Basic Socket

Socket Addresses

- A Client must specify the IP Address of the host running the server program when it initiates communication. The network infrastructure then uses this destination address to route the client's information to the proper machine. Addresses can be specified in Java using a string that contains either a numeric address. The InetAddress abstraction represents a network destination, encapsulating both names and numerical address information. The class has two subclasses, Inet4Address and Inet6Address, representing the two versions in use.
- To get the addresses of the local host, the program takes advantage of the Network Interface abstraction. The NetworkInterface class provides access to information about all of a host's interfaces. This is extremely useful, for example when a program needs to inform another program of its address.

TCP Sockets

- Java provides two classes for TCP: Socket and ServerSocket. An instance of Socket represents one end of a TCP connection. A TCP connection is an abstract two-way channel whose ends are each identified by an IP address and port number. An instance of ServerSocket listens for TCP connection requests and creates a new Socket instance to handle each incoming connection. Thus, servers handle both ServerSocket and Socket instances, while clients use only Socket.

TCP Client

- The client initiates communication with a server that is passively waiting to be contacted. The typical TCP client goes through three steps:
    1. Construct an instance of Socket: The constructor establishes a TCP connection to the specified remote host and port.
    2. Communicate using the socket's I/O streams: A connected instance of Socket contains an InputStream and OutputStream that can be used just like any other Java I/O stream
    3. Close the connection using the close() method of Socket.

TCP Server

The typical TCP server goes through two steps:

1. Construct a ServerSocket instance, specifying the local port. This socket listens for incoming connections to the specified port.

2. Repeatedly:

a. Call the accept() method of ServerSocket to get the next incoming client connection. Upon establishment of a new client connection, an instance of Socket for the new connection is created and returned by accept().

b. Communicate with the client using the returned Socket's InputStream and OutputStream.

c. When finished, close the new client socket connection using the close() method of Socket.

Input and Output Streams

- A stream is simply an ordered sequence of bytes. Java input streams support reading bytes, and output streams support writing bytes. In our TCP client and server, each Socket instance holds an InputStream and an OutputStream instance. When we write to the output stream of a Socket, the bytes can (eventually) be read from the input stream of the Socket at the other end of the connection. OutputStream is the abstract superclass of all output streams in Java. Using an OutputStream, we can write bytes to, flush, and close the output stream.

UDP Socket

- UDP provides an end-to-end service different from that of TCP. In fact, UDP performs only two functions:
    1. it adds another layer of addressing (ports) to that of IP, and
    2. it detects some forms of data corruption that may occur in transit and discards any corrupted messages. Because of this simplicity, UDP sockets have some different characteristics from the TCP sockets
- Another difference between UDP sockets and TCP sockets is the way that they deal with message boundaries: UDP sockets preserve them.

DatagramPacket

- Instead of sending and receiving streams of bytes as with TCP, UDP endpoints exchange self-contained messages, called datagrams, which are represented in Java as instances of DatagramPacket. To send, a Java program constructs a DatagramPacket instance containing the data to be sent and passes it as an argument to the send() method of a DatagramSocket.To receive, a Java program constructs a DatagramPacket instance with preallocated space (a byte[]), into which the contents of a received message can be copied (if/when one arrives), and then passes the instance to the receive() method of a DatagramSocket.

UDP Client

- 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:

1. Construct an instance of DatagramSocket, optionally specifying the local address and port.
2. Communicate by sending and receiving instances of DatagramPacket using the send() and receive() methods of DatagramSocket.
3. When finished, deallocate the socket using the close() method of DatagramSocket.

UDP Server

- The typical UDP server goes through three steps:

1. 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.
2. 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.

3. Communicate by sending and receiving DatagramPackets using the send() and receive() methods of DatagramSocket

Sending and Receiving with UDP Sockets

A subtle but important difference is that UDP preserves message boundaries. Each call to receive() on a DatagramSocket returns data from at most one call to send(). Moreover, different calls to receive() will never return data from the same call to send().