

CSCD 330

Network Programming

Winter 2015



Lecture 6

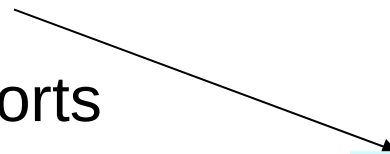
Application Layer – Socket Programming in Java

Reading for Java Client/Server see “Relevant Links”

Some Material in these slides from J.F Kurose and K.W. Ross
All material copyright 1996-2007

Chapter 2: Application Layer

- Principles of network applications
- Web and HTTP
- FTP (Skip)
- Electronic Mail
 - SMTP, POP3, IMAP
- DNS
- P2P Applications – As time permits
- Socket programming with TCP
 - Processes, Addresses and Ports
- Socket programming with UDP



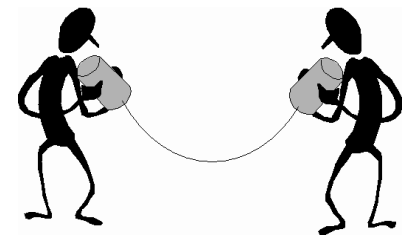
Today



Network Process Communication

Communication Between Hosts

- So, how do you “talk” to another host?
- What do you need to know?
 - **Minimum**
 - Name or Address of host
 - Agree on Protocol
 - **Rely on Other Layers of Network**
 - Lower layers to get message delivered



Addressing in Processes



- To receive messages, **process** running on a **host (machine)** must have **identifier**
 - Host device has unique 32-bit number
IP address
 - Like: **146.187.134.22**
- **Question**
 - Is IP address of host on which process runs enough for identifying the process?

Addressing in Processes

- **NO**, its not enough to have just host IP address
 - As we saw before, **146.187.134.22**
- **E**ach process must have its own identifier
 - **M**any processes can run on same host!!
 - **W**hat is the identifier called?

Addressing in Processes



- Process “identifiers” = Port Numbers
- Port Numbers
 - Standard way to uniquely identify processes
 - 16 bit numbers, from 1 – 65,535
 - Certain port numbers are **reserved** by the operating system
 - Port numbers below 1023 are reserved
 - Above 1023, anyone can use

Reserved Numbers

http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

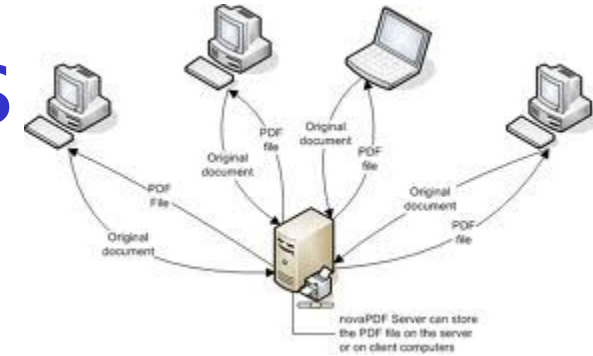
\$ /etc/services

Linux

C:\WINDOWS\system32\drivers\etc\services

Windows

Addressing in Processes



Server Side

Well Known Port Numbers

- IP and port number of the server
- Well-known and advertised so client knows where to find service

Client Side

Ephemeral Port Numbers

- Port number on client side
- Generally allocated automatically by the kernel

Addressing in Processes

- Three Groups of Port Numbers
 - 1. Well Known Port Numbers 1 - 1023
 - IANA assigns port numbers to protocols have been standardized using RFC process
 - 2. Registered Port Numbers 1024 - 49151
 - Non-RFC Server Application can reserve one of these port numbers, through IANA
 - If approved, the IANA will register that port number and assign it to the application

Addressing in Processes

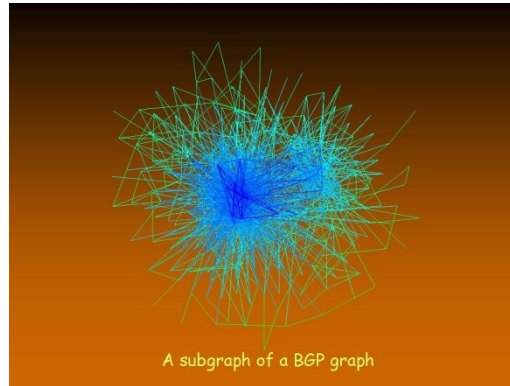
- Three Groups of Port Numbers
 - 3. Ephemeral Port Numbers 49152-65535
 - These ports are neither reserved nor maintained by IANA
- Can be used for any purpose without registration, so they are appropriate for a private protocol, or temporary connection

Wireshark Capture Showing Ephemeral Ports

95	5.170696	72.14.213.100	192.168.0.199	HTTP	HTTP/1.1 200 OK (application/vnd.google.safebrowsing-chunk)
96	5.170713	192.168.0.199	72.14.213.100	TCP	41851 > http [ACK] Seq=1097 Ack=305 Win=6912 Len=0 TSV=946223 TSER=148
97	5.177997	192.168.0.199	72.14.213.100	HTTP	GET /safebrowsing/rd/ChFnb29nLXBoaXNoLXNoYXZhchAAGNemCCDYpggyBVcTAgAD
98	5.206723	72.14.213.100	192.168.0.199	TCP	[TCP segment of a reassembled PDU]
99	5.208667	72.14.213.100	192.168.0.199	HTTP	HTTP/1.1 200 OK (application/vnd.google.safebrowsing-chunk)
100	5.208692	192.168.0.199	72.14.213.100	TCP	41851 > http [ACK] Seq=2183 Ack=1941 Win=12608 Len=0 TSV=946232 TSER=1
101	8.902129	192.168.0.199	146.187.134.22	DNS	Standard query A www.hollywoodinsiders.net
102	8.982675	146.187.134.22	192.168.0.199	DNS	Standard query response CNAME p4-pprr.geo.premiumservices.yahoo.com CN
103	8.982998	192.168.0.199	216.39.62.191	TCP	46471 > http [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=947176 TSER=0 WS=
104	9.101005	216.39.62.191	192.168.0.199	TCP	http > 46471 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=338412
105	9.101066	192.168.0.199	216.39.62.191	TCP	46471 > http [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=947205 TSER=33841209
106	9.101200	192.168.0.199	216.39.62.191	HTTP	GET /holly.html HTTP/1.1
107	9.218219	216.39.62.191	192.168.0.199	TCP	http > 46471 [ACK] Seq=1 Ack=515 Win=6912 Len=0 TSV=3384121058 TSER=94
108	9.228880	216.39.62.191	192.168.0.199	HTTP	HTTP/1.1 304 Not Modified
109	9.228905	192.168.0.199	216.39.62.191	TCP	46471 > http [ACK] Seq=515 Ack=106 Win=5888 Len=0 TSV=947237 TSER=3384
110	9.229193	192.168.0.199	216.39.62.191	TCP	46471 > http [FIN, ACK] Seq=515 Ack=106 Win=5888 Len=0 TSV=947237 TSEF
111	9.229337	216.39.62.191	192.168.0.199	TCP	http > 46471 [FIN, ACK] Seq=106 Ack=515 Win=6912 Len=0 TSV=3384121069
112	9.230257	192.168.0.199	216.39.62.191	TCP	46471 > http [ACK] Seq=515 Ack=107 Win=5888 Len=0 TSV=947237 TSER=3384

