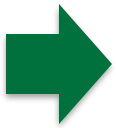
CS 10:
Problem solving via Object Oriented Programming

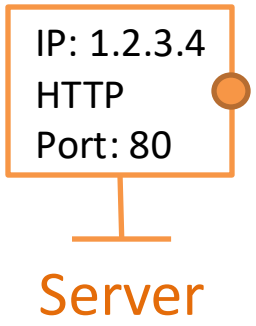
Client/Server

Agenda



1. Sockets
2. Server
3. Multithreaded server
4. Chat server

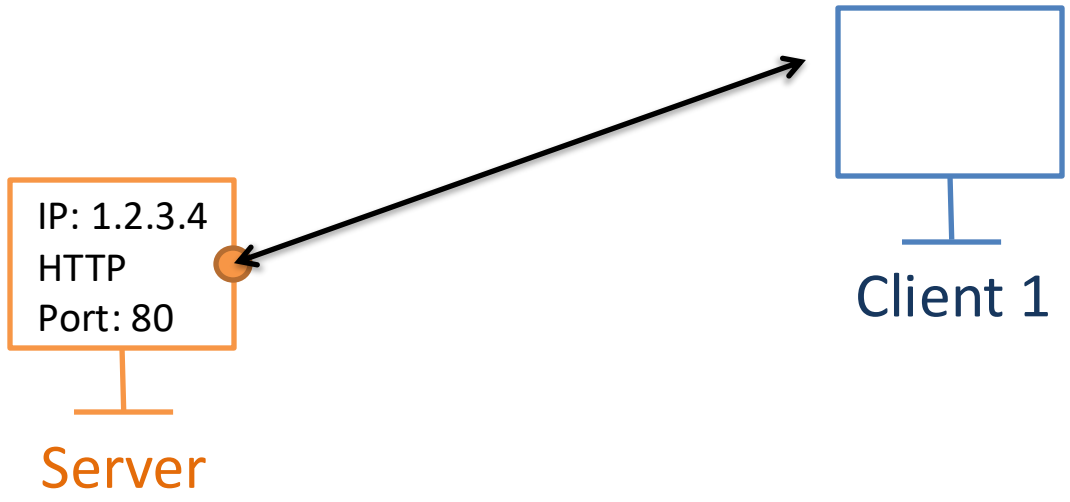
Sockets are a way for computers to communicate



Server is listening on
a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

Sockets are a way for computers to communicate

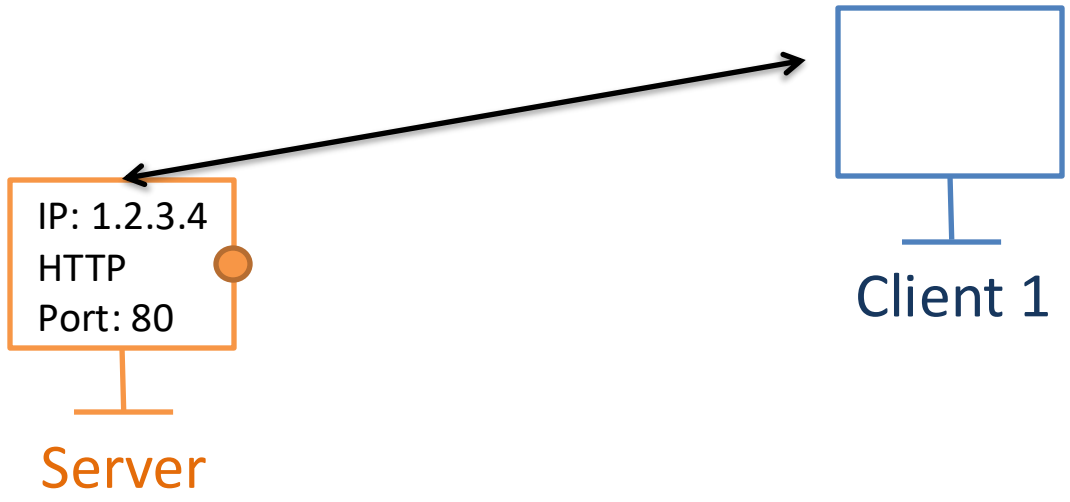


- Client 1 makes connection over socket
- Server receives connection, moves communications to own socket

Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

Sockets are a way for computers to communicate

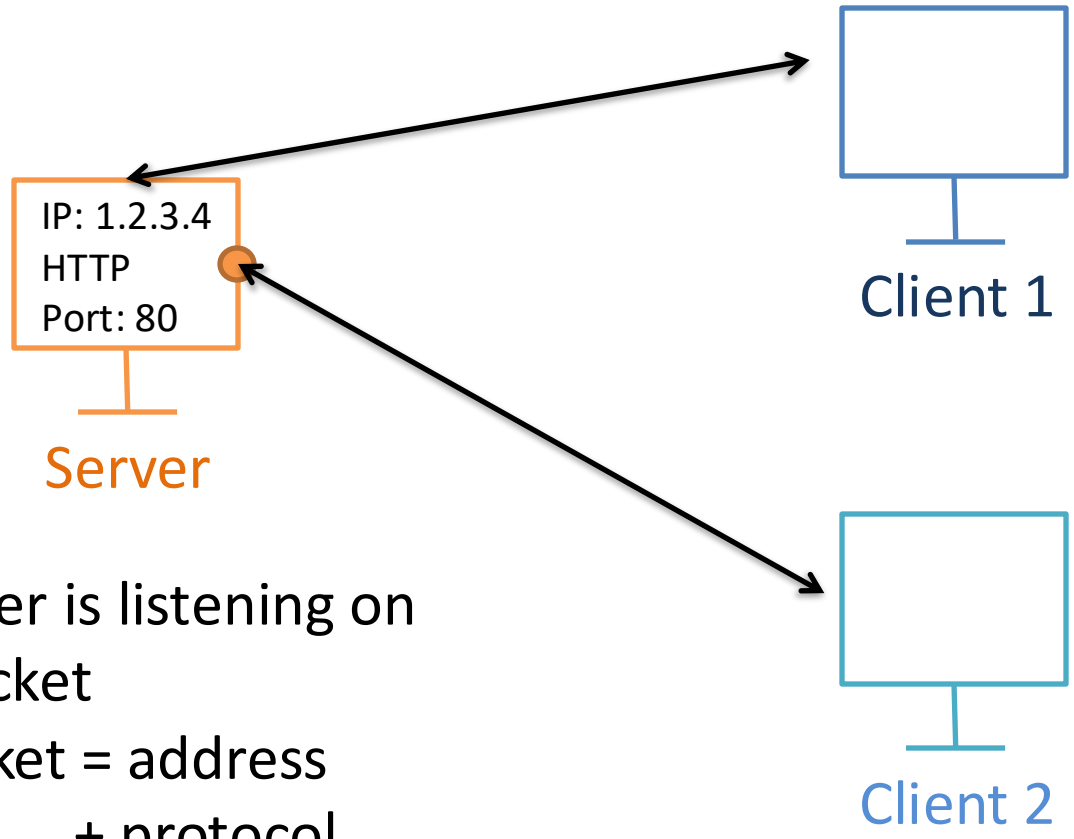


Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

- Client 1 makes connection over socket
- Server receives connection, moves communications to own socket
- Server returns to listening
- Server talking to Client 1 and ready for others

