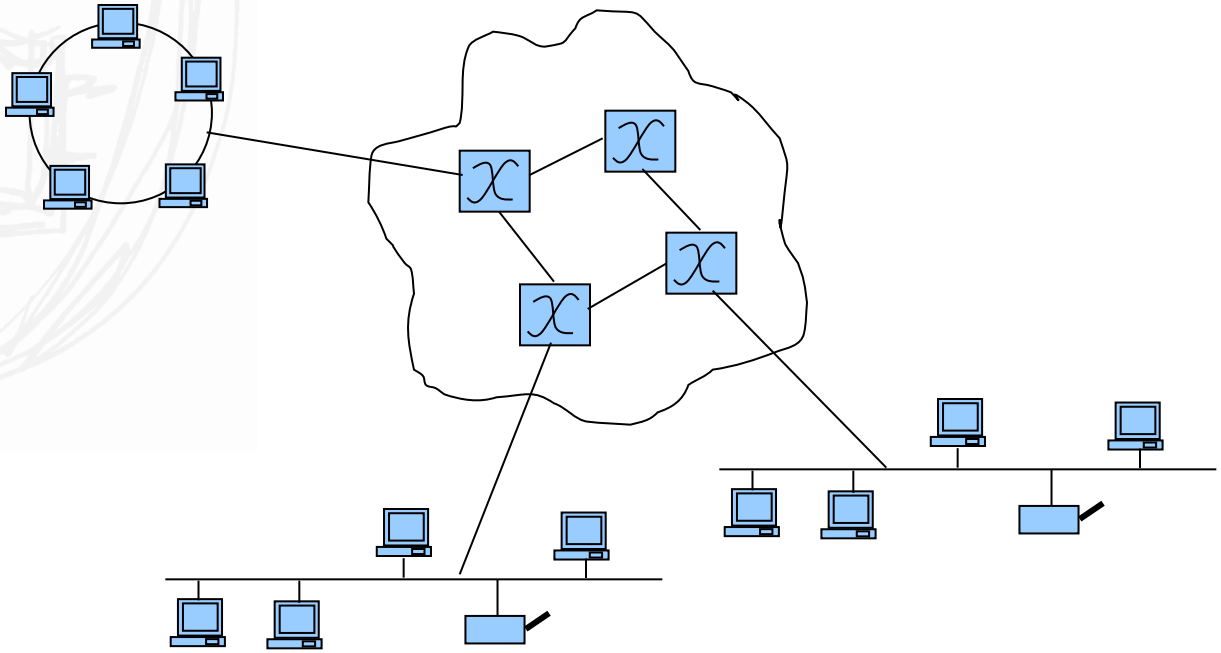
Networking

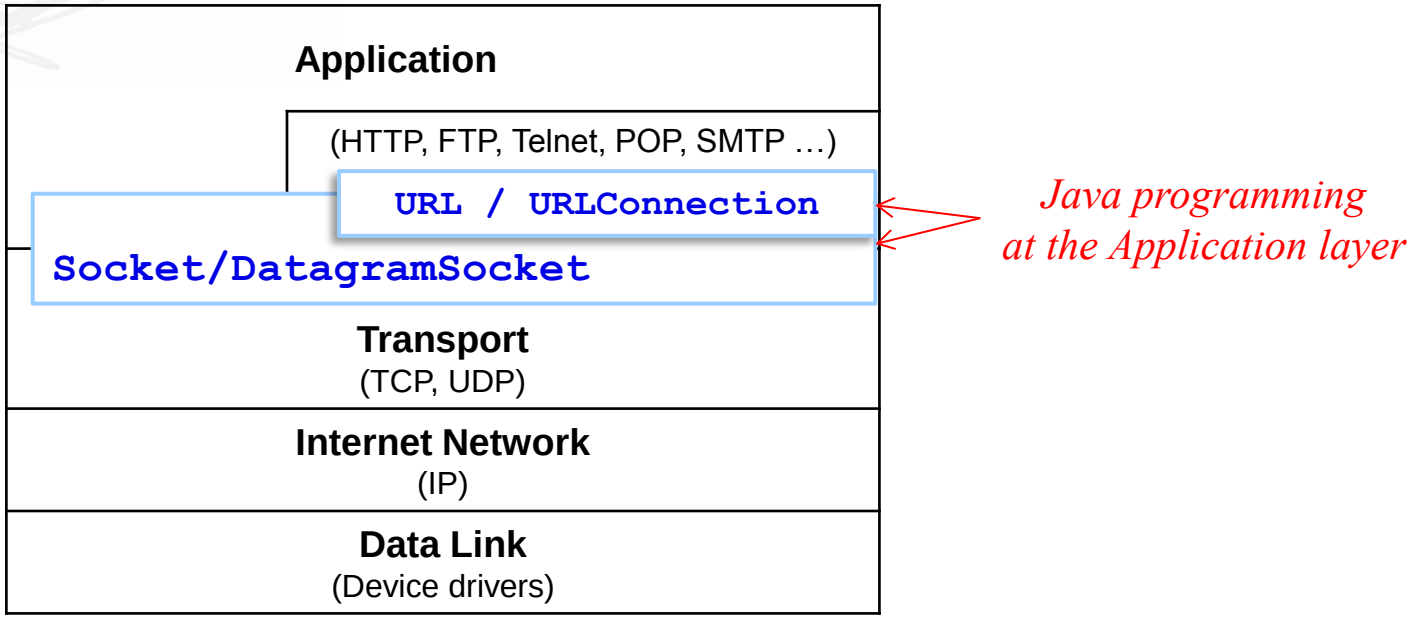
Internet



- Each computer has a unique identifier – ***IP Address***
- Data are delivered in small units – ***Packets***
- Definition of packet formats and delivery – ***Internet Protocol (IP)***

Networking Basics

- Java is the first language to provide a powerful cross-platform network support for writing programs that
  - communicate with other programs on the network
  - use and interact with the resources on the Internet and Web



Transport Protocols

- Java provides classes for networking for both TCP and UDP protocols

TCP

- Transmission Control Protocol
- TCP is a connection-based protocol that provides a *reliable flow of data* between two computers
- e.g. HTTP, FTP, Telnet

Java classes:

`Socket, ServerSocket, URL, URLConnection`

UDP

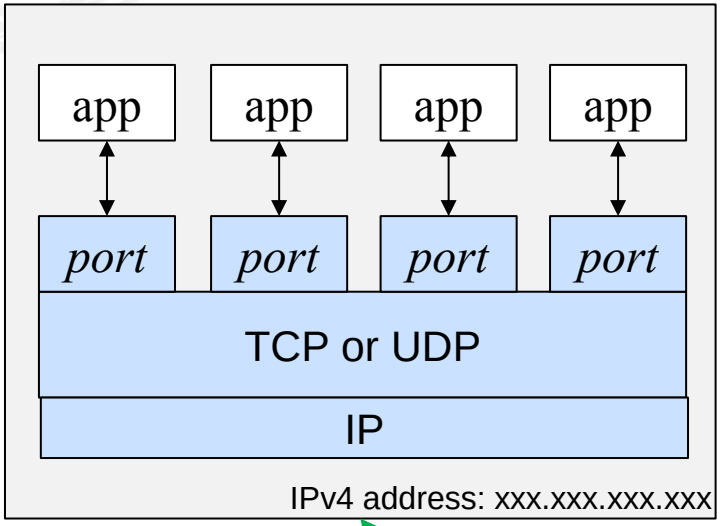
- User Datagram Protocol
- UDP is a protocol that sends *independent packets of data*, called *datagrams*, from one computer to another with no guarantees about arrival
- UDP is not connection-based like TCP
- e.g. clock server

Java classes:

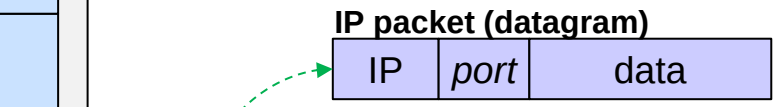
`DatagramPacket, DatagramSocket, MulticastSocket`

Ports

- *Ports* are used by TCP and UDP protocols to identify a particular process (application) to forward incoming data that transmitted through a single physical connection to the network



- Ports: 0 to 65,535