Addressing in Processes

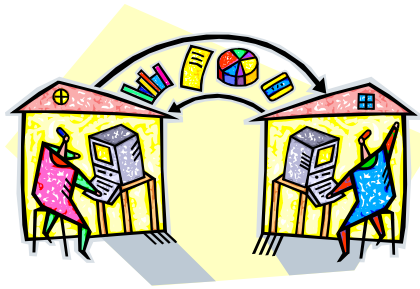
- When you create a socket in Java, you associate it to a port number
- Called “**binding to a port**”
- So, unique identifiers for addressing a process on a given machine consist of:
 - IP address and port number
 - **Example:**
 - To send HTTP message to
gaia.cs.umass.edu web server:
IP address: 128.119.245.12
Port number: 80



Client Server Programming in Java

Goal of Client/Server Programming

- What's the goal for applications in a networked environment?
 - Get information from my machine and move it to another machine or vice-versa
 - Similar to reading and writing files
 - Except ... files exist on a remote machine
 - Network programming in a nutshell !!!



What is a socket?



- A **socket** , software abstraction used to represent the "**terminals**" of a connection between two machines – like electric socket
- For a given connection
 - Socket on each machine and
 - A hypothetical "cable" running between two machines
 - Each "cable" end plugs into socket
 - Physical hardware and cabling between machines unknown and not needed for communication
 - Its an abstraction ... !!!!!

Socket programming

Goal Learn to build client/server applications that communicate using sockets

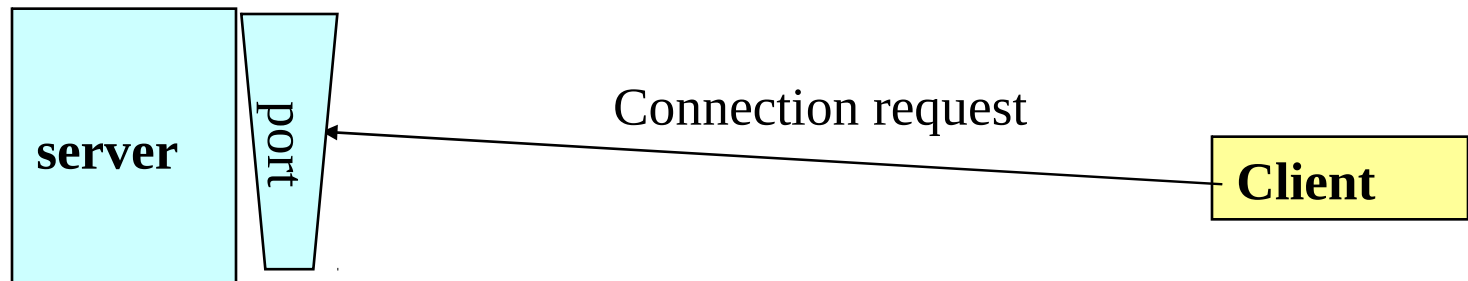
Socket API

- TCP/IP Stack funded by ARPA
 - Resides in Operating System
- Needed an API into stack
- Introduced **BSD 4.1 UNIX, 1981**

- Two types of transport service supported by the socket API:
 1. Unreliable - UDP
 2. Reliable – TCP

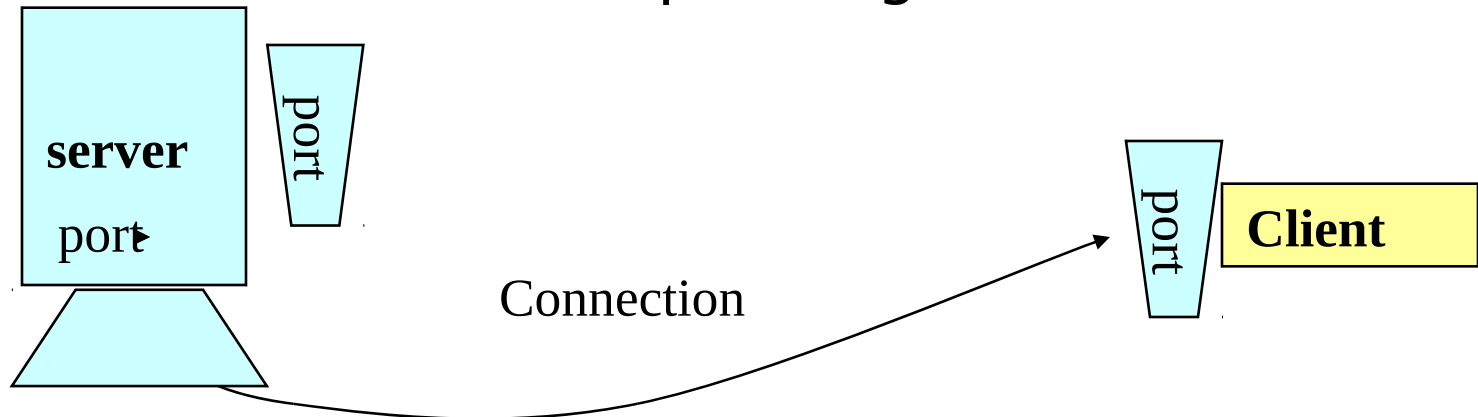
Socket Communication

- A server (program) runs on a specific computer and has a socket that is bound to a specific port
- Server waits and listens to socket for a client to make a connection request