Sockets are a way for computers to communicate

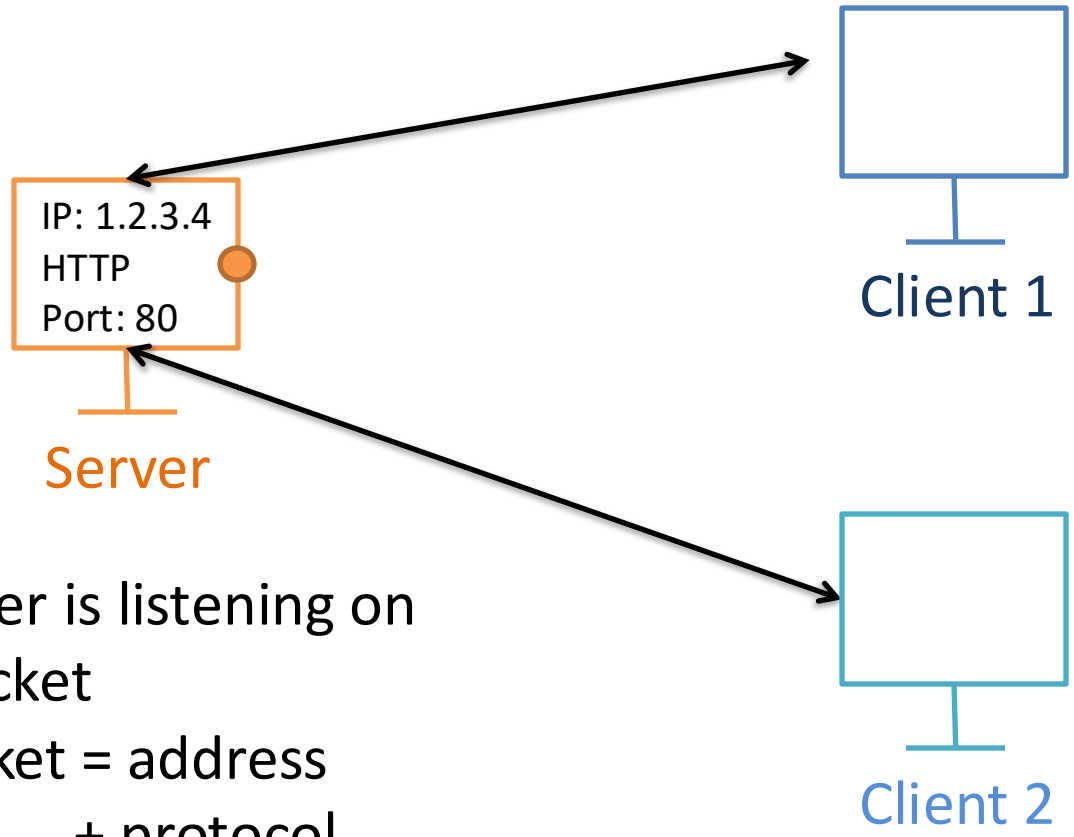


- Client 2 makes connection over socket

Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

Sockets are a way for computers to communicate



Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

- Client 2 makes connection over socket
- Server receives connection, moves communications to own socket
- Server returns to listening
- Server talking to client 1 and 2 ready for others

Agenda

1. Sockets
2. Server
3. Multithreaded server
4. Chat server

DEMO HelloServer.java: create our own server that listens for clients to connect

HelloServer.java

Run HelloServer.java

Fom terminal type “telnet localhost 4242”

Quit telnet session with Control +] then type “quit”

Try connecting from multiple terminals

We can create our own server that will listen for clients to connect and respond

HelloServer.java

Create new *ServerSocket* listening on port 4242
Port chosen because nothing else there

IP: localhost
HTTP
Port: 4242

Server

```
12 public class HelloServer {
13     public static void main(String[] args) throws IOException {
14         // Listen on a server socket for a connection
15         System.out.println("waiting for someone to connect");
16         ServerSocket listen = new ServerSocket(4242);
17         // When someone connects, create a specific socket for them
18         Socket sock = listen.accept();
19         System.out.println("someone connected");
20
21         // Now talk with them
22         PrintWriter out = new PrintWriter(sock.getOutputStream(), true);
23         BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
24         out.println("who is it?");
25         String line;
26         while ((line = in.readLine()) != null) {
27             System.out.println("received:" + line);
28             out.println("hi " + line + "!  anybody else there?");
29         }
30         System.out.println("client hung up");
31
32         // Clean up shop
33         out.close();
34         in.close();
35         sock.close();
36         listen.close();
37     }
38 }
```

Pause here until someone connects, then create Socket *sock* for them

- Create output writer and input reader using *sock*
- Send output to whomever connected

Close up
- Reader and writer
- Sockets

Read input from client until client hangs up (connection lost)
in.readLine() is null on hang up

This code can only handle one connection at a time

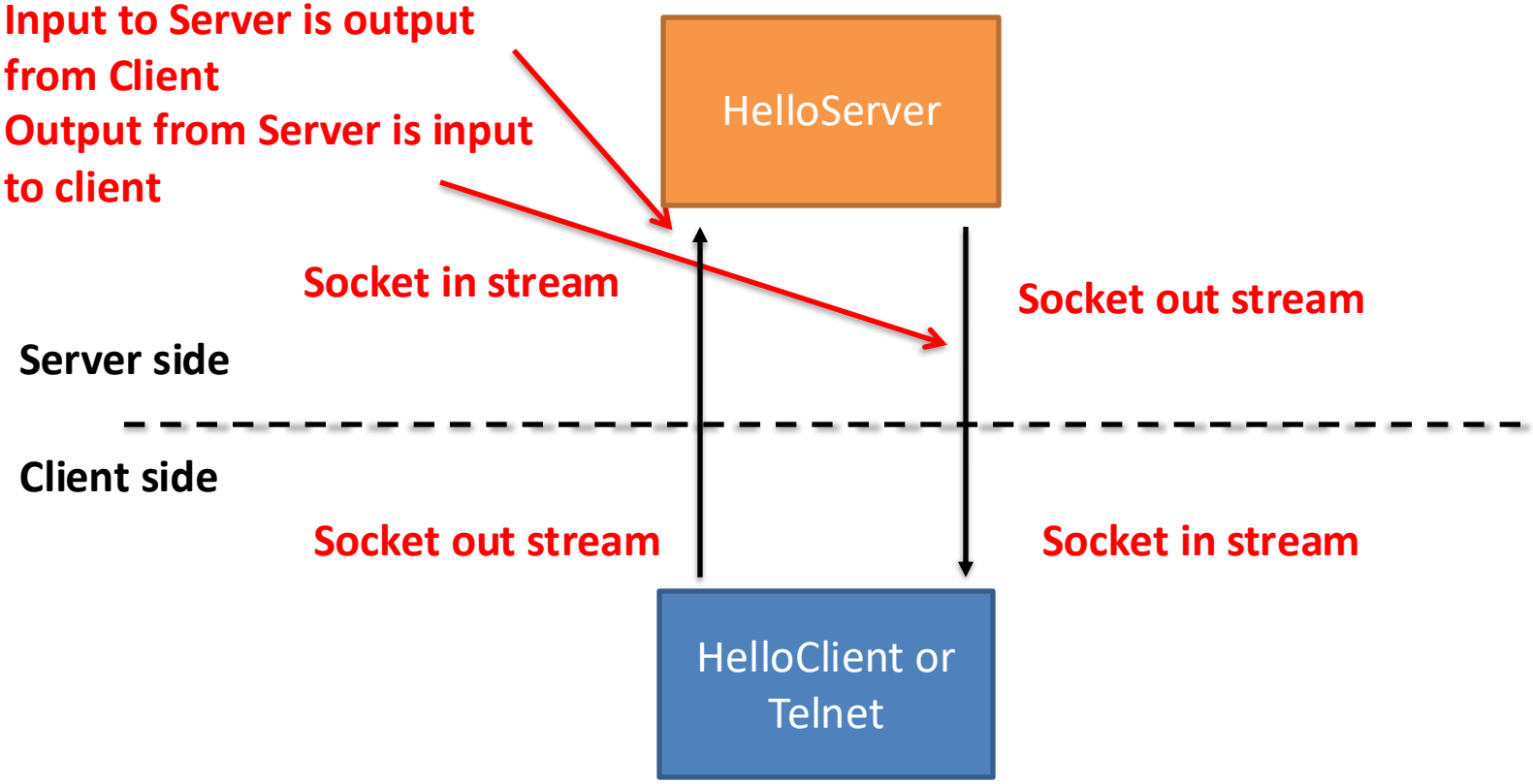
We can create our own client too

HelloServer.java and HelloClient.java

What is input and what is output is relative to each computer

- **Input to Server is output from Client**
- **Output from Server is input to client**

