Week 10 - Wednesday

COMP 2000

Last time

- What did we talk about last time?
- Finished serialization
- Internet
- Networking

Questions?

Project 3

Socket Communication

TCP/IP

- A TCP/IP connection between two hosts (computers) is defined by four things
  - Source IP
  - Source port
  - Destination IP
  - Destination port
- One machine can be connected to many other machines, but the port numbers keep the different connections straight

Clients vs. servers

- Using sockets is usually associated with a client-server model
- A **server** is a process that sits around waiting for a connection
  - When it gets one, it can do sends and receives
- A **client** is a process that connects to a waiting server
  - Then it can do sends and receives
- Clients and servers are processes, not computers
  - You can have many client and server processes on a single machine

Creating a server socket in Java

- To create a server socket, we instantiate a **`ServerSocket`** object with the port that the server will listen on

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

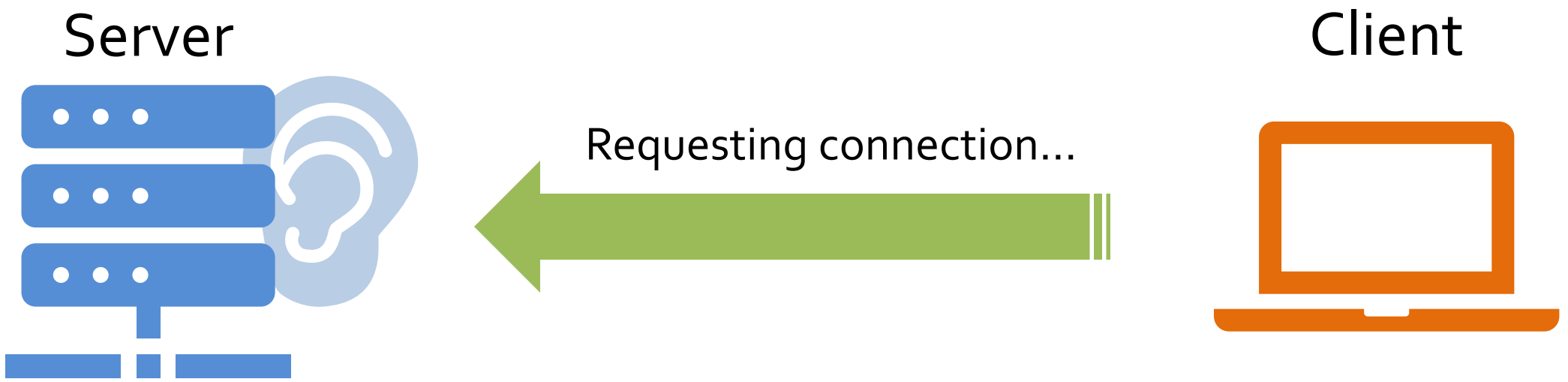
- That creates the server, but then we have to try to accept a connection

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

- The **`accept()`** method is a blocking method that will wait for a client to try to connect

Listening server

- The server sits there, waiting for a client to connect
- Until that happens, the **`accept()`** method will not return
- When it does return, it will return with a socket that can be used for communicating with the client



Connecting to a listening server

- The client code to connect to a listening server is:

```
Socket socket = new Socket(address, port);
```

- Where **`address`** is a **`String`** containing either a legal IP address (like **`"174.103.113.51"`**) or a legal domain name (like **`"otterbein.edu"`**)
- And **`port`** is the appropriate port number
- Remember that this code is running in a different program, very likely on a different computer

Port numbers

- As we discussed before, many port numbers are already reserved for specific applications
  - **20** and **21**: File Transfer Protocol (FTP)
  - **22**: Secure Shell (SSH)
  - **80**: Hypertext Transfer Protocol (HTTP)
- If you're writing a tool that uses one of those protocols, use the correct port
- If you're writing something else, make sure you don't use a port reserved for something else
  - Definitely use port 1024 or higher, since below 1024 are pretty much taken up

Loopback IP address

- It's inconvenient to need two different computers to write network code
- For testing purposes, you can often use a single computer as both the server and the client
- To do so, you need to connect to yourself
- What's your IP address?
- Well, it might always be changing
- To make things simpler, there's a loopback IP address that always refers to the computer you're currently on: `127.0.0.1`
- The IPv6 loopback address is `::1` (where `::` is notation that means "fill in with appropriate numbers of zeroes")

Using the sockets

- Now that you've got sockets, what are you going to do with them?
- Sockets allow for two-way communication
- You can get an input stream (that you can read from) and an output stream (that you can write to)
- These streams can be used where you might have used a file object or a stream created from a file
- From this point on, using sockets looks a lot like using file I/O

Socket for input

- Let's say you have a socket and you want to read some text from it
- Make a **`Scanner`** using its input stream:

```
Scanner netIn = new Scanner(socket.getInputStream());
```

- Then, you can read text just like you would from any other **`Scanner`**:

```
int value = netIn.nextInt();
```

- Creating a socket and getting its input stream can both throw an **`IOException`**, needing a **`throws`** or a **`try-catch`**, which I'm leaving out for simplicity

Socket for output

- Or maybe you want to write some text across the network
- Make a **`PrintWriter`** using its output stream:

```
PrintWriter netOut = new PrintWriter(socket.getOutputStream());
```

- Then, you can print text just like you would from **`System.out`**:

```
netOut.println("That's what she said.");
```

- Again, getting the output stream can throw an **`IOException`** if something's wrong

Text or binary data: You pick

- The previous two slides showed ways to use a socket to make a **`Scanner`** for text input or a **`PrintWriter`** for text output
- But you can just as easily send and receive binary data across a network connection
- Using a socket's input stream, you could create a **`DataInputStream`** or an **`ObjectInputStream`**

```
DataInputStream netIn = new DataInputStream(socket.getInputStream());
```

- Similarly, using a socket's output stream you could create a **`DataOutputStream`** or an **`ObjectOutputStream`**

```
DataOutputStream netOut = new DataOutputStream(socket.getOutputStream());
```

Server example

- Write a server that listens on port 4444
- When it accepts a connection, it creates a **`Scanner`** to read from the socket
- It reads lines of text and prints them to the screen until it gets **`"quit"`**

Client example

- Write a client that connects to a server at the loopback address on port 4444
- It creates a **`PrintWriter`** to writer to the socket
- It reads lines of text from the user and sends them to the socket until the user enters **`"quit"`**
- It sends this final message and then closes the socket

Quiz

Upcoming

Next time…

- Review

Reminders

- **Work on Project 3**
  - **Project 3 is now due on April 3**
- Review everything after Exam 1
- Exam 2 will be on Monday, March 30