Socket Communication

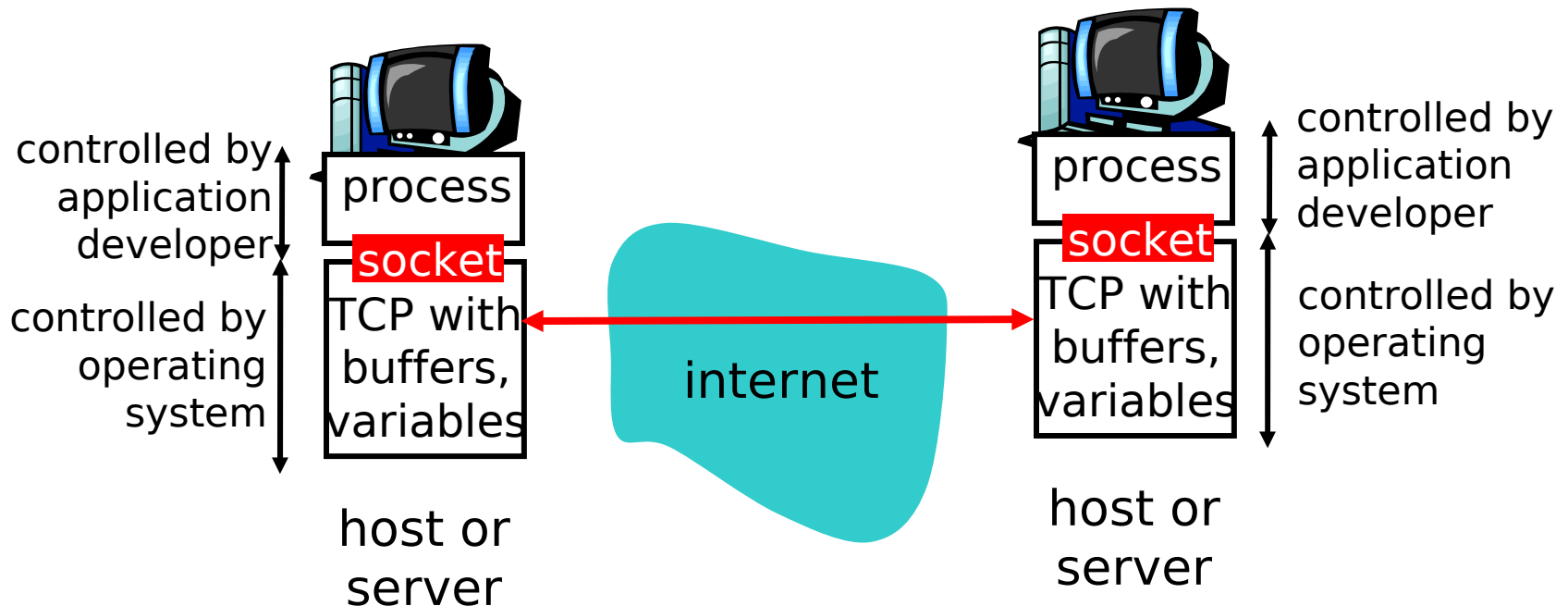
- If everything goes well, server accepts connection
- Upon acceptance, server gets **new socket** bound to a different port
- Needs a new socket (consequently a different port number) so that it can continue to listen to the original socket for connection requests while serving the connected client
- New socket is for responding back to the client



Socket Programming Using TCP

Socket: Door between application process and end-end-transport protocol (UDP or TCP)

TCP Service: reliable transfer of **bytes** from one process to another



Socket Programming Using TCP

Client must first create a connection to server before sending data!!

Client must contact server

- Server process must **first** be running
- Server must have created socket (door) that welcomes client's contact

Client contacts server by

- Creating client-side TCP socket
- Specify IP address, port number of server process
- When **client creates socket**: client TCP establishes connection to server TCP

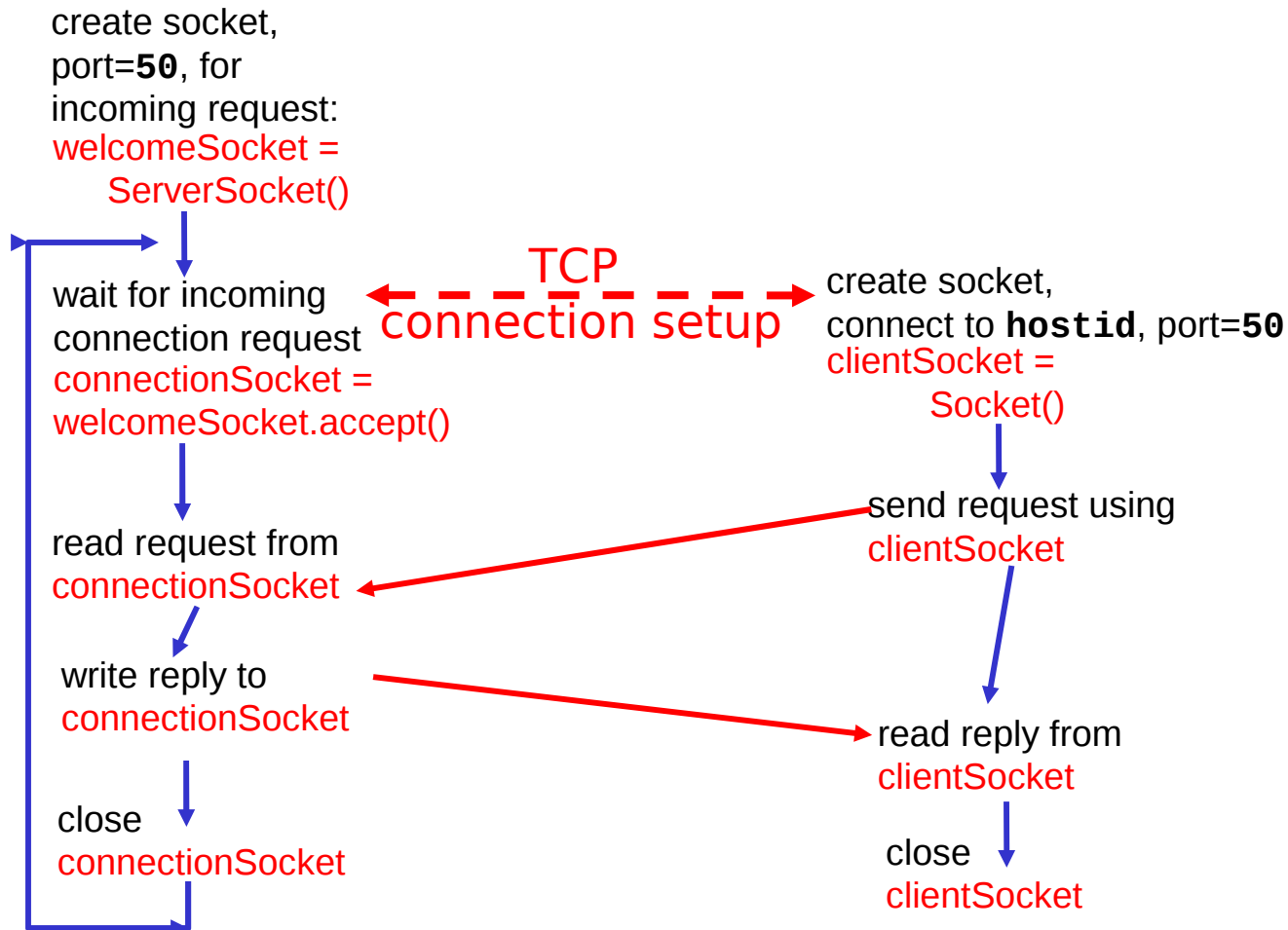
Socket Programming Using TCP

- **When contacted by client,**
 - TCP Server creates new socket
 - Does this for each client
 - Allows server to talk with multiple clients
 - Source port numbers used to distinguish clients

Client/Server socket interaction: TCP

Server (running on `hostid`)

Client



Stream jargon

- A **stream** sequence of characters that flow into or out of a process
- An **input stream** attached to some input source for the process, e.g., keyboard or socket.
- An **output stream** attached to an output source, e.g., monitor or socket.