Code for HelloServer on last slide



DEMO HelloClient.java: our Client that talks to our Server

HelloClient.java

Run HelloClient.java (waits for Server to come up)

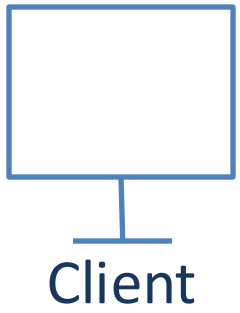
Run HelloServer.java

Our Client talks to our Server

HelloClient.java

```
11 public class HelloClient {
12     public static void main(String[] args) throws Exception {
13         String host = "localhost"; //"localhost" or something like "129.170.212.159"
14         int port = 4242;
15         int connectionDelay = 5000; //in milisecs, 5000 = 5 seconds
16         Scanner console = new Scanner(System.in);
17
18         // Open the socket with the server, and then the writer and reader
19         Socket sock = null;
20         boolean connected = false;
21         System.out.println("connecting...");
22         while (!connected) {
23             try {
24                 //try to connect to server, throws error if server not up (which we catch)
25                 sock = new Socket(host,port);
26                 connected = true;
27             }
28             catch (Exception e) {
29                 //server not up, wait connectionDelay/1000 seconds and try again
30                 System.out.println("\t server not ready, trying again in " + connectionDelay/1000 +
31                 Thread.sleep(connectionDelay); //wait
32             }
33         }
34
35         //set up input and output over socket
36         PrintWriter out = new PrintWriter(sock.getOutputStream(), true);
37         BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
38         System.out.println("...connected");
39
40         // Now listen and respond
41         String line;
42         while ((line = in.readLine()) != null) {
43             // Output what you read
44             System.out.println(line);
45
46             // Get user input from keyboard to write to the open socket (sends to server)
47             String name = console.nextLine();
48             out.println(name);
49         }
50         System.out.println("server hung up");
51
52         // Clean up shop
53         console.close();
```

Setup scanner to read client's keyboard



Loop until Server answers

Create Socket *sock* on same port as Server (4242)

sock will throw exception if Server not up, try every 5 seconds until it is up

Got Server connection, setup reader and writer

Output to console what the Server said

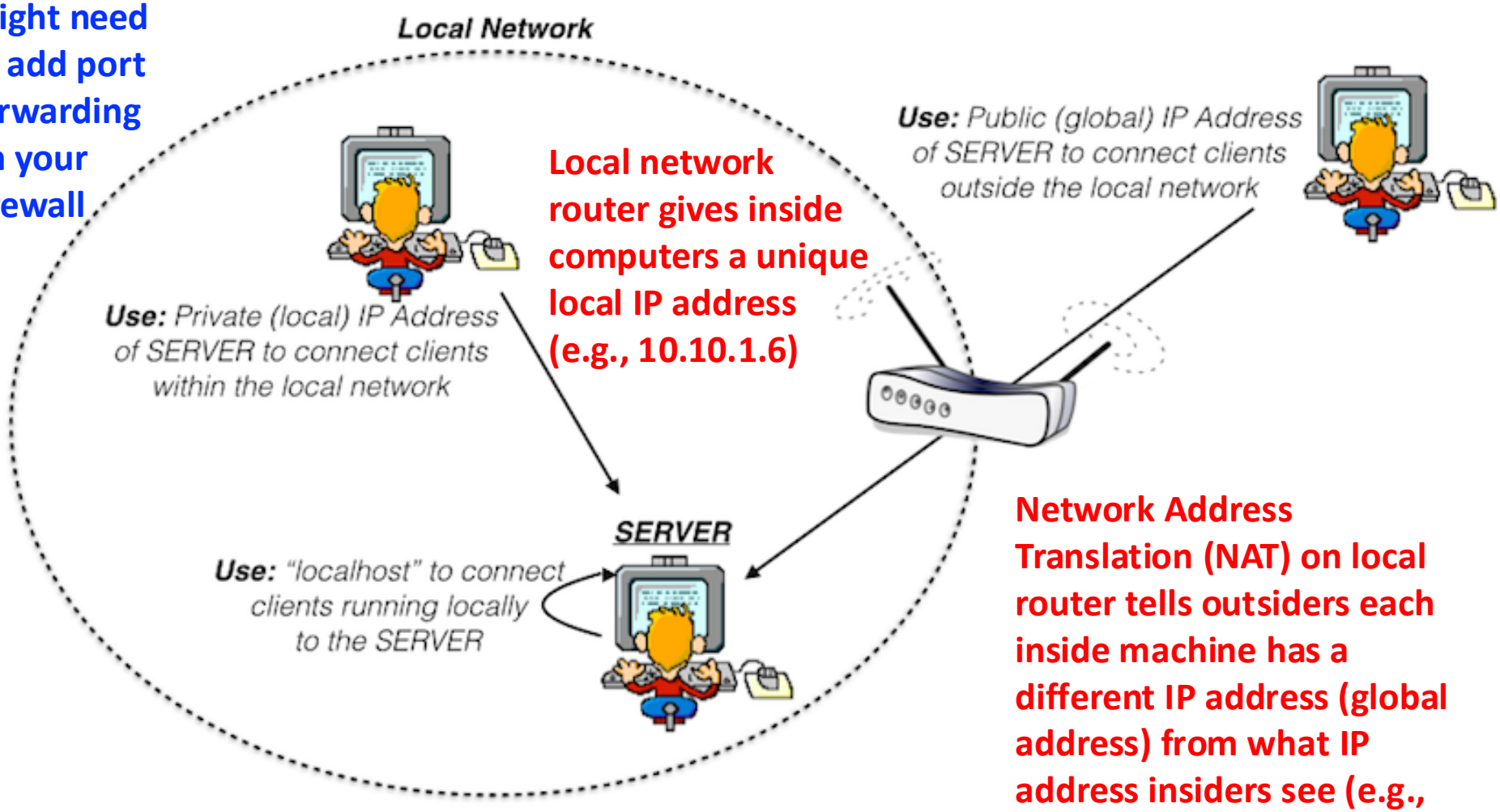
Get input from scanner and send to Server

- If Server hangs up, don't know it until you press enter on keyboard. Why?
- *console.nextLine()* "blocks" execution

Friends can connect to your server if they connect to the right IP address

Run MyIPAdressHelper.java to get your address, edit HelloClient.java

Might need to add port forwarding on your firewall



Network Address Translation (NAT) on local router tells outsiders each inside machine has a different IP address (global address) from what IP address insiders see (e.g., 129.170.212.159)

DEMO: Connecting from another machine

HelloServer.java and HelloClient.java

- Run MyIPAddressHelper on server to get IP
- Start HelloServer.java on server
- Edit HelloClient.java to change localhost to server IP address
- Run HelloClient on client machines and make connection
- Connect from student machine?