- Restricted: 0 to 1023 reserved for *well-known services*, such as HTTP (80), FTP (21), Telnet (23), ssh (22) SMTP (25), etc.



xxx.xxx.xxx.xxx

Hostname-to-IP /reverse name resolution

- local machine configuration (hosts)
- Domain Naming Service (DNS)
- Network Information Services (NIS)

Hostname
Domain name

IP Address

- Most widely deployed IP is IP version 4 (IPv4)
- IPv4 uses 32-bit (4-byte) addresses in the well known dotted decimal format: xxx.xxx.xxx.xxx
  - IP version 6 (IPv6) has been in commercial deployment since 2006, which uses 128-bit (8 groups of 4-hexadecimal) addresses

Reserved IP addresses

Classless Range	Address Range	Description
10.0.0.0/8	10.0.0.0-10.255.255.255	Private network
127.0.0.0/8	127.0.0.0-127.255.255.255	Loopback
172.16.0.0/12	172.16.0.0-172.31.255.255	Private network
192.168.0.0/16	192.168.0.0-192.168.255.255	Private network
224.0.0.0/4	224.0.0.0-239.255.255.255	IP multicast
255.255.255.255	255.255.255.255	Broadcast

Sockets

- A socket is one end-point 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
- Socket classes are used to represent the platform-independent connection between a client program (`Socket`) and a server program (`ServerSocket`)

- `ServerSocket`:
  - a socket that servers (listening applications) can use to listen for and accept connections to clients
- `Socket`:
  - implements one side of a two-way connection between your Java program and another program on the network