Socket Programming with TCP

Example: Uppercase Converter

- 1) Client reads line from standard input **inFromUser** stream, sends to server via socket **outToServer** stream
- 2) Server reads line from socket
- 3) Server converts line to uppercase, sends back to client
- 4) Client reads, prints modified line from socket **inFromServer** stream

Client Side Character Converter

- First, we will look at code for the Client side of the Upper Case Converter
- Next slide define some Java Socket characteristics client side

Java Socket programming with TCP

- `java.net.socket`
- `Socket` object is Java representation of TCP connection
 - When socket created on the client side, a connection is opened to a destination
- **Methods**
 - Two most important methods
`getInputStream()` and `getOutputStream()`
 - Return stream objects used to communicate through the socket
 - The `close()` method tells underlying operating system to terminate the connection

Client Uppercase Converter

- Code Logic
 - Accept a string to be converted from the keyboard
 - Attach itself to a Uppercase Server
 - Send it the string through the socket
 - Read the string back from the socket
 - Print the string to the terminal
 - Close the connection

Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {
```

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

```
        String sentence;
        String modifiedSentence;
```

Keyboard

Create
input stream

```
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
```

Create
client socket,
connect to server

```
        Socket clientSocket = new Socket("hostname", 6789);
```

Create
output stream
attached to socket

```
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```

Example: Java client (TCP), cont.

Create
input stream
attached to socket

```
BufferedReader inFromServer =  
    new BufferedReader(new  
        InputStreamReader(clientSocket.getInputStream()));
```

Send line
to server

```
sentence = inFromUser.readLine();  
outToServer.writeBytes(sentence + '\n');
```

**Input from
Keyboard**

Read line
from server

```
modifiedSentence = inFromServer.readLine();  
System.out.println("FROM SERVER: " + modifiedSentence);  
clientSocket.close();
```

```
}  
}
```

Java Server (TCP)

- ServerSocket represents listening TCP connection
 - Once an incoming connection is requested,
 - **ServerSocket** object will return a Socket object representing the connection
- **Methods**
 - Most important method is **accept()**
 - Returns a Socket, connected to a client
 - The **close()** method tells operating system to stop listening for requests on the socket
 - Other Methods are also provided

Server Uppercase Converter

- Code Logic
 - Set up a ServerSocket
 - Listen for a client on the ServerSocket and spawn a new socket for the client
 - Attach input and output streams to new socket
 - Read the string from the socket and convert it to uppercase
 - Send it the uppercase string through the socket
 - Close the connection to the new socket
 - Return to listening for a new client

Example: Java server (TCP)

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

```
class TCPServer {
```

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

```
        String clientSentence;
        String capitalizedSentence;
```

Create
welcoming socket
at port 6789

```
        ServerSocket welcomeSocket = new ServerSocket(6789);
```

Waits, on welcoming
socket for contact
by client

```
        while(true) {
```

```
            Socket connectionSocket = welcomeSocket.accept();
```

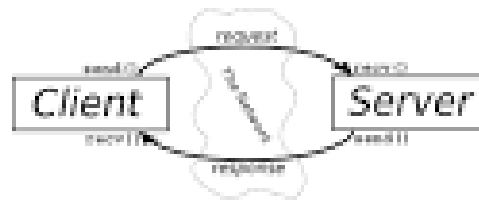
Create input
stream, attached
to socket

```
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()))
```


Example: Java server (TCP), cont

```
    Create output stream, attached to socket }
    DataOutputStream outToClient =
    → new DataOutputStream(connectionSocket.getOutputStream());
    Read in line from socket }
    clientSentence = inFromClient.readLine();
    capitalizedSentence = clientSentence.toUpperCase() + '\n';
    Write out line to socket }
    outToClient.writeBytes(capitalizedSentence);
    connectionSocket.close ();
    }
    }
    }
```

End of while loop,
loop back and wait for
another client connection



Running the Client/Server

Example of Server Creating Socket Object

- Demo a Server that listens on a port for a connection, and creates a “socket” object for talking to client
 - Has Ephemeral port numbers
 - Can use telnet to play the role of a client
 - **Steps:**
 1. Start my Server: HelloServer 5555
 2. telnet localhost 5555
- I am running my server, HelloServer on 5555



Java .net Package

Java Sockets Programming

- The package `java.net` provides support for socket programming (and more).
- Typically you import everything defined in this package with:

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

Classes

InetAddress

Socket

ServerSocket

DatagramSocket

DatagramPacket

InetAddress class

- Static methods you can use to create new InetAddress objects.

`getByName(String host)`

`getAllByName(String host)`

`getLocalHost()`

InetAddress addr =

`InetAddress.getByName("www.ewu.edu");`

- Throws **UnknownHostException**

Sample Code: Lookup.java

- Uses InetAddress class to lookup hostnames found on command line.

> java Lookup www.ewu.edu

www.ewu.edu: 146.187.224.198

> java Lookup www.yahoo.com

www.yahoo.com: 209.131.36.158

Lookup.java Code

```
try {  
  
    InetAddress a = InetAddress.getByName(hostname);  
  
    System.out.println (hostname + ":" + a.getHostAddress());  
  
} catch (UnknownHostException e) {  
  
    System.out.println("No address found for " + hostname);  
  
}
```

Server Classes

- Two classes of socket used for TCP

1. `java.net.ServerSocket` class

- Used by server applications to obtain port and listen for client requests

- **ServerSocket class – some constructors**

`public ServerSocket(int port) throws IOException`

- Attempts to create a server socket bound to specified port. An exception happens if port is already bound by another application

`public ServerSocket(int port, int backlog) throws IOException`

- Similar to previous constructor, backlog parameter specifies how many incoming clients to store in wait queue

Server Classes

- Here are common methods of ServerSocket class:

public Socket accept() throws IOException

- Waits for an incoming client.
- Method blocks until either client connects to server on specified port or socket times out, assuming at time-out value has been set using setSoTimeout() method. Otherwise, this method blocks indefinitely

public void setSoTimeout(int timeout)

- Sets time-out value for how long the server socket waits for a client during accept()

Server and Client Class

2. `java.net.Socket` class is socket

both client and server use to communicate

- Client obtains Socket object by instantiating one
- Server obtains Socket object from return value of `accept()` method