Agenda

1. Sockets
2. Server
3. Multithreaded server
4. Chat server

Currently our server can only handle one client at a time, use Threads for more users

Use Java's Thread mechanism to overcome single client issue

- **Want multiple "concurrent" users**
- **Trick: give each user its own socket**
- **Use *threads* that run concurrently with main process (more on threads next class)**
- **Threads are lighter processes than main program**
- **Threads inherit from Java's Thread class**

```
public class MainApp {
   void main() {
         MyThread t = new MyThread();
         t.start();
```

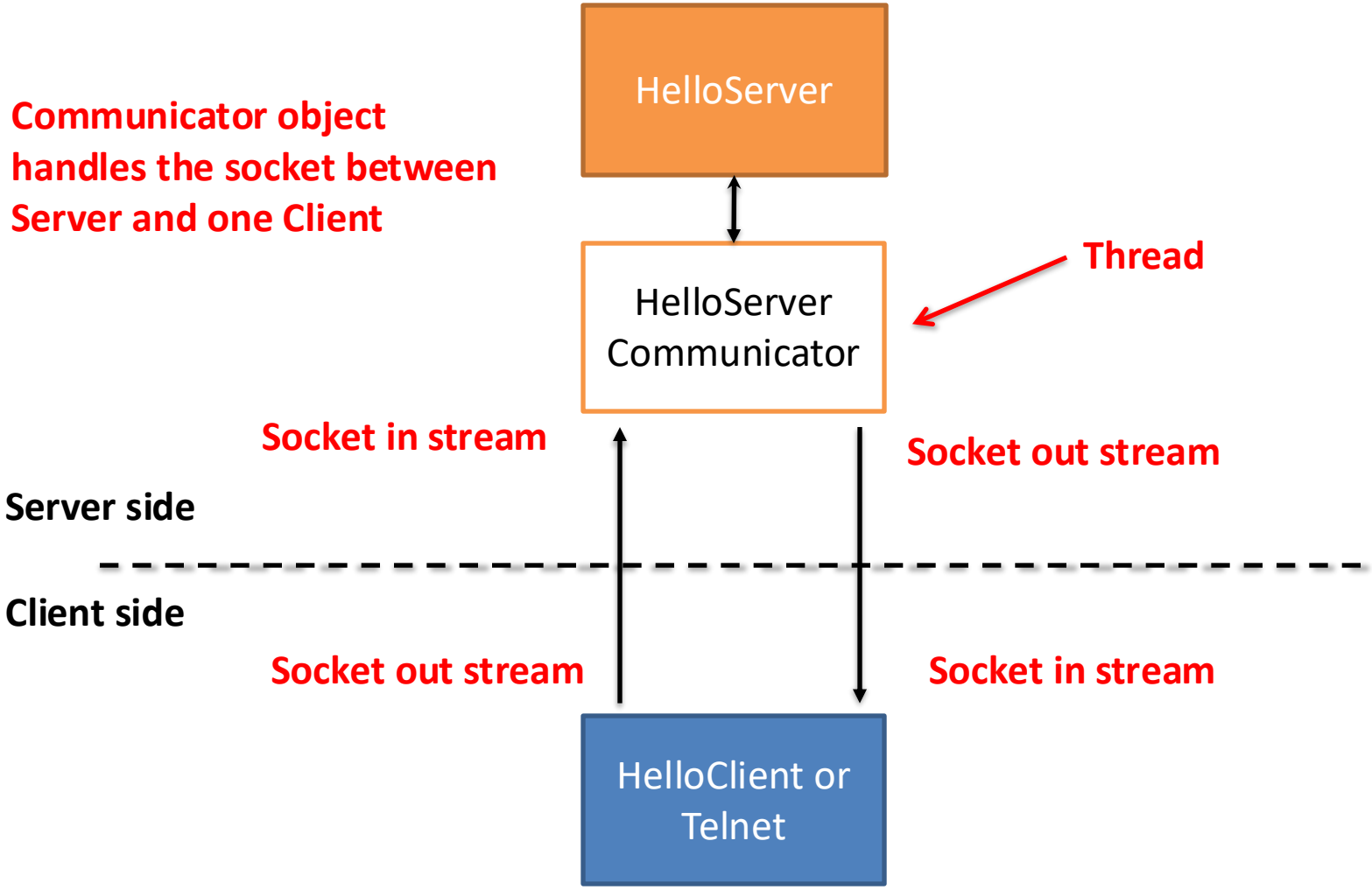
- **Instantiate object of type that extends Thread**
- **Call start on thread object to start thread process running "concurrently" with main process**

```
public class MyThread extends Thread {
   void run() {
      …
   }
```

- **Class extends Thread**
- **Threads begin running at *run()* method, not *main()***
- **Each thread responsible for handling one client**

We can create a "Communicator" on a separate thread for each Client connection

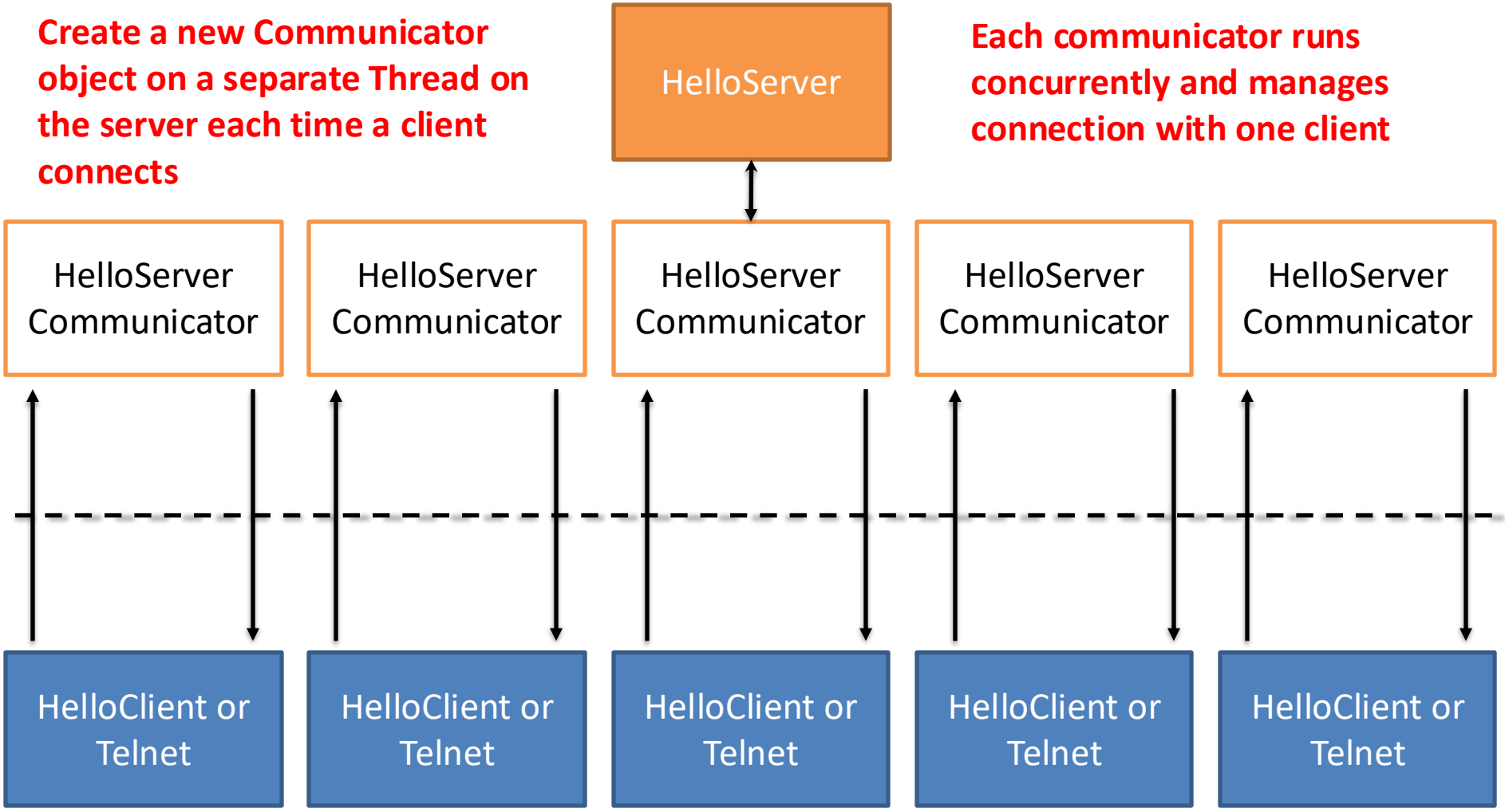
Use Java's Thread mechanism to overcome single client issue



