NETWORKING

- Java supports both the **TCP** and **UDP** protocol families.
- Network communications is provided through the **java.net** package. The basic classes we will use are: **URL**, **Socket**, **ServerSocket**, and **InetAddress**.

Uniform Resource Locator (URL)

- **URL**s are the most fundamental component of the **WWW**. Browsers use URLs to identify and locate information on the web.
- The easiest way to create a **URL object** is to create it from a **String** that represents the "**human-readable**" form of the URL address.
- For example, the URL for the Data Comm site is:
- **http://milliways.bcit.ca/**
- In your Java program, you can use a string containing the above text to create a URL object:

  URL dcomm = new URL("http:// milliways.bcit.ca/ ");

- The URL object created above represents an **absolute** URL. An absolute URL contains all of the information necessary to reach the resource in question. You can also create URL objects from a **relative** URL address.
- A **URL** specification is based on **four components**:

  1. The **protocol** to use, separated from the rest of the components by a "**:**".
  2. The **hostname** or **IP address** of the host, delimited on the left by "//", and on the right by "/".
  3. The **port number** which is optional. It is delimited on the left from the hostname by a "**:**", and on the right by a "/".
  4. The actual **filename**.

- The **URL class** provides several methods that let you query URL objects. You can get the protocol, host name, port number, and filename from a URL using these accessor methods:
- ***getProtocol()***

  Returns the protocol identifier component of the URL.

- ***getHost()***

  Returns the host name component of the URL.

- ***getPort()***

  Returns the port number component of the URL. The ***getPort()*** method returns an integer that is the port number. If the port is not set, ***getPort()*** returns -1.

- ***getFile()***

  Returns the filename component of the URL.

- ***getRef()***

  Returns the reference component of the URL.

- The following example creates a URL to a homepage on the Data Comm site and then examines its properties:

```
import java.net.*;
class myURL
{
      public static void main(String args[]) throws Exception
      {
            URL hp = new URL("http://milliways.bcit.ca:80/index.html
            ");
            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());
      }
}
```

- The above will produce the following output:

Protocol: http
Port: 80
Host: milliways.bcit.ca
File: /Index.html
Ext:http://milliways.bcit.ca:80/index.html

- After successfully creating a URL, you can call the URL's ***openConnection()*** method to connect to it. When you connect to a URL you are initializing a **communication link** between your **Java program** and the **URL** over the network.

- The ***openConnection()*** method creates a new **URLConnection**, initializes it, connects to the URL, and returns the **URLConnection** object.

- If something goes wrong--for example, the server is down--then the ***openConnection()*** method throws an **IOException**.

- The following example creates a **URLConnection** object using **openConnection** and then uses the object to examine the document's properties and content:

```
import java.net.*;
import java.io.*;
class myURL
{
      public static void main(String args[]) throws Exception
      {
            int c;
            URL hp = new URL("http","142.232.66.1",80,"/");
            URLConnection hpCon = hp.openConnection();
            System.out.println("Date: " + hpCon.getDate());
            System.out.println("Type: " + hpCon.getContentType());
            System.out.println("Exp: " + hpCon.getExpiration());
            System.out.println("Last M: " + hpCon.getLastModified());
            System.out.println("Length: " + hpCon.getContentLength());
            if (hpCon.getContentLength() > 0)
            {
                  System.out.println("=== Content ===");
                  InputStream input = hpCon.getInputStream();
                  int i=hpCon.getContentLength();
                  while (((c = input.read()) != -1) && (--i > 0))
                  {
                        System.out.print((char) c);
                  }
                  input.close();
            }
            else
            {
                  System.out.println("No Content Available");
            }
      }
}
```

- The above program establishes an http connection to the local host over port 80 and requests the / or **default document**, usually **index.html**.

- The contents are retrieved and the header values are printed out.

InetAddress

- Java supports Internet naming through the **InetAddress** class.
- The **InetAddress** object is created using one of the following three methods:

  1. ***getLocalHost()***

     Returns an **InetAddress** object that represents the local host.

  2. ***getByName()***

     Returns an **InetAddress** object for a hostname passed to it.

  3. ***getAllByName()***

     Returns an array of **InetAddress** objects that represent all of the addresses that a particular name resolves to.

- Here are some examples that use the methods just described to print out the addresses and names of the **localhost**, the local **mailhost**, and some interesting Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
    public static void main(String args[]) throws UnknownHostException
    {
        InetAddress Address = InetAddress.getLocalHost();
        System.out.println(Address);
        InetAddress addr = InetAddress.getByName("mailhost");
        System.out.println(addr);
        Address = InetAddress.getByName("www.cryptonomicon.com");
        System.out.println(Address);
        InetAddress SW1[] = InetAddress.getAllByName("web.mit.edu/network/");
        for (int i=0; i<SW1.length; i++)
            System.out.println(SW1[i]);
        InetAddress SW2[] = InetAddress.getAllByName("milliways.bcit.ca");
        for (int i=0; i<SW2.length; i++)
            System.out.println(SW2[i]);