- **Constructor:**

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

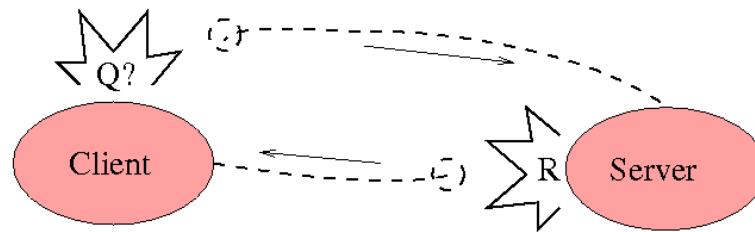
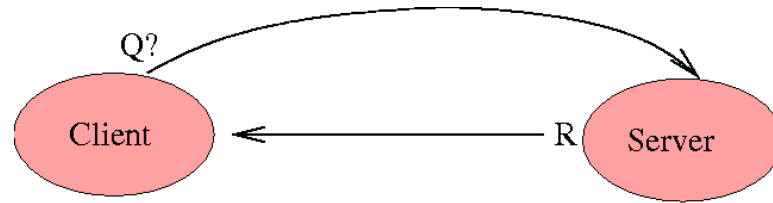
- This method attempts to connect to specified server at specified port
- If this constructor does not throw an exception, connection is successful and client is connected to server

Server and Client Class

- Two most important methods of Socket class are:
 - `public InputStream getInputStream() throws IOException`
 - Returns input stream of socket
 - Input stream is connected to the output stream of the remote socket
 - `public OutputStream getOutputStream() throws IOException`
 - Returns output stream of the socket
 - Output stream is connected to input stream of remote socket

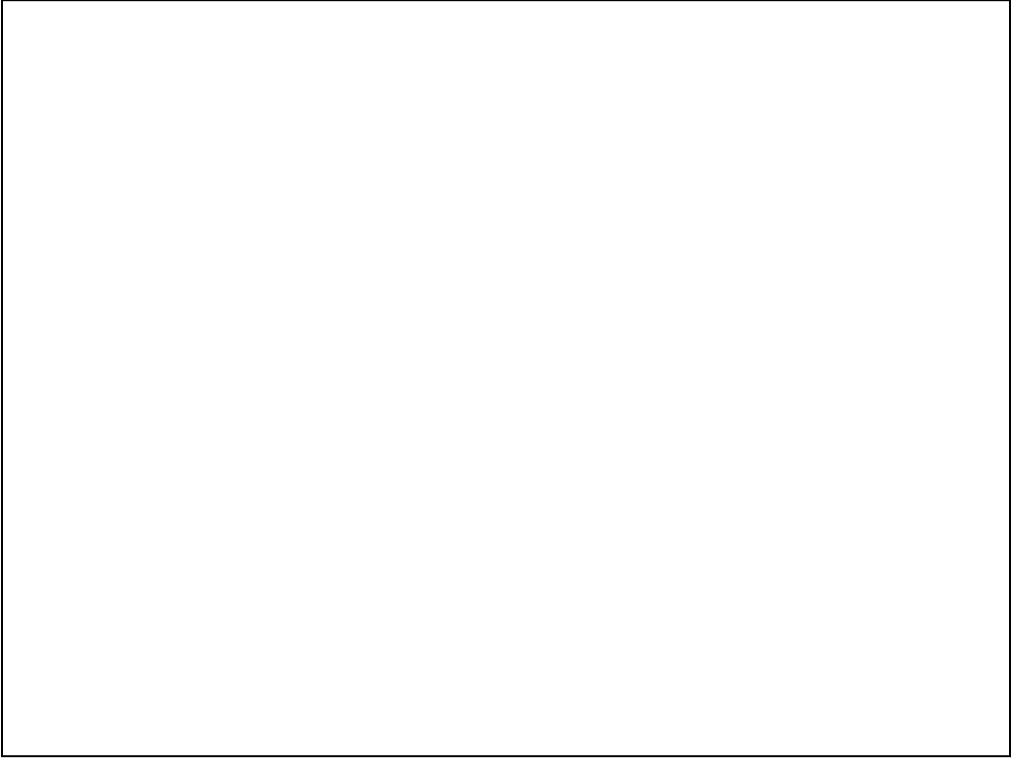
Summary

- Covered briefly, client-server programming in Java environment
- Uses sockets – virtual pipe connections across the Internet
- Allows you to abstract away the details of the network layers (to a large degree)
- Lets you read/write data to processes on local or very remote hosts
- Can get very complicated in functionality
- Will explore more complicated client/server applications in programming assignments