By using Threads, one Server can handle multiple concurrent Clients

Use Java's Thread mechanism to overcome single client issue

Create a new Communicator object on a separate Thread on the server each time a client connects



Each communicator runs concurrently and manages connection with one client

DEMO HelloMultithreadedServer.java: handle multiple Clients concurrently

HelloMultithreadedServer.java

- Starts new thread with new HelloServerCommunicator on each connection

HelloServerCommunicator.java

- Extends Thread
- Override run
- Tracks thread ID
- Otherwise the same as single threaded version

Run HelloMultithreadedServer.java with multiple students connecting (after editing HelloClient.java with IP address)

By using Threads, one Server can handle multiple concurrent Clients

HelloMultithreadedServer.java

```
14 public class HelloMultithreadedServer {
15     private ServerSocket listen;// where clients initially connect
16
17     public HelloMultithreadedServer(ServerSocket listen) {
18         this.listen = listen;
19     }
20
21     /**
22      * Listens to listen and fires off new communicators to handle the clients
23      */
24     public void getConnections() throws IOException {
25         System.out.println("waiting for someone to connect");
26
27         // Just keep accepting connections and firing off new threads to handle them.
28         int num = 0;
29         while (true) {
30             //listen.accept in next line blocks until a connection is made
31             HelloServerCommunicator client = new HelloServerCommunicator(listen.accept(), num++);
32             client.setDaemon(true); // handler thread terminates when main thread does
33             client.start(); //start new thread running
34         }
35
36     }
37
38     public static void main(String[] args) throws IOException {
39         new HelloMultithreadedServer(new ServerSocket(4242)).getConnections();
40     }
41 }
42
```

Create a ServerSocket to listen for incoming connections

- *num* keeps track of how many connections have been made
- Loop forever
- Put new connections on their own Thread with Communicator

setDaemon(true) means stop this Thread when the main Thread ends

Block until Client connects, then return new Socket

start() causes a Thread to begin running in Thread Object's *run()* method

- Pass new ServerSocket on port 4242 to constructor
- Then call *getConnections()*

Big idea: start a new thread whenever a client connects so this thread can go back to listening for new clients

HelloServerCommunicator runs on its own Thread, handles one Client's connection

HelloServerCommunicator.java

```
 9 public class HelloServerCommunicator extends Thread {
10     private Socket sock = null;      // to talk with client
11     private int id;                  // for marking the messages (just for clarity in reading conso
12
13     public HelloServerCommunicator(Socket sock, int id) {
14         this.sock = sock;
15         this.id = id;
16     }
17
18     /**
19      * The body of the thread is basically the same as what we had in main() of the single-threade
20      */
21     public void run() {
22         // Smother any exceptions, to match the signature of Thread.run()
23         try {
24             System.out.println("#" + id + " connected");
25
26             // Communication channel
27             BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
28             PrintWriter out = new PrintWriter(sock.getOutputStream(), true);
29
30             // Talk
31             out.println("who is it?");
32             String line;
33             while ((line = in.readLine()) != null) {
34                 System.out.println("#" + id + " received:" + line);
35                 out.println("hi " + line + "!  anybody else there?");
36             }
37             System.out.println("#" + id + " hung up");
38
39             // Clean up
40             out.close();
41             in.close();
42             sock.close();
43         }
44         catch (IOException e) {
45             e.printStackTrace();
46         }
47     }
48 }
```

- Extends Thread
- When *start()* called on Thread, it calls Thread's *run()* method

Save socket to talk to Client and keep id for convenience

Print id number so we can track who is communicating

Setup *run()* to function the same as single-threaded version

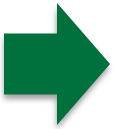
Now this Thread runs independently of other Threads

Handles one Client connection

Stops when HelloMultithreadedServer stops (daemon true)

Agenda

1. Sockets
2. Server
3. Multithreaded server
4. Chat server



DEMO: Chat application

ChatServer.java and ChatClient.java

- Run MyIPAddressHelper on server to get IP
- Start ChatSever.java on server
- Edit ChatClient.java to change localhost to server IP address (in main())
- Run ChatClient.java to connect to ChatServer
- Run ChatClient.java from student machine?

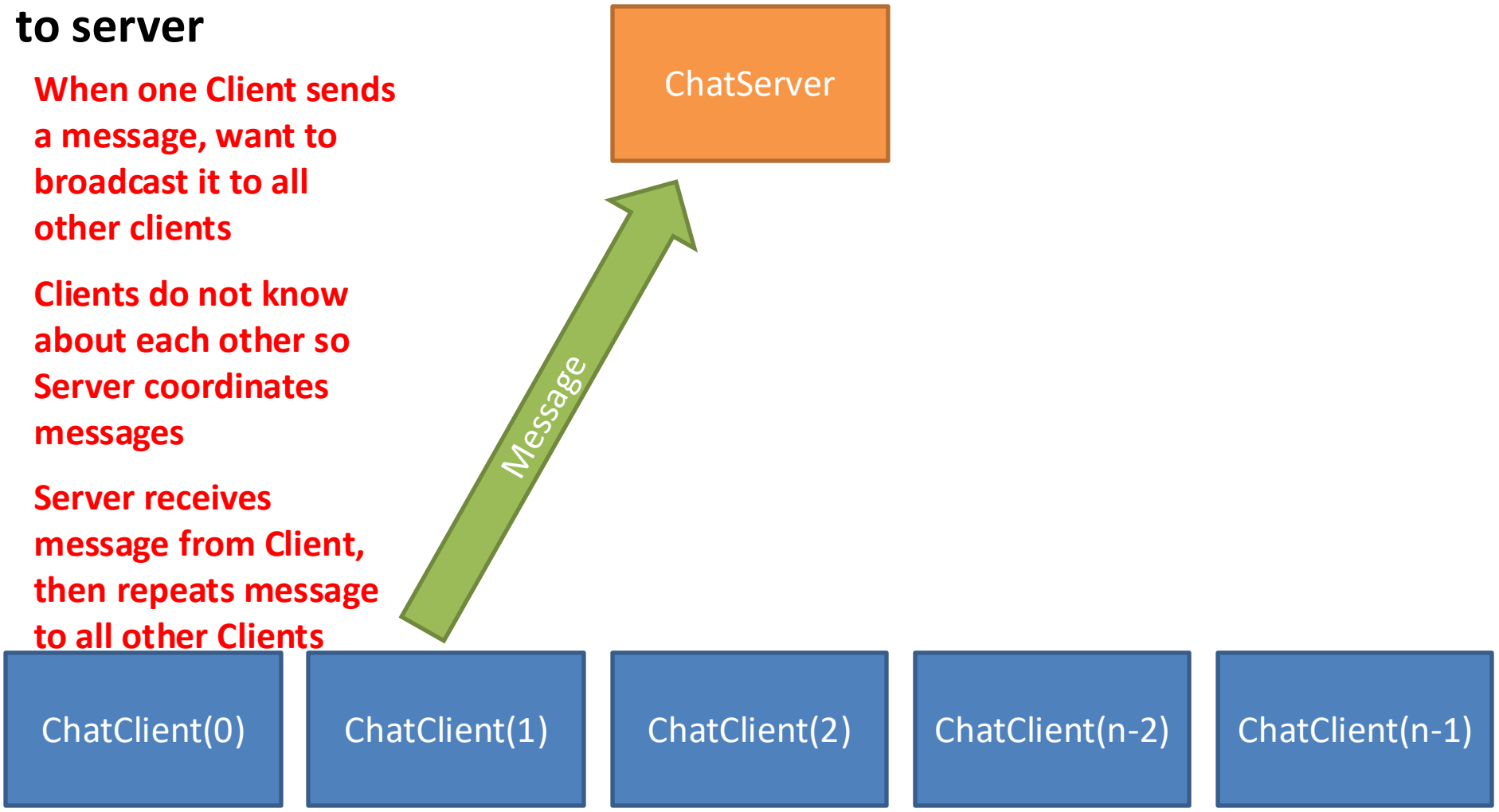
Goal: Chat server allows communication between multiple clients

