**1. Explain in detail about URL’s, URL Connection, encoding and decoding.**

**The URLEncoder and URLDecoder Classes**

**URLEncoder**

The java.net.URLEncoder class contains a single static method called encode( ) that **encodes** a String.

**public static String encode(String s)**

This method always uses the **default** encoding of the platform on which it runs, so it

will produce **different** **results** on different systems.

**public static String encode(String s, String encoding)**

**throws UnsupportedEncodingException**

Both variants change any non-alphanumeric characters into **% sequences**. Both also encode all non-ASCII characters.

The **space** is converted into a **plus** sign.

import java.net.URLEncoder;

import java.io.UnsupportedEncodingException;

public class EncoderTest {

public static void main(String[] args) {

try {

System.out.println(URLEncoder.**encode**("This string has spaces", "UTF-8"));

System.out.println(URLEncoder.**encode**("This\*string\*has\*asterisks", "UTF-8"));

System.out.println(URLEncoder.**encode**("This%string%has%percent%signs“,"UTF-8"));

System.out.println(URLEncoder.**encode**("This+string+has+pluses", "UTF-8"));

System.out.println(URLEncoder.**encode**("This/string/has/slashes", "UTF-8"));

System.out.println(URLEncoder.**encode**("This\"string\"has\"quote\"marks“,"UTF-8"));

System.out.println(URLEncoder.**encode**("This:string:has:colons", "UTF-8"));

System.out.println(URLEncoder.**encode**("This.string.has.periods“,"UTF-8"));

System.out.println(URLEncoder.**encode**("This=string=has=equals=signs“,"UTF-8"));

System.out.println(URLEncoder.**encode**("This&string&has&ampersands", "UTF-8"));

}

catch (UnsupportedEncodingException ex) {

throw new RuntimeException("Broken VM does not support UTF-8");

}

}

}

Output:

javac -encoding UTF8 EncoderTest

java EncoderTest

This+string+has+spaces

This\*string\*has\*asterisks

This%25string%25has%25percent%25signs

This%2Bstring%2Bhas%2Bpluses

This%2Fstring%2Fhas%2Fslashes

This%22string%22has%22quote%22marks

This%3Astring%3Ahas%3Acolons

This.string.has.periods

This%3Dstring%3Dhas%3Dequals%3Dsigns

This%26string%26has%26ampersands

**URLDecoder**

The corresponding URLDecoder class has two static methods that **decode** strings encoded in x-www-form-url-encoded format.

That is, they convert all **plus signs to spaces and all percent escapes to their corresponding character.**

**public static String decode(String s) throws Exception**

**public static String decode(String s, String encoding) // Java 1.4**

**throws UnsupportedEncodingException**

The first variant is used in Java 1.3 and 1.2. The second variant is used in Java 1.4

and later. If you have any doubt about which encoding to use, pick UTF-8.

An **IllegalArgumentException** may be thrown if the string contains a percent sign that isn’t followed by two hexadecimal digits or decodes into an illegal sequence.

**Retrieving Data from a URL**

The URL class has several methods that retrieve data from a URL:

public InputStream **openStream**( ) throws IOException

public URLConnection **openConnection**( ) throws IOException

public URLConnection **openConnection(Proxy proxy)** throws IOException

public Object **getContent**( ) throws IOException

public Object **getContent**(Class[] classes) throws IOException

These methods differ in that they return the data at the URL as an instance of different classes.

**public URLConnection openConnection( ) throws IOException**

The openConnection( ) method opens a **socket** to the specified URL and returns a URLConnection object.

A URLConnection represents an open connection to a network resource. If the call fails, openConnection( ) throws an **IOException**.

You can’t read HTTP **header** using the **URL** class, but you can do with the **URLConnection** object returned by the openConnection( ) method

Java 1.5 adds one overloaded variant of this method that specifies the proxy server to pass the connection through:

**public URLConnection openConnection(Proxy proxy) throws IOException**

This overrides any proxy server set with the usual socksProxyHost, socksProxyPort,

http.proxyHost, http.proxyPort, http.nonProxyHosts, and similar system properties.