Stream (TCP) Programming

Server host

1. Create a server socket on a port

```
ServerSocket serverSocket =
        new ServerSocket(port);
```

2. Create a socket to listen to a connecting client

```
Socket socket =
        serverSocket.accept();
```

4. Obtain I/O streams

```
socket.getInputStream();
socket.getOutputStream();
```

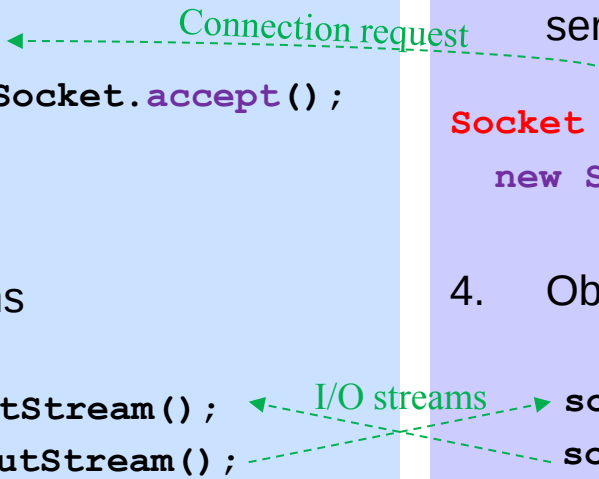
Client host

3. Create a socket to connect to the server

```
Socket socket =
  new ServerSocket(serverHost,port);
```

4. Obtain I/O streams

```
socket.getInputStream();
socket.getOutputStream();
```



Example: Data Transmission through Sockets

Server

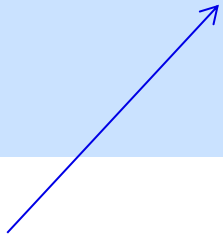
```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server =new ServerSocket(port);
socket=server.accept();

in = new DataInputStream
        (socket.getInputStream());
out = new DataOutStream
        (socket.getOutputStream());

System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

Client

```
int port = 8000;
String host="localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;


socket=new Socket(host, port);

in=new DataInputStream
      (socket.getInputStream());
out=new DataOutputStream
      (socket.getOutputStream());

out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

Basic Steps of Socket Programming

```
1    Open a socket
2      Open an input stream and output stream to the socket
3          Read from and write to the stream
                                    according to the server's PROTOCOL
4      Close the streams
5    Close the socket
```

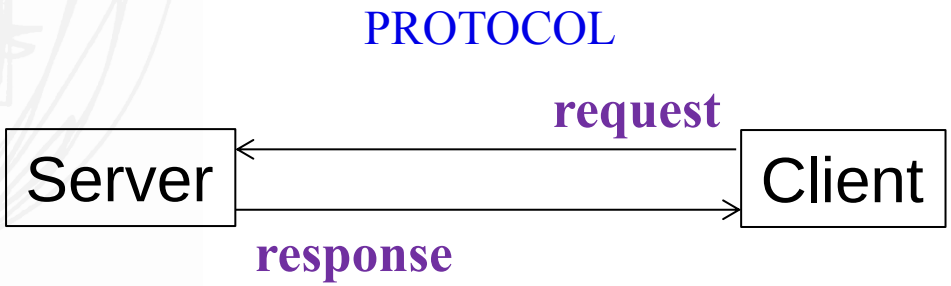


You need to define your application-specific protocol

Protocol examples:

HTTP: **`GET, POST, OPTIONS`** …
FTP: **`LIST, CWD, RETR`** (get) …

Communication Protocol



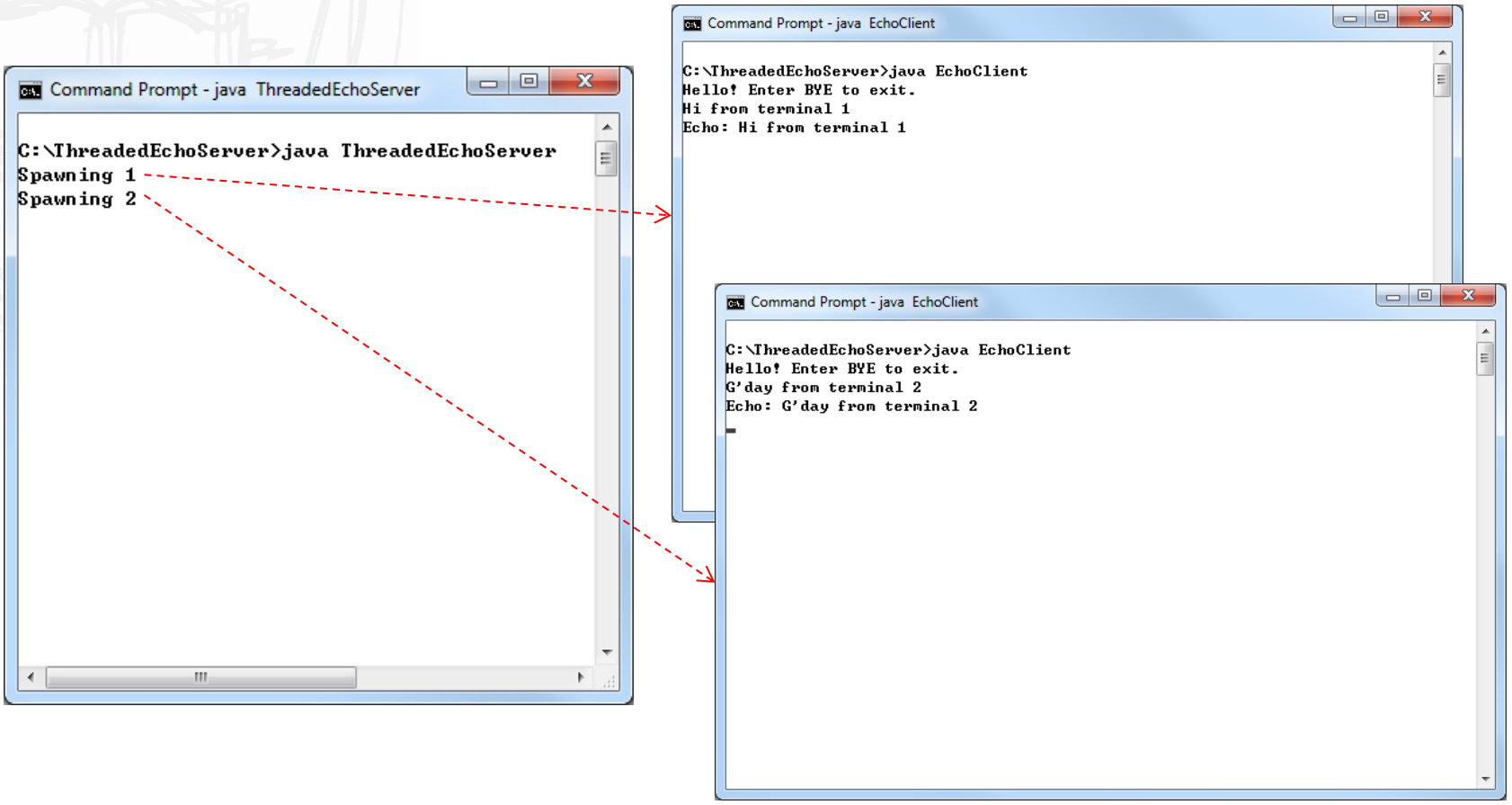
- Decide on the different commands (requests) and data to exchange for each server operation
- Associate keyword or numeric codes with each command to identify possible responses

Threaded Servers

- Serving multiple clients (Listener Mechanism)

```
while(true) {
  Socket incoming = serverSocket.accept();
  Runnable clientHandler = new ClientHandler(incoming);
  new Thread(clientHandler).start();
}
class ClientHandler implements Runnable {
  ...
  public void run(){
    try {
      InputStream inStream = incoming.getInputStream();
      OutputStream outStream = incoming.getOutputStream();
      /* processing input and send response */
      // synchronization is necessary for safety of shared data
      incoming.close();
    } catch {
      // handle exception
    }
  }
}
```

Example: Threaded Echo Server



Example: Threaded Echo Server

Server main loop

```
public class ThreadedEchoServer {
   public static void main(String[] args ) {
      try {

            int i = 1;
            ServerSocket s = new ServerSocket(8000);

            while (true) {
               Socket incoming = s.accept();
               System.out.println("Spawning " + i);
               Runnable r = new ThreadedEchoHandler(incoming);
               Thread t = new Thread(r);
               t.start();
               i++;

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

Example: Threaded Echo Server

Server Client Handler

```
class ThreadedEchoHandler implements Runnable
{  . . .
   public void run() {
     try{
       try {
         InputStream inStream = incoming.getInputStream();
         OutputStream outStream = incoming.getOutputStream();
         Scanner in = new Scanner(inStream);
         PrintWriter out = new PrintWriter(outStream, true /*autoFlush*/);
         out.println( "Hello! Enter BYE to exit." );
         // echo client input
         boolean done = false;
         while (!done && in.hasNextLine()) {
           String line = in.nextLine();
           out.println("Echo: " + line);
           if (line.trim().equals("BYE"))
             done = true;
         }
       } finally {
         incoming.close();
       }
     }
   }
}
```

Echo server protocol to disconnect

Example: Echo Client

```
public class EchoClient {
  public static void main(String[] args) throws IOException {
    try {
       Socket echoSocket = new Socket (
                              InetAddress.getLocalHost().getHostName(), 8000);
       PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true);
       Scanner in = new Scanner(echoSocket.getInputStream());
       System.out.println(in.nextLine());
       Scanner stdIn = new Scanner(System.in);
       String userInput;
       while ((userInput = stdIn.nextLine()) != null) {
         out.println(userInput);
         if(! "BYE".equals(userInput)) System.out.println(in.nextLine());
         else break;
       }
      out.close();
      in.close();
      echoSocket.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Serializable Objects for Socket

- Send and receive objects on socket streams

```
ObjectInputStream
ObjectOutputStream
```

- The objects must be **`Serializable`**
  - To enable transmission of objects on socket streams

```
class MyClass implements Serializable
```