Client sends message to server

When one Client sends a message, want to broadcast it to all other clients

Clients do not know about each other so Server coordinates messages

Server receives message from Client, then repeats message to all other Clients

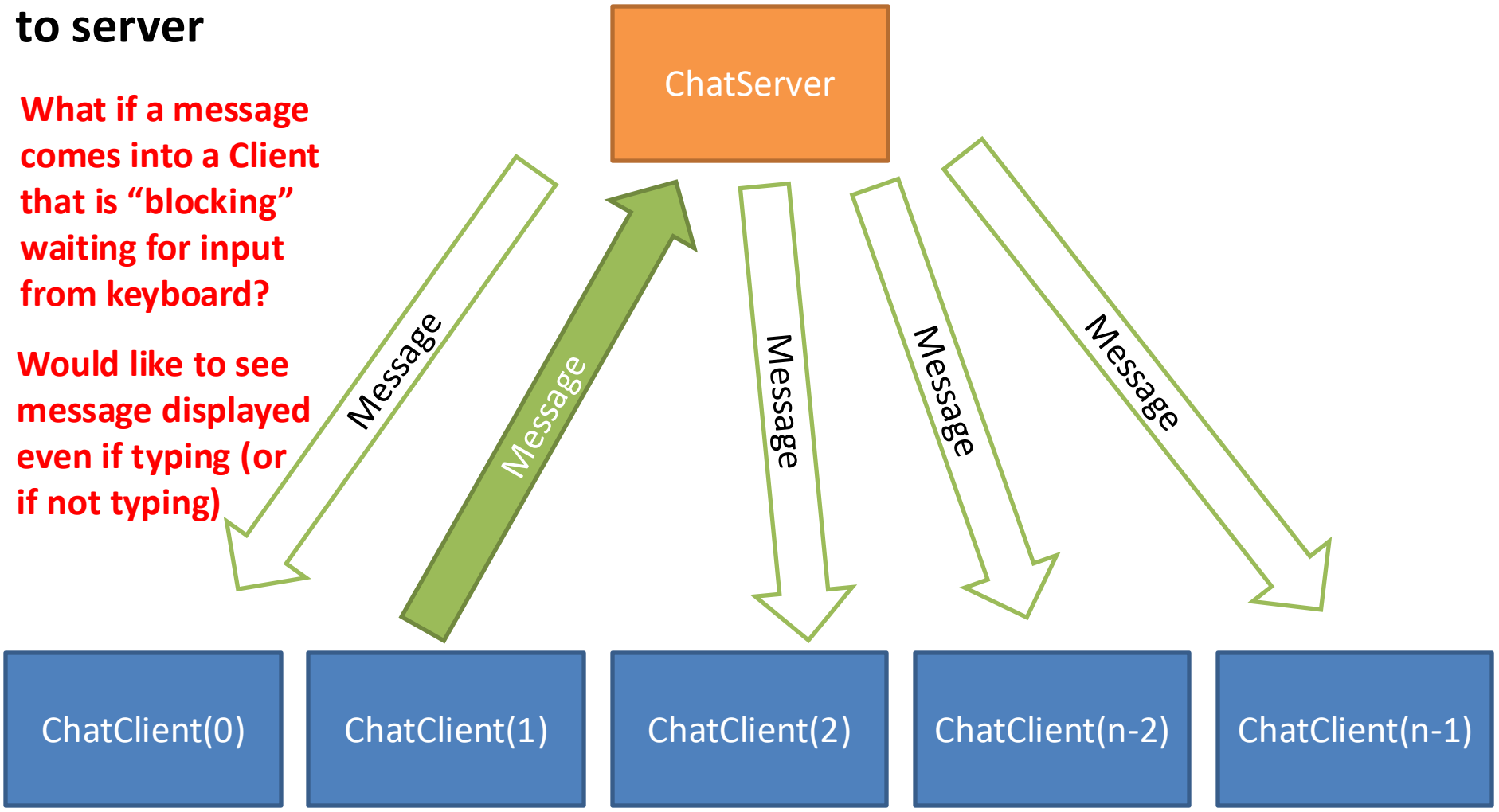


Goal: Chat server allows communication between multiple clients

Client sends message to server

What if a message comes into a Client that is "blocking" waiting for input from keyboard?

Would like to see message displayed even if typing (or if not typing)