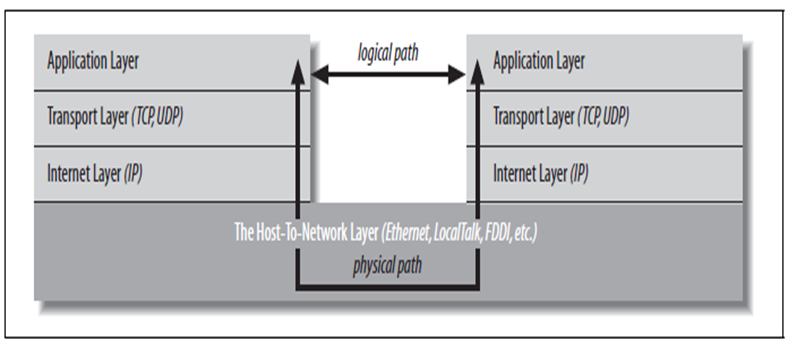
If the protocol handler does not support proxies, the argument is ignored and the connection is made directly if possible.

**2. Differentiate TCP and UDP and specify the TCP/IP layers.**

|  |  |
| --- | --- |
| **TCP** | **UDP** |
| **Reliability: TCP is connection-oriented protocol.** | **Reliability: UDP is connectionless protocol.** |
| **Ordered: If you send two messages along a connection, one after the other, you know the first message will get there first.** | **Ordered: If you send two messages out, you don’t know what order they’ll arrive in i.e. no ordered** |
| **Heavyweight: – When the low level parts of the TCP “stream” arrive in the wrong order, resend requests have to be sent, and all the out of sequence parts have to be put back together, so requires a bit of work to piece together.** | **Lightweight: No ordering of messages, no tracking connections.** |
| **Examples: SMTP, FTP.** | **Examples: DNS, Voice over IP (VoIP), Trivial File Transfer Protocol (TFTP).** |

**The Layers of a Network(Cont…)**

We uses the standard TCP/IP **four-layer model** appropriate for the Internet, shown in Figure.



**The Host-to-Network Layer:**

The host-to-network layer is also **called** as the physical layer, data link layer.

The host-to-network layer **defines** how a particular network interface - such as an Ethernet card - sends IP datagram's over its physical connection to the local network and the world.

The part of the host-to-network layer **made up** of the hardware that connects different computers (wires, fiber optic cables, microwave relays, or smoke signals) is sometimes called the physical layer of the network.

Since the physical layer is **analog**, and bits and bytes are digital, this process involves a digital-to-analog conversion on the sending end and an analog-to-digital conversion on the receiving end.

Since all real analog systems have **noise**, **error** correction and **redundancy** need to be built into the way data is translated into electricity. This is done in the data link layer.

The most common data link layer is **Ethernet**.

As a Java programmer, you don’t need to **worry** about this layer.

**The Layers of a Network(Cont…)**

**The Host-to-Network Layer:**

Since the physical layer is **analog**, and bits and bytes are digital, this process involves a digital-to-analog conversion on the sending end and an analog-to-digital conversion on the receiving end.

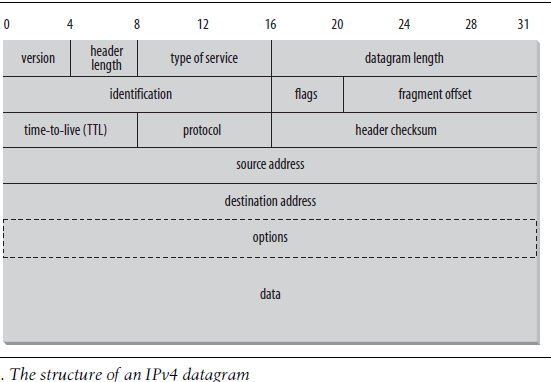
Since all real analog systems have **noise**, **error** correction and **redundancy** need to be built into the way data is translated into electricity. This is done in the data link layer.