Example: Sending and Receiving Objects

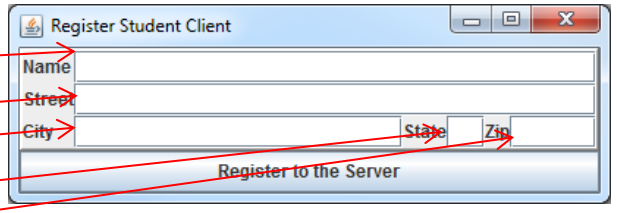
Student class

```
public class StudentAddress implements Serializable {

  private String name;
  private String street;
  private String city;
  private String state;
  private String zip;

  public StudentAddress(String name, String street, String city,
    String state, String zip) {
    this.name = name;
    this.street = street;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getName() {
    return name;
  }

  public String getStreet() {
    return street;
  }

. . .
}
```

Example: Sending and Receiving Objects

Student Client

```
// Establish connection with the server
Socket socket = new Socket(host, 8000);

// Create an output stream to the server
ObjectOutputStream toServer =
    new ObjectOutputStream(socket.getOutputStream());

// Get text field
String name = jtfName.getText().trim();
String street = jtfStreet.getText().trim();
String city = jtfCity.getText().trim();
String state = jtfState.getText().trim();
String zip = jtfZip.getText().trim();

// Create a Student object and send to the server
StudentAddress s = new StudentAddress(name, street, city, state, zip);

toServer.writeObject(s);
```



Example: Sending and Receiving Objects

Student Server

```
// Create a server socket
ServerSocket serverSocket = new ServerSocket(8000);

// Create an object output stream
outputToFile = new ObjectOutputStream(
      new FileOutputStream("student.dat", true));

while (true) {
  // Listen for a new connection request
   Socket socket = serverSocket.accept();

  // Create an input stream from the socket
   inputFromClient = new ObjectInputStream(socket.getInputStream());

  // Read from input
   Object object = inputFromClient.readObject();

  // cast the object and print student information
   StudentAddress student = (StudentAddress) object;
  System.out.println("Student name: " + student.getName());
  System.out.println("Student Address: " + student.getStreet() +", "  + . . .);
}
```

Reading and Writing at Sockets

- Java to Java
  - Text:
    - **`Scanner/PrintWriter`** or **`BufferedReader/BufferedWriter`**
    - Java sends Unicode characters
  - Primitive data:
    - **`DataInputStream/DataOutputStream`**
    - Java **`int`**, **`double`**, UTF-8 data for Web
  - Objects:
    - **`ObjectInputStream/ObjectOutputStream`**
    - Often this is the best way

- Java to unknown
  - **`DataInputStream/DataOutputStream`**
    - Send data as text using byte transfers
    - Not all languages handle characters as Unicode characters

Socket Timeouts

- Reading from a socket blocks !
  1. Set the timeout

```
Socket socket = new Socket(…);
socket.setSoTimeout ( milliseconds );

            // all read/write throws SocketTimeoutException
```

  2. Catch the exception

```
try {
   InputStream in = soccket.getInputStream();
   ...
} catch (InterruptedIOException) {
   // react to timeout
}
```

- The Socket constructor can block indefinitely until connected

```
Socket socket = new Socket();
socket.connect(new InetSocketAddress(hostname, port), timeout);
```

Interruptible Sockets

- Enable users to cancel a socket connection that does not appear to produce results
- Use **`SocketChannel`**
  - when a thread is interrupted during an open, read or write operation, the operation does not clock, but is terminated with an exception

```
SocketChannel channel = SocketChannel.open(
                                new InetSocketAddress(host, port));
try {
  Scanner in = new Scanner (channel);
  OutputStream outStream = Channels.newOutputStream(channel);
  while(!Thread.currentThread().isInterrupted())
     // read and write to the socket (channel)
} catch {…}
} finally {
   channel.close()
}
```

Internet Address

- Convert between host name and Internet address
  - **`InetAddress`** encapsulates the IP address (IPv4 and IPv6)

```
InetAddress address = InetAddress.getByName("www.uow.edu.au");
```

InetAddress
+getByName(String host): InetAddress +getAllByName(String host): InetAddress[] +getLocalHost(): InetAddress +getLocalHost(): String +getAddress(): byte[] +getHostAddress(): String +getHostName(): String

Selected methods are shown

Example: HTTP Session

Example conversation between HTTP client and Web server on a socket at port 80

Client request

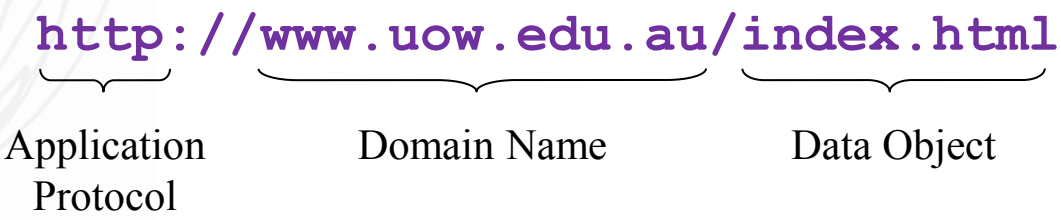
```
GET /index.html HTTP/1.1\r\n
Host: www.uow.edu.au\r\n
\r\n
```