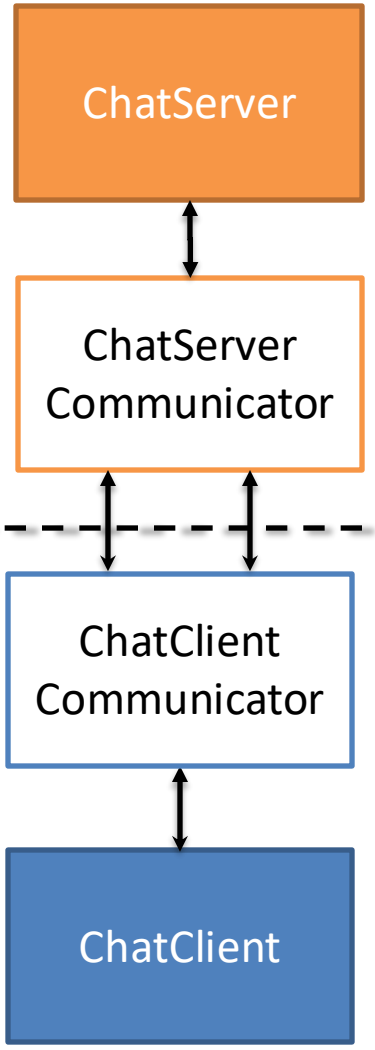


Client listens for keyboard on main thread creates Communicator on second thread

Client

Server uses Communicator, one for each client

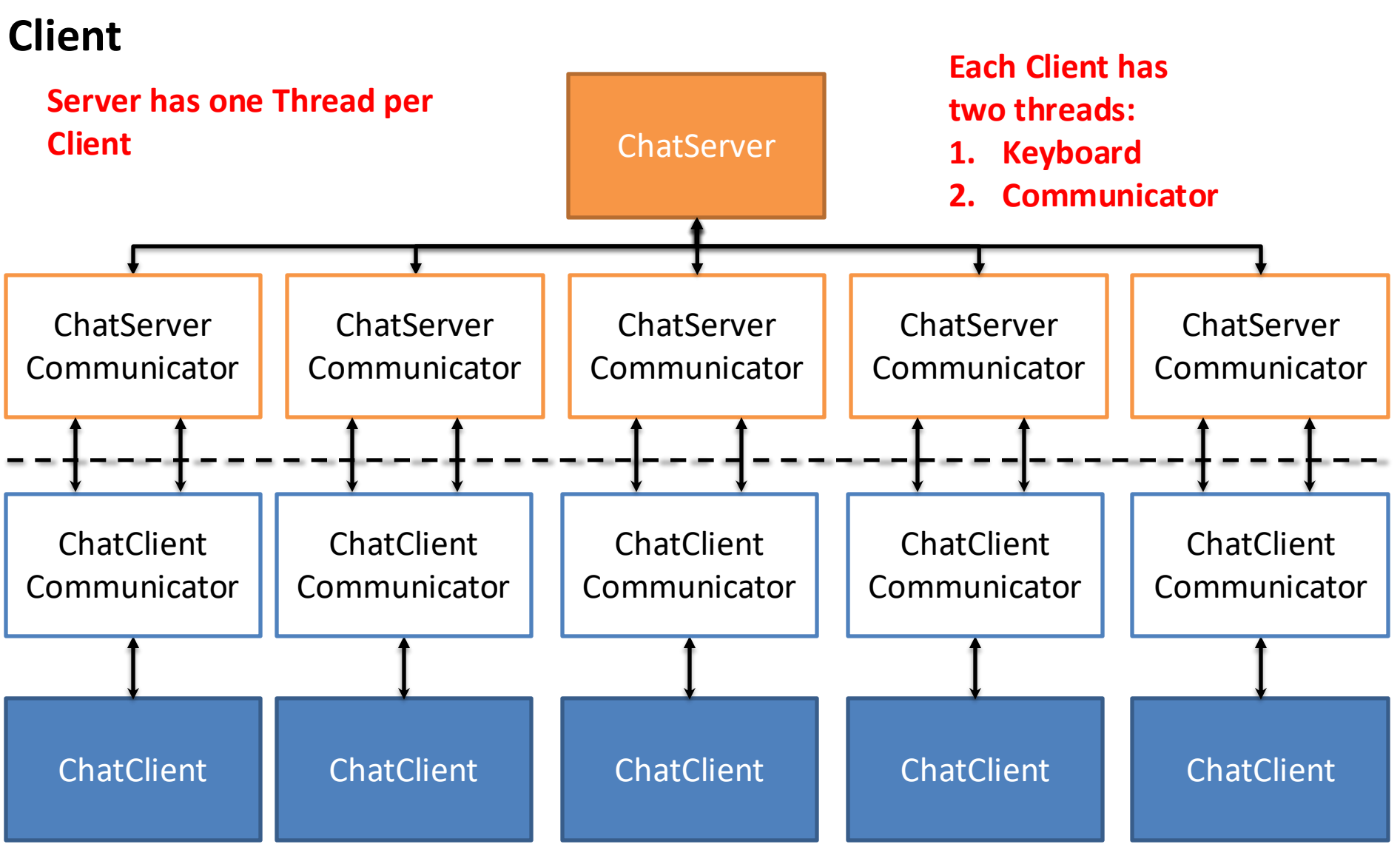
Both Server and Client side are now multi-threaded



Client uses two threads:

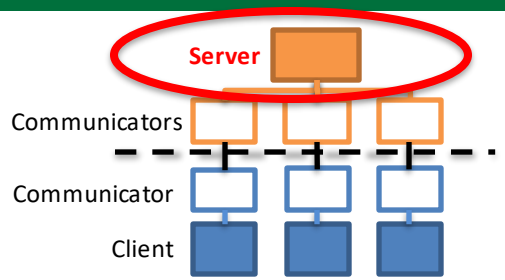
1. **Listen for keyboard input (blocks Thread until Enter key pressed)**
2. **Communicates with server on separate Thread (does not block waiting for keyboard input)**

Client listens for keyboard on main thread creates Communicator on second thread



ChatServer manages one Communicator for each Client

ChatServer.java



Set up *ServerSocket* to listen for Client connections

```
14 public class ChatServer {
15     private ServerSocket listen;                    // for accepting connections
16     private ArrayList<ChatServerCommunicator> comms; // all the connections with clients
17
18     public ChatServer(ServerSocket listen) {
19         this.listen = listen;
20         comms = new ArrayList<ChatServerCommunicator>();
21     }
22
23     /**
24      * The usual loop of accepting connections and firing off new threads to handle them
25      */
26     public void getConnections() throws IOException {
27         while (true) {
28             //listen.accept in next line blocks until new connection
29             ChatServerCommunicator comm = new ChatServerCommunicator(listen.accept(), this);
30             comm.setDaemon(true);
31             comm.start();
32             addCommunicator(comm);
33         }
34     }
35
36     /**
37      * Adds the handler to the list of current client handlers
38      */
39     public synchronized void addCommunicator(ChatServerCommunicator comm) {
40         comms.add(comm);
41     }
42
43     /**
44      * Removes the handler from the list of current client handlers
45      */
46     public synchronized void removeCommunicator(ChatServerCommunicator comm) {
47         comms.remove(comm);
48     }
```

- Create one Communicator for each Client
- Keep Communicators in *comms* ArrayList

Block until Client connection, then create new Communicator running on its own Thread

- Returns new socket for this Communicator
- Also pass reference to this ChatServer object so clients can call methods on this object (call *broadcast()*)

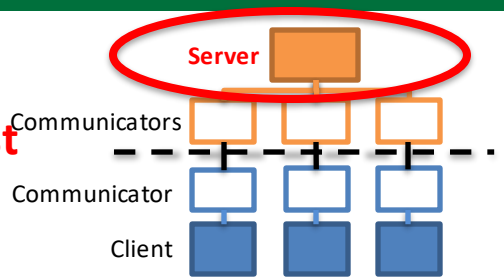
Set daemon, start Thread running, add to *comms* Arraylist

Add or remove Communicator Object from *comms* ArrayList

ChatServer manages one Communicator for each Client

ChatServer.java

- **Synchronized keyword makes sure that if two messages arrive at the same time, that broadcast finishes the first message before the second**
- **Topic of next class**




```
45      */
46     public synchronized void removeCommunicator(ChatServerCommunicator comm) {
47         comms.remove(comm);
48     }
49
50     /**
51      * Sends the message from the one client handler to all the others (but not echoing
52      */
53     public synchronized void broadcast(ChatServerCommunicator from, String msg) {
54         for (ChatServerCommunicator c : comms) {
55             if (c != from) {
56                 c.send(msg);
57             }
58         }
59     }
60
61     public static void main(String[] args) throws Exception {
62         System.out.println("waiting for connections");
63         new ChatServer(new ServerSocket(4242)).getConnections();
64     }
65 }
```