The most common data link layer is **Ethernet**.

As a Java programmer, you don’t need to **worry** about this layer.

**The Internet Layer:**

Each IP **datagram** contains a header between **20 and 60 bytes** long and a payload that contains up to **65,515** bytes of data.



**The Transport Layer:**

The transport layer is **responsible** for ensuring that packets are received in the order they were sent and making sure that no data is lost or corrupted.

If a packet is **lost**, the transport layer can ask the sender to retransmit the packet.

IP networks implement this by adding an **additional** header to each datagram that contains more information.

**The Transport Layer:**

The transport layer is **responsible** for ensuring that packets are received in the order they were sent and making sure that no data is lost or corrupted.

If a packet is **lost**, the transport layer can ask the sender to retransmit the packet.

IP networks implement this by adding an **additional** header to each datagram that contains more information.

**The Application Layer:**

The layer that **delivers** data to the user is called the application layer.

The three lower layers all work together to **define** how data is transferred from one computer to another.

The application layer **decides** what to do with the data after it’s transferred.

For example, an application protocol like **HTTP** (for the World Wide Web) makes sure that your web browser knows to display a graphicimage as a picture, not a long stream of numbers.

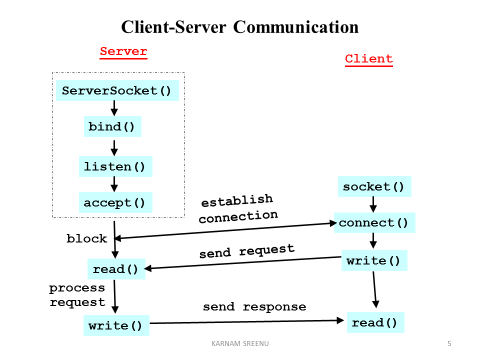
**3. Explain briefly about Socket and ServerSocket classes.**

A **socket** is a connection between two hosts. It can perform seven basic operations:

* Connect to a remote machine
* Bind to a port
* Listen for incoming data
* Accept connections from remote machines on the bound port
* Send data
* Receive data
* Close a connection

Java’s **Socket** class, which is used by both clients and servers, has methods that correspond to the first four of these operations.

The last three operations are needed only by servers, which wait for clients to connect to them. They are implemented by the ServerSocket class.



Java programs normally use client sockets in the following fashion:

1. The program creates a **new socket** with a constructor.

2. The socket attempts to **connect** to the remote host.

3. Once the connection is **established**, the local and remote hosts get

input and output streams from the socket and use those streams to send

data to each other.

This connection is **full-duplex**; both hosts can send and receive data simultaneously.

There will normally be some agreed-upon hand-shaking followed by the transmission of data from one to the other.

4. When the transmission of data is complete, one or both sides **close** the connection.

**The Socket Class**

The **java.net.Socket** class is Java’s fundamental class for performing client-side TCP operations.

The methods of the Socket class set up and tear down connections and set various **socket** **options**.

Because TCP sockets are more or less reliable connections, the interface that the Socket class provides to the programmer is **streams**.

The actual **reading** and **writing** of data over the socket is accomplished via the familiar **stream classes**.

**Sockets for Clients(Cont…): The Socket Class (Cont…)**

**The Constructors**

**public Socket(String host, int port) throws UnknownHostException, IOException**

This constructor creates a **TCP socket** to the specified port on the specified host and **attempts** to connect to the remote host.

For example:

try {

Socket toOReilly = new Socket("www.oreilly.com", 80);

// send and receive data...

}

catch (UnknownHostException ex) {

System.err.println(ex);

}

catch (IOException ex) {

System.err.println(ex);

}

In this constructor, the host argument is just a **hostname** expressed as a String.

If the domain name server cannot resolve the hostname or is not functioning, the constructor throws an **UnknownHostException**.