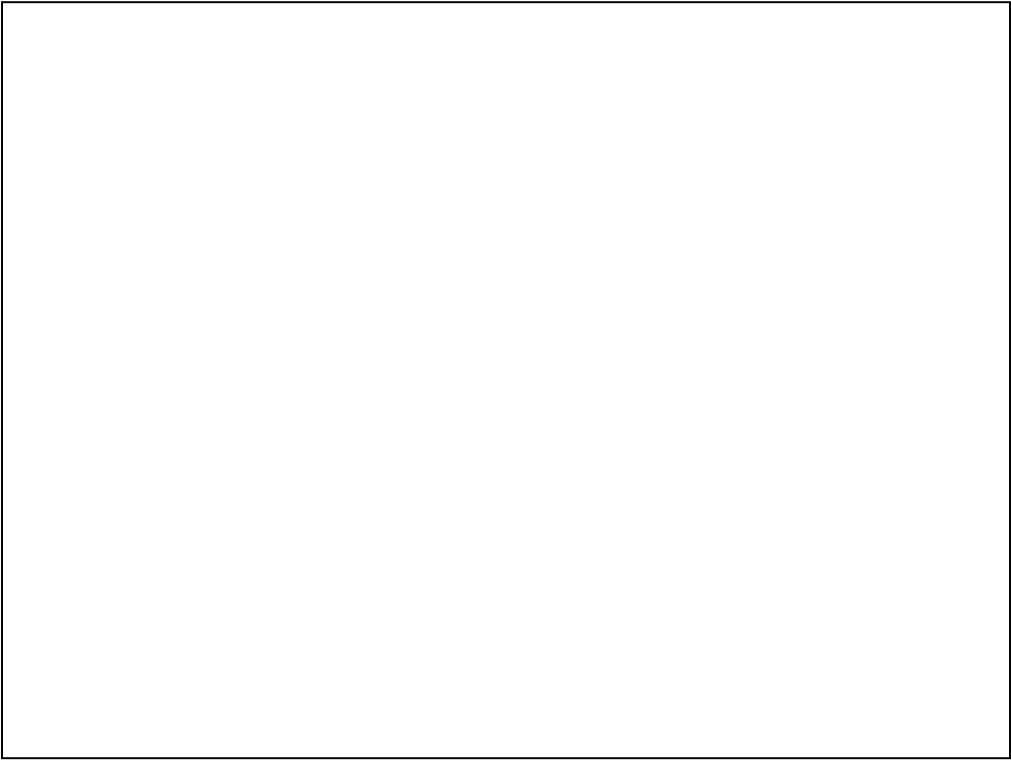


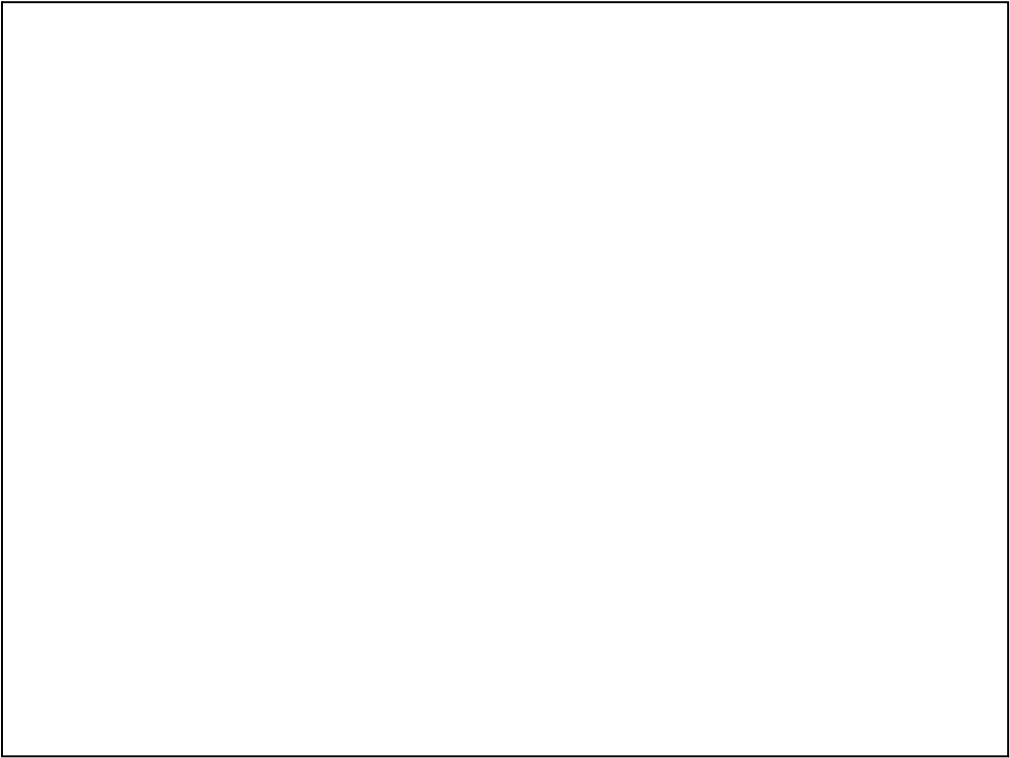
Next Lecture ... Continue with Client/server ...
New Assignment: Simple TCP Client/Server