Server response

```
HTTP/1.0 200 OK
Date: Sun, 13 May 2012 12:19:04 GMT
Server: Apache/2.2.19 (Unix) mod_ssl/2.2.19
        OpenSSL/0.9.7d DAV/2
Content-Type: text/html
Content-Length: 50415
Age: 1289
X-Cache: HIT from kami.its.uow.edu.au
Via: 1.1 kami.its.uow.edu.au:80
     (squid/2.7.STABLE6)
Connection: close

<!DOCTYPE html>
<html>
<head><meta charset="UTF-8">
. . .
```

URL (Uniform Resource Locator)

- URL address
  - a pointer to a "resource" on the World Wide Web



- **`URL`** class
  - **`URL`** (and **`URLConnection`**) objects encapsulates much of complexity of retrieving information from a remote site
  - Higher level than making a socket connections and issuing HTTP requests

```
URL url = new URL(urlString);
URL url = new URL(URL baseURL, String relativeURL);
```

  - Guaranteed protocols: http, https, ftp, file, and jar

URL and URLConnection Classes

- Retrieving contents of resources directly from a URL

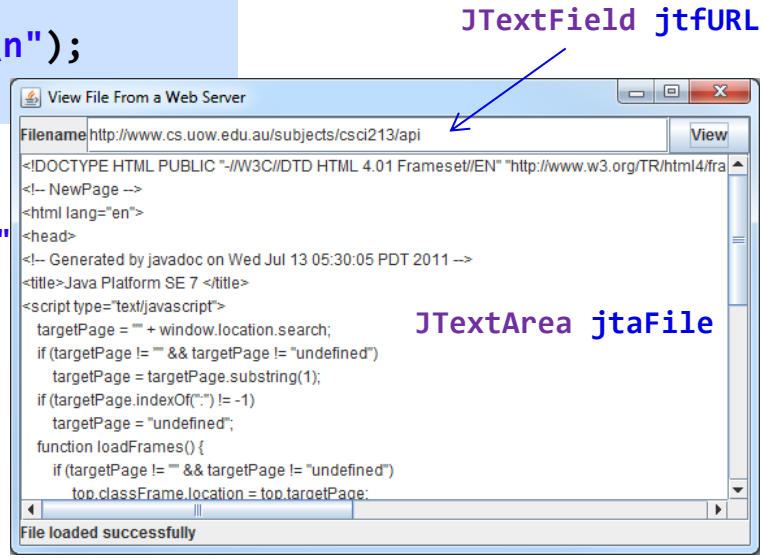
```
URL url = new URL(urlString);  // create a URL object for the file
InputStream inStream = url.openStream(); // open an input stream
Scanner in = new Scanner(inStream);
```

- Using **URLConnection** class for additional information about a web resource

```
URLConnection conn = url.openConnection();
// 1. set request properties eg. setConnectionTimeout() etc.
conn.setDoInput(true);
conn.setDoOutput(true);
// 2. open a connection
conn.connect();
// 3. query header information with, if required
Map header = conn.getHeaderField();
// 4. get I/O streams from the connection
InputStream in = conn.getInputStream();  //also can use getContent()
PrintWriter out = new PrintWriter(conn.getOutputStream());
```

Example: Retrieving Files from Web

```
private void showFile() {
Scanner input = null; // Use Scanner for getting text input
  URL url = null;
  try {
    // Obtain URL from the text field
    url = new URL(jtfURL.getText().trim());
    // Create a Scanner for input stream
    input = new Scanner(url.openStream());
    // Read a line and append the line to the text area
    while (input.hasNext()) {
        jtaFile.append(input.nextLine() + "\n");
    }
    jlblStatus.setText("File loaded successfully");
  } catch (MalformedURLException ex) {
     jlblStatus.setText("URL " + url + " not found.");
  } catch (IOException e) {
      jlblStatus.setText(e.getMessage());
  } finally {
      if (input != null) input.close();
  }
}
```