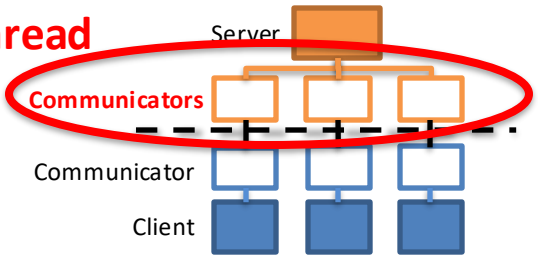
Clients will ask Server to broadcast message to all Clients, loop over each Communicator (except Client that sent message) and ask Communicator to send a message to its Client

***main()* set up ServerSocket listening on port 4242**

Each ChatServerCommunicator runs on own Thread and talks with one Client

ChatServerCommunicator.java

Extend Thread to run in own thread




```
9  public class ChatServerCommunicator extends Thread {
10     private Socket sock;                  // each instance is in a different thread and has its own socket
11     private ChatServer server;               // the main server instance
12     private String name;                  // client's name (first interaction with server)
13     private BufferedReader in;               // from client
14     private PrintWriter out;              // to client
15
16     public ChatServerCommunicator(Socket sock, ChatServer server) {
17         this.sock = sock;
18         this.server = server;
19     }
20
21     public void run() {
22         try {
23             System.out.println("someone connected");
24
25             // Communication channel
26             in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
27             out = new PrintWriter(sock.getOutputStream(), true);
28
29             // Identify -- first message is the name
30             name = in.readLine();
31             System.out.println("it's "+name);
32             out.println("welcome "+name);
33             server.broadcast(this, name + " entered the room");
34
35             // Chat away
36             String line;
37             while ((line = in.readLine()) != null) {
38                 String msg = name + ":" + line;
39                 System.out.println(msg);
40                 server.broadcast(this, msg);
41             }
42
43             // Done
44             System.out.println(name + " hung up");
45             server.broadcast(this, name + " left the room");
46
47             // Clean up -- note that also remove self from server's list of handlers so it doesn't broadcas
48             server.removeCommunicator(this);
49             out.close();
50             in.close();
51             sock.close();
52         }
```

- Save socket to communicate with Client
- Save ChatServer to communicate with ChatServer Object (e.g., call *broadcast()*)

run() called when Thread is started

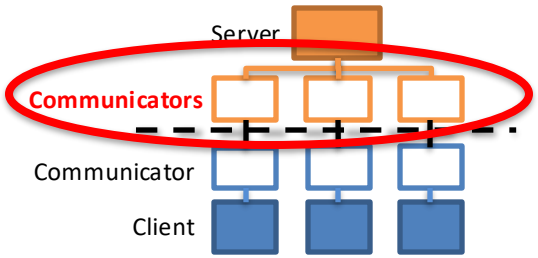
Set up in reader and out writer as before

On any input from Client, call *broadcast()* on Server *broadcast()* on Server will call *send()* on each Communicator (next slide)

When Client hangs up, call *removeCommunicator()* on Server and shut down this Thread

Each ChatServerCommunicator runs on own Thread and talks with one Client

ChatServerCommunicator.java




```
58      /**
59       * Sends a message to the client
60       * @param msg
61       */
62      public void send(String msg) {
63          out.println(msg);
64      }
65  }
66
```

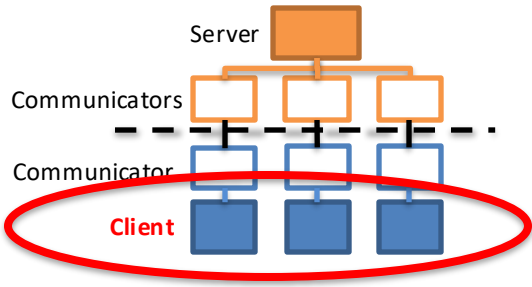
When another Client sends a message to the Server via *broadcast()* method, the Server will call *send()* on each Communicator to broadcast the message to all Clients

ChatClient manages keyboard input and creates a ChatClientCommunicator

ChatClient.java

```
11 public class ChatClient {
12     private Scanner console;                // input from the user
13     private ChatClientCommunicator comm;    // communication with the server
14     private boolean hungup = false;             // has the server hung up on us?
15
16     public ChatClient(Socket sock) throws IOException {
17         // For reading lines from the console
18         console = new Scanner(System.in);
19
20         // Fire off a new thread to handle incoming messages from server
21         comm = new ChatClientCommunicator(sock, this);
22         comm.setDaemon(true);
23         comm.start();
24
25         // Greeting; name request and response
26         System.out.println("Please enter your name");
27         String name = console.nextLine(); //blocks until keyboard input
28         comm.send(name);
29     }
30
31     /**
32      * Get console input and send it to server;
33      * stop & clean up when server has hung up (noted by hungup)
34      */
35     public void handleUser() throws IOException {
36         while (!hungup) {
37             //console.nextLine() blocks until text is entered
38             comm.send(console.nextLine());
39         }
40     }
41
42     /**
43      * Notes that the server has hung up (so handleUser loop will terminate)
44      */
45     public void hangUp() {
46         hungup = true;
47     }
48
49     public static void main(String[] args) throws IOException {
50         new ChatClient(new Socket("localhost", 4242)).handleUser();
51     }
52 }
```

Set up scanner for keyboard input



Create Communicator on another Thread (so not stopped by blocking scanner), start Thread running

While Server is connected, block here (due to scanner *nextLine()*) until user presses enter key, then tell Communicator to send keyboard messages to Server

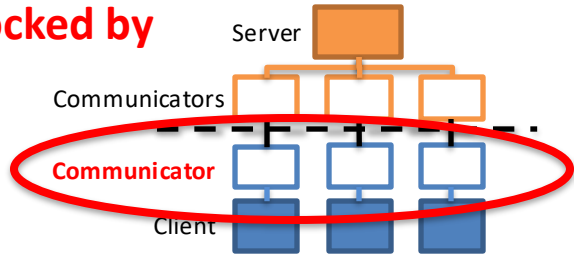
If Server hangs up, Communicator will call this method to inform ChatClient Object

main()* calls constructor passing socket on localhost port 4242 then *handleUser()

ChatClientCommunicator runs on its own Thread to communicate with Server

ChatClientCommunicator.java

Run on own Thread so not blocked by scanner




```
10 public class ChatClientCommunicator extends Thread {
11     private Socket sock;         // the underlying socket for communication
12     private ChatClient client;       // for which this is handling communication
13     private BufferedReader in;       // from server
14     private PrintWriter out;     // to server
15
16     public ChatClientCommunicator(Socket sock, ChatClient client) throws IOException {
17         this.sock = sock;
18         this.client = client;
19         in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
20         out = new PrintWriter(sock.getOutputStream(), true);
21     }
22
23     public void send(String msg) {
24         //called when have keyboard input
25         this.out.println(msg);
26     }
27
28     public void run() {
29         // Get lines from server; print to console
30         try {
31             String line;
32             while ((line = in.readLine()) != null) {
33                 System.out.println(line);
34             }
35         }
36         catch (IOException e) {
37             e.printStackTrace();
38         }
39         finally {
40             client.hangUp();
41             System.out.println("server hung up");
42         }
43
44         // Clean up
45         try {
46             out.close();
47             in.close();
48             sock.close();
49         }
50         catch (IOException e) {
51             e.printStackTrace();
52         }
53     }
```

- Save *socket* to communicate with ChatServer
- Save *client* to communicate with ChatClient Object (call *hangUp()* if server hangs up)

Send keyboard message passed by ChatClient Object to Server

Read data from ChatServer and write to console

If ChatServer hangs up, tell ChatClient Object, then end Thread