    }
}
```

Sockets

- A socket can be used to provide a point-to-point, stream-based connection between hosts on the Internet.
- More specifically, they can be used to connect Java's I/O system to other remote programs on the Internet.
- The are two main constructors for creating sockets provided by **java.net.Socket**:
- ***Socket (String host, int port)***

  Creates a socket connecting the local host to the named host and port.

- ***Socket (InetAddress Address, int port)***

  Creates a socket using a preexisting **InetAddress** object and a port.

- Exceptions thrown by the two constructors:

  UnknownHostException
  IOException

- A socket can be examined for the address and port information associated with it, using the following methods:
- ***getInetAddress ()***

  Returns the **InetAddress** associated with the Socket object.

- ***getPort()***

  Returns the remote port this Socket object is connected to.

- ***getLocalPort***

  Returns the local port this Socket object is connected to.

- Once the Socket object has been created, it can be used to gain access to the **input** and **output streams** associated with it:
- ***getInputStream()***

  Returns the **InputStream** associated with this socket.

- ***getOutputStream()***

  Returns the **OutputStream** associated with this socket.

- ***close()***

  Closes both the **InputStream** and **OutputStream**.

- The program given in your text (EchoClient) is a simple example of how to establish a connection from a client program to a server program through the use of sockets.
- This client program, **EchoClient**, connects to a standard **echo** server (in this case on port 7000) via a socket. The client does reads and writes on the socket.
- **EchoClient** sends all text typed into its standard input to the **echo** server by writing the text to the socket.
- The server echoes all input it receives from the client back through the socket to the client. The client program reads and displays the data passed back to it from the server.
- The following three lines of code within the first try block of the **main()** method are critical--they establish the socket connection between the client nd the server and open an input stream and an output stream on the socket:

  ```
  s = new Socket(args[0], 7000);

  BufferedReader in = new BufferedReader(new
          inputStreamReader(s.getInputStream()));

  PrintStream out = new PrintStream(s.getOutputStream());
  ```

- The first line in this sequence creates a new **Socket** object and names it **s**.
- The program uses the hostname specified by the user, which must be a fully qualified name of the machine that we want to connect to.
- The second argument is the port number. **Port number 7000** is the port that my **echo** server listens to.
- The second line in the code snippet above opens an **input** stream on the socket, and the third line opens an **output** stream on the socket. This allows us to read and write to and from the socket.
- **EchoClient** merely needs to write to the output stream and read from the input stream to communicate through the socket to the server. The rest of the program achieves this.

- The next section of code reads from **EchoTest**'s standard input stream (where the user can type data) a line at a time.

- **EchoClient** immediately writes the input text followed by a newline character to the output stream connected to the socket.

```
// read keyboard and write to TCP socket
try
{
        line = kybd.readLine();
        out.println(line);
}
```

- Followed by the code that reads a line of information from the **input stream** connected to the socket.

```
// read TCP socket and write to console
try
{
        line = in.readLine();
        System.out.println(line);
}
```

- The **readLine()** method **blocks** until the server echos the information back to **EchoClient**. When **readline()** returns, **EchoClient** prints the information to the standard output.

- This loop continues--**EchoClient** reads input from the user, sends it to the Echo server, gets a response from the server and displays it--until the user types an **end-of-input** character.

- This client program is straightforward and simple because the echo server implements a simple protocol.

- The client sends text to the server, the server echoes it back. When your client programs are talking to a more complicated server such as an http server, your client program will also be more complicated.

ServerSocket

- The main difference between **Sockets** and **ServerSocket** is that a **ServerSocket** will wait for a client to connect to it, whereas **Socket** treats the unavailability of a client to connect to as an error.

- When a **ServerSocket** is created, it will register itself with the system as having an interest in accepting client connections.

- It has an additional method, ***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 communications with the client.

- The following constructor creates a server socket on the specified **port** with a queue length of **50**:

  ServerSocket (int port);

- The following constructor creates a server socket on the specified **port** with a queue length of **maxQueue**:

  ServerSocket (int port, int maxQueue);

- When a connection request is detected, the ***accept*** method is invoked and the Server Socket creates a new socket, connects it to the client.

- In this way it leaves the connection on the original port free to listen for more connections.

- Here is a code fragment which creates a **ServerSocket**:

```
public static void main(String args[]) throws Exception
{
   ServerSocket acceptSocket = new ServerSocket(port);
   while (true)
   {
         String request;
         Socket s = acceptSocket.accept();
         OutputStream output = s.getOutputStream();
         InputStream input = s.getInputStream();