JEditorPane

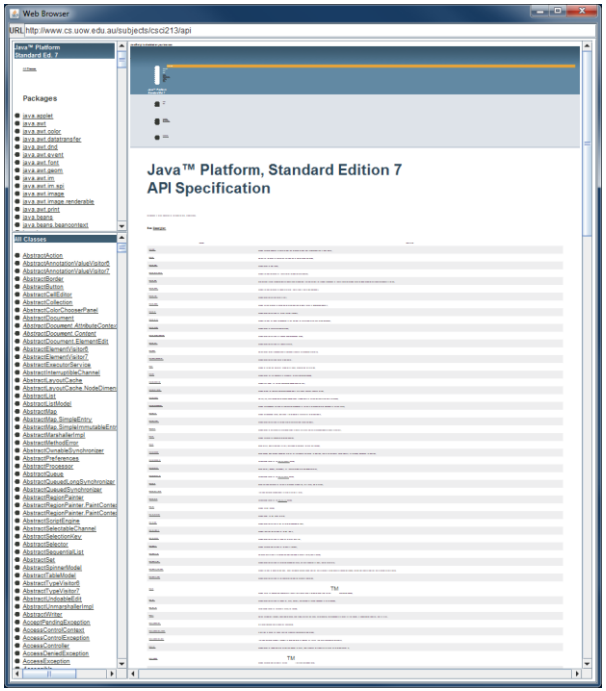
- The GUI component **`JEditorPane`** can be used to display plain ***text, HTML***, and ***RTF*** files automatically
  - you don't have to write code to explicit read data from the files
  - **`JEditorPane`** is a subclass of **`JTextComponent`**
  - Supports frames, hyperlinks, images and CSS but not JavaScript and plugins. Supports can be added by recognizing relevant parts of HTML
- To display the content of a file described by a URL

```
public void setPage(URL url) throws IOException
```

- **`JEditorPane`** generates **`HyperlinkEvent`** when a hyperlink in the editor pane is clicked
  - Through this event, you can get the URL of the hyperlink and display it using the **`setPage(url)`** method

Example: Displaying HTML

```
// JEditor pane to view HTML files
private JEditorPane jep = new JEditorPane();
add(new JScrollPane(jep), BorderLayout.CENTER);
. . .
//Get the page from the URL text field
public void actionPerformed(ActionEvent e)
  try {
     // Get the URL from text field
     URL url = new URL(jtfURL.getText().trim());

     // Display the HTML file
     jep.setPage(url);
  }
. . .
//Get the page by following the link in the pages
public void hyperlinkUpdate(HyperlinkEvent e) {
  if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
    try {
       jep.setPage(e.getURL());
    } catch (IOException ex) {
        System.out.println(ex);
    }
  }
}
```



Datagram (UDP) Programming

- A datagram is an independent, self-contained message sent over the network whose *arrival*, *arrival time*, and *content* are not guaranteed
- The clients and servers do not have and do not need a dedicated point-to-point channel
- Two Java classes: **`DatagramPacket`** and **`DatagramSocket`** to support UDP programming
- Datagram packets can be broadcast to multiple recipients all listening to a **`MulticastSocket`**.

Send and Receive Datagrams

- To send data, you put the data in a **`DatagramPacket`** and send the packet using a **`DatagramSocket`**
  - The address and port to send a **`DatagramPacket`** is included in **`DatagramPacket`**
  - The **`DatagramSocket`** needs only to know the port to send
  - A single **`DatagramSocket`** can send data to and receive data from many independent hosts
- To receive data, you receive a **`DatagramPacket`** object from a **`DatagramSocket`** and then read the contents of the packet
  - Receiving **`DatagramSocket`** does not need to know the host, needs only to know the port to listen
  - Get the address and port of the sender from received **`DatagramPacket`**
    - **`getAddress()`**
    - **`GetPort()`**

Datagram Sockets

- A client socket can be anonymous
  - constructing a **`DatagramSocket`** without port and the actual port is assigned by the system and placed in the outgoing datagrams that a server can use to respond)

- A server socket must specify the port to listen so that a client can send requests

- There's no distinction between client sockets and server sockets, as there is with TCP

Construct DatagramPacket

- **`DatagramPacket`** for receiving datagrams

```
public DatagramPacket(byte[] buffer, int length)
```

  - Most native UDP implementations don't support more than 8,192 bytes of data per datagram
  - The theoretical limit for an IPv4 datagram is 65,507 bytes of data, and a DatagramPacket with a 65,507-byte buffer can receive any possible IPv4 datagram without losing data. IPv6 datagrams raise the theoretical limit to 65,536 bytes
  - Almost all UDP datagrams you're likely to encounter will have 8K of data or fewer

- **`DatagramPacket`** for sending datagrams

```
public DatagramPacket(byte[] data, int length,
                      InetAddress destination, int port)
```

Read Content of Datagrams

- The **`getData( )`** method returns a byte array containing the data from the datagram
- Receiving text

```
public String(byte[] buffer, String encoding)

e.g. String s = new String(dp.getData( ), "ASCII");
```

- Receiving data
  - to convert the byte array returned by getData( ) into a ByteArrayInputStream first and then chain it to desired I/O stream

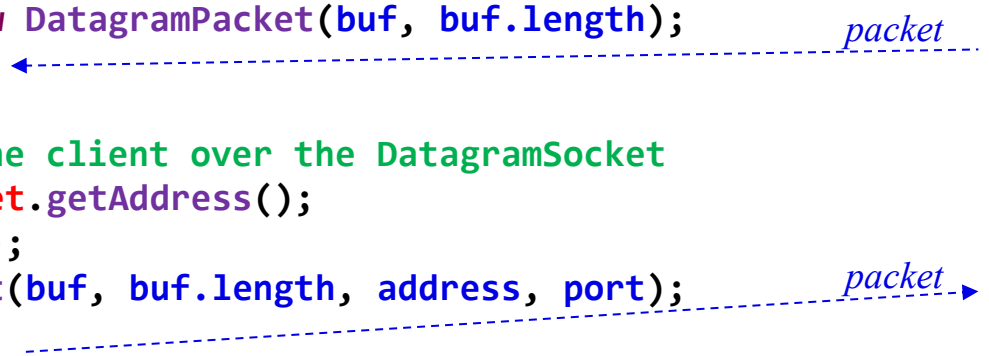
```
public ByteArrayInputStream(byte[] buffer, int offset, int length)

e.g. InputStream in = new ByteArrayInputStream(dp.getData( ),
                                    dp.getOffset(), dp.getLength( ));
     DataInputStream din = new DataInputStream(in);
     int aNumber = din.readInt();
```

