server Application using Datagrams (UDP)

```
import java.io.*;
import java.net.*;
public class udpreceiver {
        public static void main(String ar[]) {
                try{
                        int port=Integer.parseInt(ar[0]);
                        byte[] msg=new byte[100];
                        DatagramSocket ds=new DatagramSocket(port);
                        DatagramPacket packet=new DatagramPacket(msg,msg.length);
                                ds.receive(packet);

                       String m=new String(msg,0,packet.getLength());
                       System.out.println(packet.getAddress().getHostName()+":"+m);
                        ds.close();
                }
                catch(Exception e) {
                        System.out.println(e);
                }
         }
  }
```

Sender:

```
import java.io.*;
import java.net.*;
public class udpsender {
public static void main(String ar[]) {
        try {
                String host=ar[0];
                int port=Integer.parseInt(ar[1]);
                byte[] msg;
                String s=ar[2];

                        for(int i=3;i<ar.length;i++)
                                s+=" "+ar[i];

                  msg=s.getBytes();
```

```
        DatagramPacket p=new DatagramPacket(msg,msg.length,InetAddress.getLocalHost(),port);
                DatagramSocket ds=new DatagramSocket();
                        ds.send(p);
                        ds.close();
                    }
            catch(Exception e) {
                    System.out.println(e);
            }     }
}
```

Chat Application:

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

public class ChatClient {
  public static void main(String[] args) throws Exception {

        DatagramSocket s = new DatagramSocket();
        byte[] buf = new byte[1000];
        DatagramPacket dp = new DatagramPacket(buf, buf.length);

    InetAddress hostAddress = InetAddress.getByName("localhost");

while (true) {
     BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
     String outMessage = stdin.readLine();
     if (outMessage.equals("bye"))
             break;
     String outString = "Client say: " + outMessage;
     buf = outString.getBytes();

     DatagramPacket out = new DatagramPacket(buf, buf.length, hostAddress, 9999);
     s.send(out);
     s.receive(dp);
     String rcvd = "rcvd from " + dp.getAddress() + ", " + dp.getPort() + ": "
        + new String(dp.getData(), 0, dp.getLength());
     System.out.println(rcvd);
    }
} }
```

Chat Server:

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

public class ChatServer {
  public static void main(String[] args) throws Exception {

    int PORT = 4000;
    byte[] buf = new byte[1000];
    DatagramPacket dgp = new DatagramPacket(buf, buf.length);
    DatagramSocket sk;

    sk = new DatagramSocket(PORT);
    System.out.println("Server started");
    while (true) {
      sk.receive(dgp);
      String rcvd = new String(dgp.getData(), 0, dgp.getLength()) + ", from address: "
          + dgp.getAddress() + ", port: " + dgp.getPort();
      System.out.println(rcvd);

      BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
      String outMessage = stdin.readLine();
      buf = ("Server say: " + outMessage).getBytes();
  DatagramPacket out = new DatagramPacket(buf, buf.length, dgp.getAddress(), dgp.getPort());
      sk.send(out);
    }
  }
}
```

1.7 Reading File from the Server

Server Application

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

public class ContentsServer{

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

  // establishing the connection with the server

        ServerSocket sersock = new ServerSocket(1234);
        System.out.println("Server Started.....!");
        Socket sock = sersock.accept();
        System.out.println("Connection is successful and wating for Client");

  // reading the file name from client
BufferedReader fileRead =new BufferedReader(new InputStreamReader(sock.getInputStream( )));
        String fname = fileRead.readLine( );

  // reading file contents
      BufferedReader contentRead = new BufferedReader(new FileReader(fname) );

  // keeping output stream ready to send the contents
        OutputStream ostream = sock.getOutputStream( );
        PrintWriter pwrite = new PrintWriter(ostream, true);
        String str;

      while((str = contentRead.readLine()) !=  null)        // reading line-by-line from file
          {

                    pwrite.println(str);                    // sending each line to client
           }
        sock.close();
        sersock.close();                                    // closing network sockets
        pwrite.close();
        fileRead.close();
        contentRead.close();
      }
}
```

Client Application

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

public class ContentsClient {
        public static void main( String args[ ] ) throws Exception {

    Socket sock = new Socket( "127.0.0.1", 1234);

// reading the file name from keyboard. Uses input stream
        System.out.print("Enter the file name");
        BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
          String fname = keyRead.readLine();

// sending the file name to server. Uses PrintWriter
        OutputStream  ostream = sock.getOutputStream( );
        PrintWriter pwrite = new PrintWriter(ostream, true);
        pwrite.println(fname);

// receiving the contents from server.  Uses input stream
    InputStream istream = sock.getInputStream();
    BufferedReader socketRead = new BufferedReader(new InputStreamReader(istream));

            FileWriter fw=new FileWriter("abc.txt");
            String str;
                while((str = socketRead.readLine())  !=  null)          // reading line-by-line
                    {
                        System.out.println(str);
                                fw.write(str);
                     }
                fw.close();
                pwrite.close();
                socketRead.close();
                keyRead.close();
         }
 }
```