If the socket cannot be opened for some other reason, the constructor throws an **IOException**.

There are many reasons a connection attempt might fail: the host you’re trying to reach may **not** be accepting connections, a dialup Internet connection may be **down**, or routing problems may be **preventing** your packets from reaching their destination.

Sockets for Clients(Cont…): The Socket Class (Cont…)

import java.net.\*;

import java.io.\*;

public class LowPortScanner {

public static void main(String[] args) {

String host = "localhost";

try {

Socket s = new Socket(host, 80);

System.out.println("There is a server on port " + 80 + " of “+ host);

}

catch (UnknownHostException ex) {

System.err.println(ex);

}

catch (IOException ex) {

System.out.println("There is no server on port);

}

} // end main

}

**Sockets for Clients(Cont…): The Socket Class (Cont…)**

**public Socket(InetAddress host, int port) throws IOException**

It differs by using an **InetAddress** object to specify the host rather than a hostname.

It throws an **IOException** if it can’t connect, but does not throw an UnknownHostException; if the host is unknown, you will find out when you create the InetAddress object.

For example:

try {

InetAddress oreilly = InetAddress.getByName("www.oreilly.com");

Socket oreillySocket = new Socket(oreilly , 80);

}

catch (UnknownHostException ex) {

System.err.println(ex);

}

catch (IOException ex) {

System.err.println(ex);

}

**public Socket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException**

This constructor creates a socket to the specified port on the specified host and tries to connect.

It connects to the host and port specified in the first two arguments.

It connects from the local network interface and port specified by the last two arguments.

The network interface may be either physical (e.g., a different Ethernet card) or virtual (a multihomed host).

If **0** is passed for the localPort argument, Java chooses a random available port between 1,024 and 65,535.

**Sockets for Clients(Cont…): The Socket Class (Cont…)**

For example:

try {

InetAddress inward = InetAddress.getByName("router");

Socket socket = new Socket("mail", 25, inward, 0);

}

catch (UnknownHostException ex) {

System.err.println(ex);

}

catch (IOException ex) {

System.err.println(ex);

}

**public Socket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException**

This constructor is identical to the previous one except that the host is passed as an InetAddress, not a String.

It creates a TCP socket to the specified port on the specified host from the specified interface and local port, and tries to connect.

If it fails, it throws an IOException.

For example:

try {

InetAddress inward = InetAddress.getByName("router");

InetAddress mail = InetAddress.getByName("mail");

Socket socket = new Socket(mail, 25, inward, 0);

}

catch (UnknownHostException ex) { System.err.println(ex);

}

catch (IOException ex) { System.err.println(ex);

}

**protected Socket( )**

It creates a new Socket **without** connecting it, and is usually called by subclasses of java.net.Socket.

You can connect later by passing a SocketAddress to one of the **connect( )** methods.

The most common reason to create a Socket object without connecting is to set **socket options**; many of these cannot be changed after the connection has been made.

**4. List and explain any five client Socket options.**

**TCP\_NODELAY**

public void setTcpNoDelay(boolean on) throws SocketException

public boolean getTcpNoDelay( ) throws SocketException

setTcpNoDelay(true) turns off buffering for the socket. setTcpNoDelay(false) turns it back on.

getTcpNoDelay( ) returns true if buffering is off and false if buffering is on.

These two methods are each declared to throw a **SocketException**.

They will be thrown only if the underlying socket implementation doesn’t support the TCP\_NODELAY option.

**SO\_TIMEOUT**

public void setSoTimeout(int milliseconds) throws SocketException

publicint getSoTimeout( ) throws SocketException

Normally when you try to read data from a socket, the read( ) call blocks **as long as** necessary to get enough bytes.

By setting SO\_TIMEOUT, you ensure that the call **will not block** for more than a fixed number of milliseconds.

When the timeout expires, an InterruptedIOException is thrown.

These two methods each throw a **SocketException** if the underlying socket implementation does not support the SO\_TIMEOUT option.