Example: Client and Server

Steps for Server

The server continuously receives datagram packets over a datagram socket. Each datagram packet received by the server indicates a client request for a quotation. When the server receives a datagram, it replies by sending a datagram packet that contains a one-line "quote of the moment" back to the client.

```
// Create a DatagramSocket
DatagramSocket socket = new DatagramSocket(port);

// receive requests from clients
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);

// sends the response to the client over the DatagramSocket
InetAddress address = packet.getAddress();
int port = packet.getPort();
packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);
```



Example: Client and Server

Steps for Client

The client application sends a single datagram packet to the server indicating that the client would like to receive a quote of the moment. The client then waits for the server to send a datagram packet in response.

```
// Create a DatagramSocket without a port number
DatagramSocket socket = new DatagramSocket();

// send a request to the server
byte[] buf = new byte[256];
InetAddress address = InetAddress.getByName(hostname);
DatagramPacket packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);

// get a response to the server over the DatagramSocket
packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);

// form a string from bytes
String receivedString = new String(packet.getData(), 0, packet.getLength());
```

The packet contains the address and port

packet

packet

Stream Socket vs. Datagram Socket

Stream Socket

- A dedicated point-to-point channel between a client and server.
- Use TCP (Transmission Control Protocol) for data transmission.
- Lossless and reliable.
- Sent and received in the same order.

Datagram Socket

- No dedicated point-to-point channel between a client and server.
- Use UDP (User Datagram Protocol) for data transmission.
- May lose data and not 100% reliable.
- Data may not received in the same order as sent.

IP Multicast

- IP multicast is the delivery of a message to a group of destination computers simultaneously in a single transmission from the source over IP network
- Many tasks require a multicast model of communication
  - IP multicast is widely used in networks such as enterprise and multimedia content delivery networks like IPTV
- A **`MulticastSocket`** is a (UDP) **`DatagramSocket`**, with additional capabilities for joining "groups" of other multicast hosts on the internet
- A **multicast group** is specified by a ***multicast IP*** address (**224.0.0.1-239.255.255.255**) and by a standard UDP port number
- All datagram packets have a Time-To-Live (TTL) value
  - TTL is the maximum number of routers that the datagram is allowed to cross; when it reaches the maximum, it is discarded

Joining a Multicast Group

```
// 1. create a multicast group with a multicast IP address
InetAddress groupAdd = InetAddress.getByName("224.xxx.xxx.xxx");
                                                    // any multicast IP

// 2. create a MulticastSocket with a desired port
MulticastSocket s = new MulticastSocket(port);      // any available port

// 3. join the group
s.joinGroup(groupAdd);

// Ready to send DatagramPacket using the group address
// to all subscribing recipients within the TTL range of the packet
```

Example: Broadcasting to Multiple Recipients

Steps for Server

Instead of sending quotes to a specific client that makes a request, the new server now needs to broadcast quotes at a regular interval.

```
byte[] buf = quoteString.getByte();   // string to send in byte

// don't wait for request...just broadcast a quote to a group
InetAddress group = InetAddress.getByName("230.xxx.xxx.0");

DatagramPacket packet = new DatagramPacket(buf, buf.length, group, port);
socket.send(packet);

sleep((long)Math.random() * FIVE_SECONDS);
```

Example: Broadcasting to Multiple Recipients

Steps for Client

The client needs to be modified so that it passively listens for quotes and does so on a **`MulticastSocket`**.

```
// Create a MulticastSocket with the port number and become a member of the group
MulticastSocket socket = new MulticastSocket(port);
InetAddress group = InetAddress.getByName("230.xxx.xxx.0");
socket.joinGroup(group);

// passively receive a quote without requesting
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);

// form a string from bytes
String receivedString = new String(packet.getData(), 0, packet.getLength());

socket.leaveGroup(group);
socket.close();
```

The server can also use a **`MulticastSocket`**. The socket used by the server to send the **`DatagramPacket`** is not important. What's important when broadcasting packets is the ***addressing information*** contained in the **`DatagramPacket`**, and the socket used by the client to listen for it

Resources

- Besides *Core Java* books and *Oracle Java Tutorials* recommended for the subject, some materials covered on this topic can be found in the following book:

Y. Daniel Liang, *Introduction to Java Programming*, Comprehensive Version, 9th ed, Prentice Hall, 2012