         ……..
         ……..
   }
}
```

Datagrams

- A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.
- Applications that communicate via datagrams send and receive completely independent packets of information. These clients and servers do not have and do not need a dedicated point-to-point channel.
- The delivery of datagrams to its destination is not guaranteed, nor is the order of arrival.
- Java implements datagrams on top of the **UDP** protocol using two classes.
- The **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s.
- **DatagramPacket**s can be created using one of two constructors:
  - **DatagramPacket (byte ibuf, int ilength)**

    Used for receiving data over a **DatagramSocket.**
  - **DatagramPacket (byte ibuf, int ilength, InetAddress iaddr, int iport)**

    Used for sending data. Appends the target address and port specified, which are used by **DatagramSocket** to determine where the data in the packet will be sent.
- Also implemented are several methods for accessing the internal state of a **DatagramPacket**:
- ***getAddress ()***

  Returns an **InetAddress** object containing the host name of the machine that sent the packet. Typically used for sending.
- ***getPort ()***

  Returns the integer destination port number. Typically used for sending.
- ***getData ()***

  Returns the byte array of data contained in the datagram. Used to retrieve data from the datagram after it has been received.
- ***getLength ()***

Returns the length of valid data in the byte array that is returned from the ***getData*** method.

- The examples provided illustrate a simple client-server model using datagrams.

- The programs use UDP connections to send datagrams between a client application and a server application.
- In the **Client** application, the user types a message into a **JTextField** on the client application and presses enter.
- The message is converted into a **byte** array and placed in a datagram packet that is sent to the server.
- The server receives the packet and displays the information in the packet, then echoes the packet back to the client. When the client receives the packet, the client displays the information in the packet.
- The constructor for class **Server** first creates the graphical user interface where the packets of information will be displayed.
- Next, the constructor creates the **DatagramSocket** the **try** block:

  socket = new DatagramSocket(5000);

- The **DatagramSocket** constructor that takes an integer port number argument **(5000)** to bind the server to a port where packets can be received from clients.
- **Clients** sending packets to this **Server** specify port **5000** in the packets they send. The constructor throws a ***SocketException*** if it fails to bind the **DatagramSocket** to a port.
- **Server** method **waitForPackets** uses an infinite loop to wait for packets to arrive at the **Server.**
- First, it creates a **DatagramPacket** object in which a received packet of information can be stored.
- The **DatagramPacket** constructor for this purpose receives two arguments-a **byte** array in which the data is stored and the length of the **byte** array.

- Next, it calls the ***receive*** method on the socket and waits for a packet to arrive.

  socket.receive(receivePacket);

- The ***receive*** method blocks until a packet arrives then stores the packet in its **DatagramPacket** argument **(receivePacket).**

- When a packet arrives, the packet's contents are appended to **JTextArea display.**
- The **DatagramPacket** method ***getAddress*** returns an **InetAddress** object containing the host name of the computer from which the packet was sent.
- Method ***getPort*** returns an integer specifying the port number through which the host computer sent the packet.
- Method ***getLength*** returns an integer representing the number of bytes of data that were sent.
- Method ***getData*** returns a **byte** array containing the data that was sent. The **byte** array is used to initialize a **String** in our program so the data can be output to the **JTextArea.**
- Next, **sendpacket** (the one to be sent back to the client) is instantiated and four arguments are passed to the **DatagramPacket** constructor.
- The first argument specifies the **byte** array to be sent.
- The second argument specifies the number of bytes to be sent.
- The third argument specifies the client computer's Internet address to which the packet will be sent.
- The fourth argument specifies the port where the client is waiting to receive packets.
- The following line of code sends the packet over the network:

  socket.send(sendPacket);

- Class **Client** works similarly to class **Server** except that the **Client** sends packets only when it is told to do so by the user typing a message in the **JTextField** and pressing the enter key in the **JTextField.**
- When this occurs, method ***actionPerformed*** is invoked, the **String** the user entered in the **JTextField** is converted into a **byte** array and the **byte** array is used to create a **DatagramPacket.**
- The **DatagramPacket** is initialized with the **byte** array, the length of the **String** that was entered by the user, the Internet address to which the packet is to be sent **(InetAddress.getByName ("milliways.bcit.ca")** in this example), and the port number at which the **Server** is waiting for packets.
- Then the packet is sent. Note that the client in this example must know that the server is receiving packets at port 5000; otherwise, the packets will not be received by the server.
- Notice that the **DatagramSocket** constructor call in this application does not specify any arguments.
- This allows the system to select the next available port number for the **DatagramSocket.**

- The client does not need a specific port number because the server receives the client's port number as part of the **DatagramPacket** sent by the client.

- Thus, the server can send packets back to the same computer and port number from which the server receives a packet of information.

- **Client** method **waitForPackets** uses an infinite loop to wait for packets using the following statement which blocks until a packet is received:

  socket.receive(receivepacket);

- Note that this does not prevent the user from sending a packet because the GUI events are handled in a different thread. It only prevents the **while** loop from continuing until a packet arrives at the **Client.**

- When a packet arrives, it is stored in **receivePacket** and its contents are displayed in the **JTextArea.**

- The user can type information into the **Client** window's **JTextField** and press the enter key at any time, even while a packet is being received.

- The ***actionPerformed*** method processes the **TTextField** event and sends the packet containing the data in the **JTextField.**