End





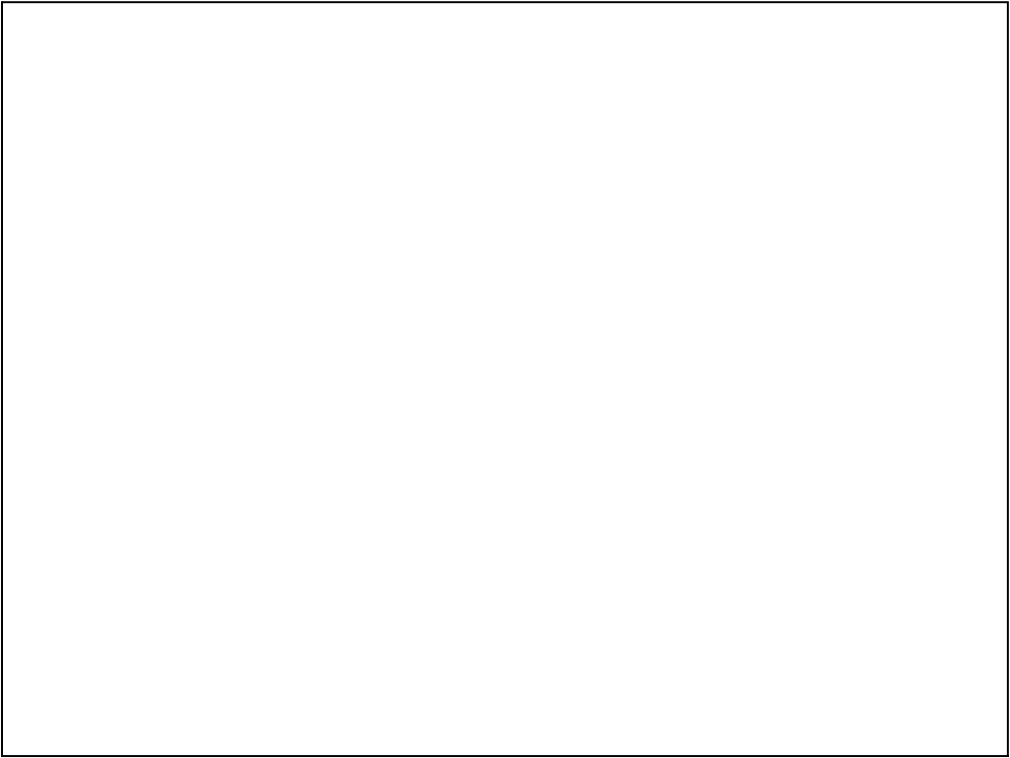


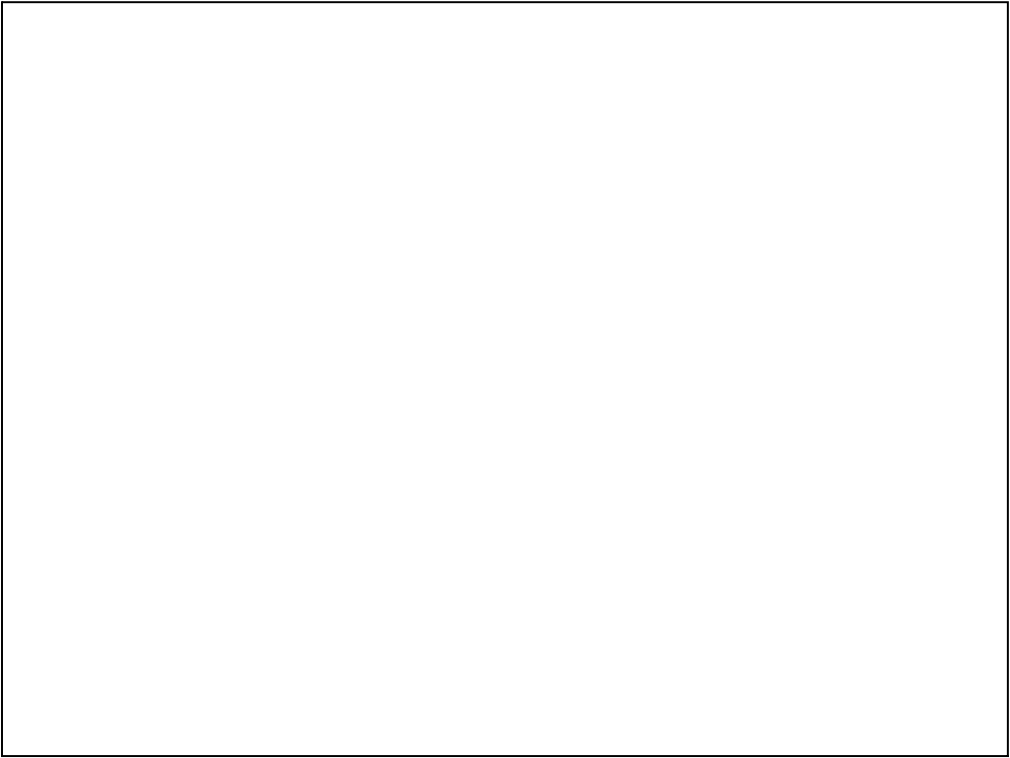


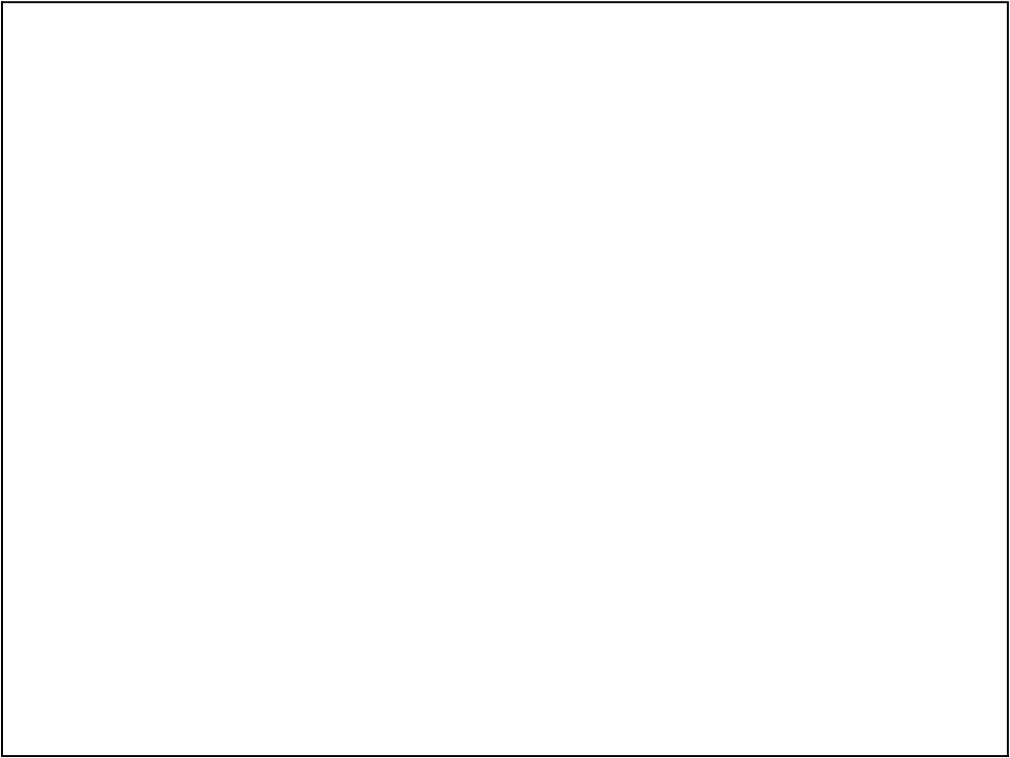


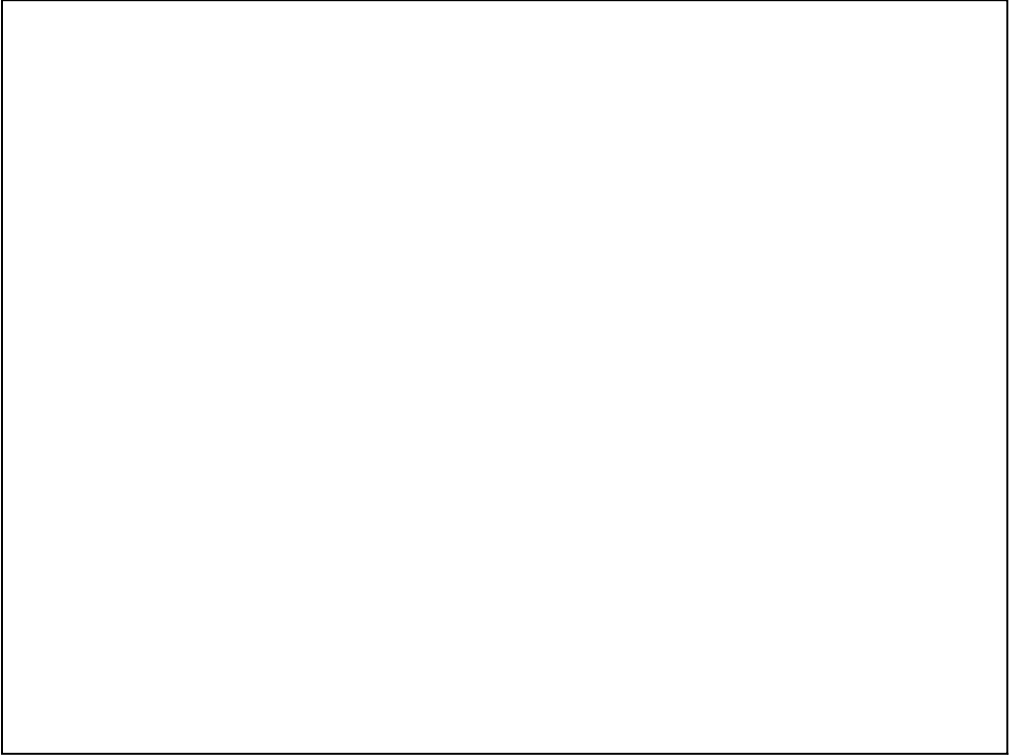












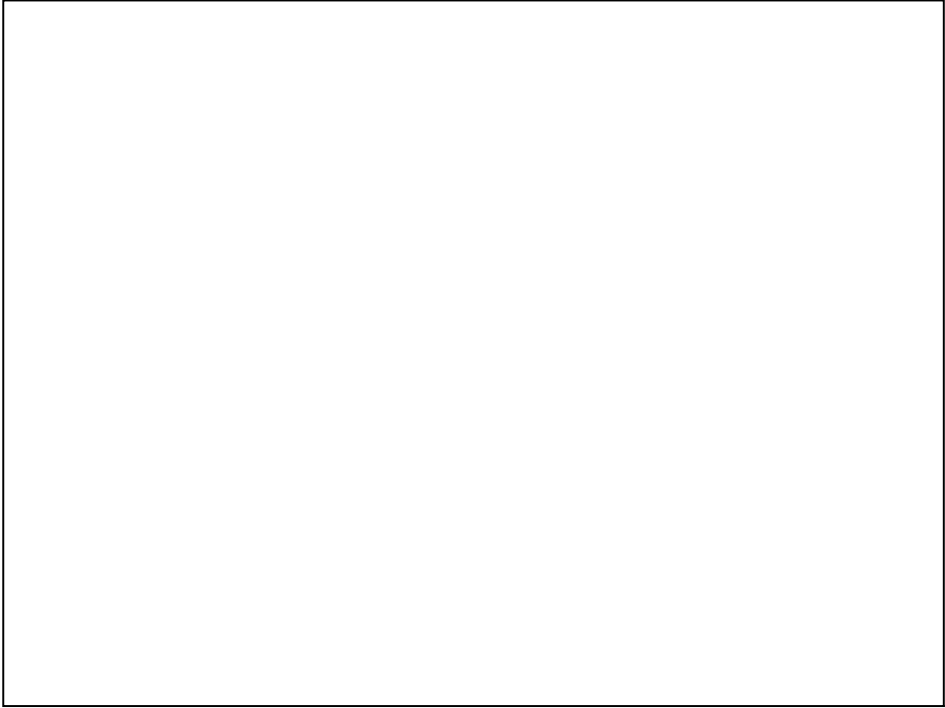






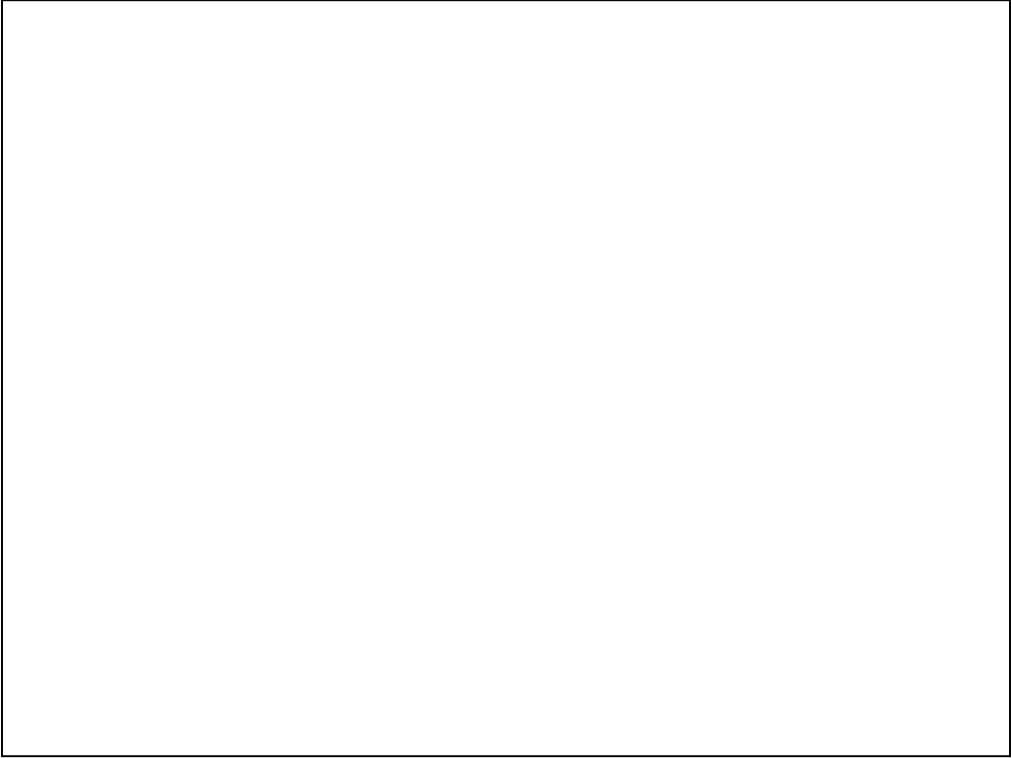






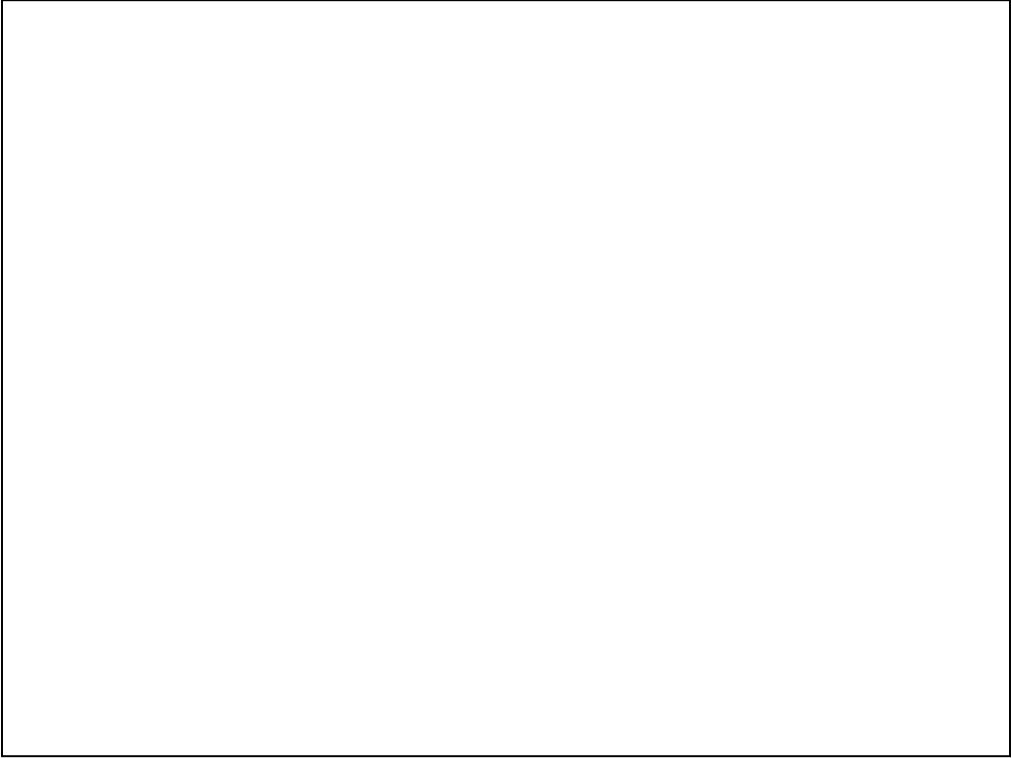














Client Side Character Converter

- First, we will look at code for the Client side of the Upper Case Converter
- Next slide define some Java Socket characteristics client side



Client Uppercase Converter

- Code Logic
 - Accept a string to be converted from the keyboard
 - Attach itself to a Uppercase Server
 - Send it the string through the socket
 - Read the string back from the socket
 - Print the string to the terminal
 - Close the connection





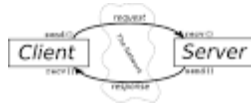


Server Uppercase Converter

- Code Logic
 - Set up a ServerSocket
 - Listen for a client on the ServerSocket and spawn a new socket for the client
 - Attach input and output streams to new socket
 - Read the string from the socket and convert it to uppercase
 - Send it the uppercase string through the socket
 - Close the connection to the new socket
 - Return to listening for a new client







Running the Client/Sever