The setSoTimeout( ) method also throws an **IllegalArgumentException** if the specified timeout value is negative.

**SO\_RCVBUF**

Most TCP stacks use buffers to improve network performance.

Larger buffers tend to improve performance for reasonably fast connections while slower, dialup connections do better with smaller buffers.

Starting in Java 1.2, there are methods to get and set the suggested receive buffer size used for network input:

public void setReceiveBufferSize(int size)// Java 1.2

throws SocketException, IllegalArgumentException

public int getReceiveBufferSize( ) throws SocketException // Java 1.2

The getReceiveBufferSize( ) method returns the number of bytes in the buffer that can be used for **input** from this socket.

The setReceiveBufferSize( ) method suggests a number of bytes to use for buffering **input** on this socket.

**SO\_SNDBUF**

Starting in Java 1.2, there are methods to get and set the suggested send buffer size

used for network output:

public void setSendBufferSize(int size) // Java 1.2

throws SocketException, IllegalArgumentException

public int getSendBufferSize( ) throws SocketException // Java 1.2

The getSendBufferSize( ) method returns the number of bytes in the buffer used for **output** on this socket.

The setSendBufferSize( ) method suggests a number of bytes to use for buffering **output** on this socket.

**SO\_KEEPALIVE**

If SO\_KEEPALIVE is turned on, the client will occasionally send a data packet over an idle connection (most commonly once every two hours), just to make sure the server hasn’t crashed.

If the server fails to respond to this packet, the client keeps trying for a little more than 11 minutes until it receives a response.

If it doesn’t receive a response within 12 minutes, the client closes the socket.

Without SO\_KEEPALIVE, an inactive client could live more or less forever without noticing that the server had crashed.

Java 1.3 adds methods to turn SO\_KEEPALIVE on and off and to determine its current state:

public void setKeepAlive(boolean on) throws SocketException // Java 1.3

public boolean getKeepAlive( ) throws SocketException // Java 1.3

The default for SO\_KEEPALIVE is false

**SO\_REUSEADDR // Java 1.4**

When a socket is closed, it may not immediately release the local address, especially if a connection was open when the socket was closed.

It can sometimes wait for a small amount of time to make sure it receives any lingering packets that were addressed to the port that were still crossing the network when the socket was closed.

If the **SO\_REUSEADDR** is turned on (it’s turned off by default), another socket is allowed to bind to the port even while data may be outstanding for the previous socket.

In Java this option is controlled by these two methods:

public void setReuseAddress(boolean on) throws SocketException

public boolean getReuseAddress( ) throws SocketException

5. Write a program to display the date and time of the server at client.

Example program to get current date and time from the local machine.

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

public class **DaytimeClientt** {

public static void main(String[] args) {

try {

Socket theSocket = new Socket("localhost", 13);

InputStream timeStream = theSocket.getInputStream( );

StringBuffer time = new StringBuffer( );

int c;

while ((c = timeStream.read( )) != -1)

time.append((char) c);

String timeString = time.toString().trim( );

System.out.println("It is " + timeString + " at " + "localhost");

} // end try

catch (UnknownHostException ex) { System.err.println(ex);

}

catch (IOException ex) { System.err.println(ex);

}

} // end main

} // end DaytimeClient

import java.net.\*; import java.io.\*; import java.util.Date;

public class **DaytimeServer** {

public static void main(String[] args) {

try {

ServerSocket server = new ServerSocket(13);

Socket connection = null;

while (true) {

try { connection = server.accept( );

Writer out = new OutputStreamWriter(connection.getOutputStream( ));

Date now = new Date( );

out.write(now.toString( ) +"\r\n");

out.flush( ); connection.close( );

}

catch (IOException ex) {}

finally {

try { if (connection != null) connection.close( );

}

catch (IOException ex) {}

} } // end while

} // end try

catch (IOException ex) { System.err.println(ex); } // end catch

} // end main

}