Programming Methods
Dr Robert Harle

IA NST CS and CST

Lent 2008/09

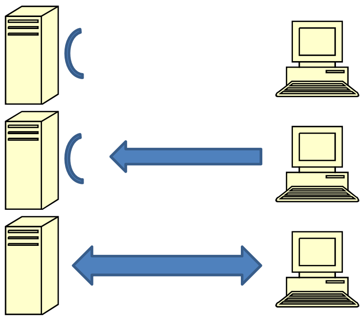
Handout 4

Our Motivating Example

- We're going to make the world's simplest web server
  - It's not going to challenge apache etc
  - But it will give us something to think about...
- I can't assume you know how the internet works (pixie dust primarily)
  - So we'll start with a really brief review
  - You can find out much more online

Client-Server

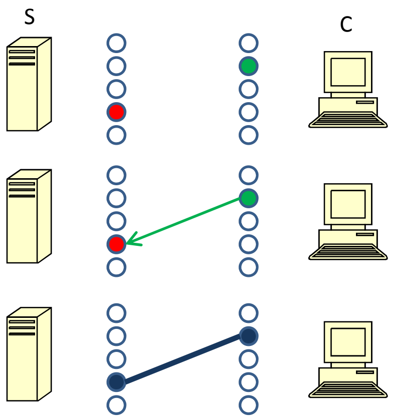
- The key notion these days is that of client-server
  - A *server* is a machine that sits there waiting for connections from one or more *clients*
  - The web is packed with servers that deliver web pages and browsers that act as clients



- But what if we have multiple types of server application (web, email, etc)?
- How does the client know where to connect?
- For this we use Berkeley sockets ("sockets")

Sockets

- Each machine has an **address** *(c.f. Phone number)*
- Each machine has lots of **ports** to contact it on *(c.f. Phone extensions)*
- In software we create **sockets** which are software objects that represent a connection between two systems



Server S is listening on port P (red)
Client C creates a socket and associates it with some port (green)

C attempts to connect to its socket to port P on server S (shorthand is S:P)

S responds and creates a socket at its end to represent the now open connection in software

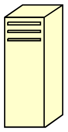
Java Sockets

- Java class library does the hard work for us
- Class **Socket**
  - Represents a socket (there was a clue in the title...)
- Class **ServerSocket**
  - Listens for connections of a specified port
  - Each time a connection comes in, it queues it up
  - Each time we call accept() on it, it gives us a Socket object that is attached to the connection at the top of the queue

Usage

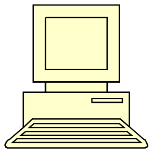
- Setup a listening socket on port 10000 (server):

```
ServerSocket ss = null;
Socket result = null;
try {
   ss = new ServerSocket(10000);
   // The call below will wait until there is a connection
   // from someone and then give us access to that
   // connection via variable result
   result = ss.accept();
} ...
```



- Create a socket and connect to mymachine.com, port 10000 (client):

```
Socket s = null;
try {
   s = new Socket(“mymachine.com", 10000);
}
catch(IOException ioe) {
   ...
}
```



Usage

- Once we have a connected Socket, it's just a place to get or receive data
  - You can just apply the usual stream reading or writing tools that you have seen in the practicals
  - E.g

```
Socket s = new Socket("somewhere.com",4000);
Reader r = new InputStreamReader(s.getInputStream());
BufferedReader br = new BufferedReader(r);
String text = br.getLine();
```

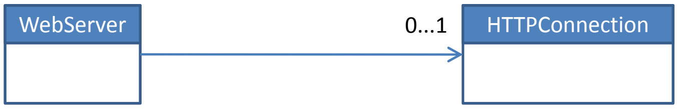
Note that the docs for BufferedReader say: *"In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders*." Which design pattern is in use here?

Common Ports You Might Know (?)

- 21 – FTP
- 22 – SSH
- 23 – Telnet
- 53 – DNS
- 80 – Internet (web traffic)

- When you go to www.howtogetafirst.com this is just client-server in action
  - Your machine connects to a preconfigured DNS machine that tells it a numerical address for the machine associated with www.howtogetafirst.com
  - Your machine then connects to that address on port 80
  - If there is a web server there, it gets the web page

Web Server Design



- *Really* simplistic
- **WebServer** encapsulates the server part (listening for connections)
- **HTTPConnection** encapsulates a single, live HTTP connection and handles any requests

WebServer State

```
public class WebServer {

/**
 * The connection to a client (if any)
 */
private HTTPConnection mConnection = null;

/**
 * The server port
 */
private int mPort;

/**
 * Constructor stores the web server's port
 * @param port
 */
public WebServer(int port) {
        mPort = port;
}
```

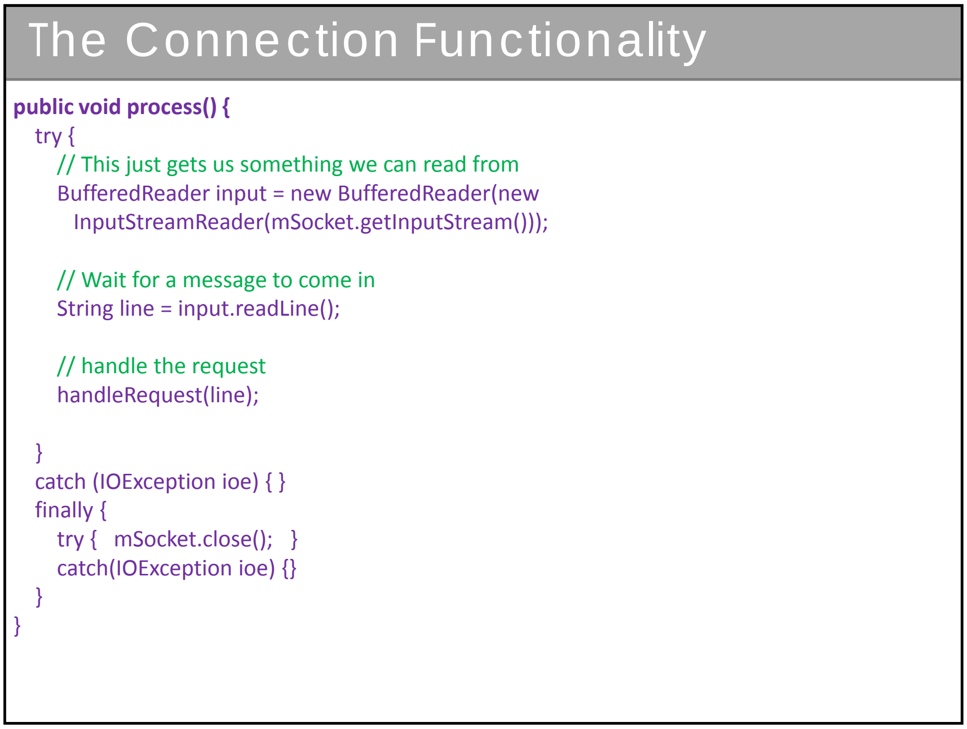
- Initialise our HTTP connection to null to indicate there is no live connection
- Private port number to listen on
- Constructor requires that a port number be specified when creating the server

WebServer Process

```
public void runServer() {

   while (true) {

      // Wait until we are contacted
      listenForNewConnection();

      // mConnection now set up
      mConnection.process();
   }

}
```

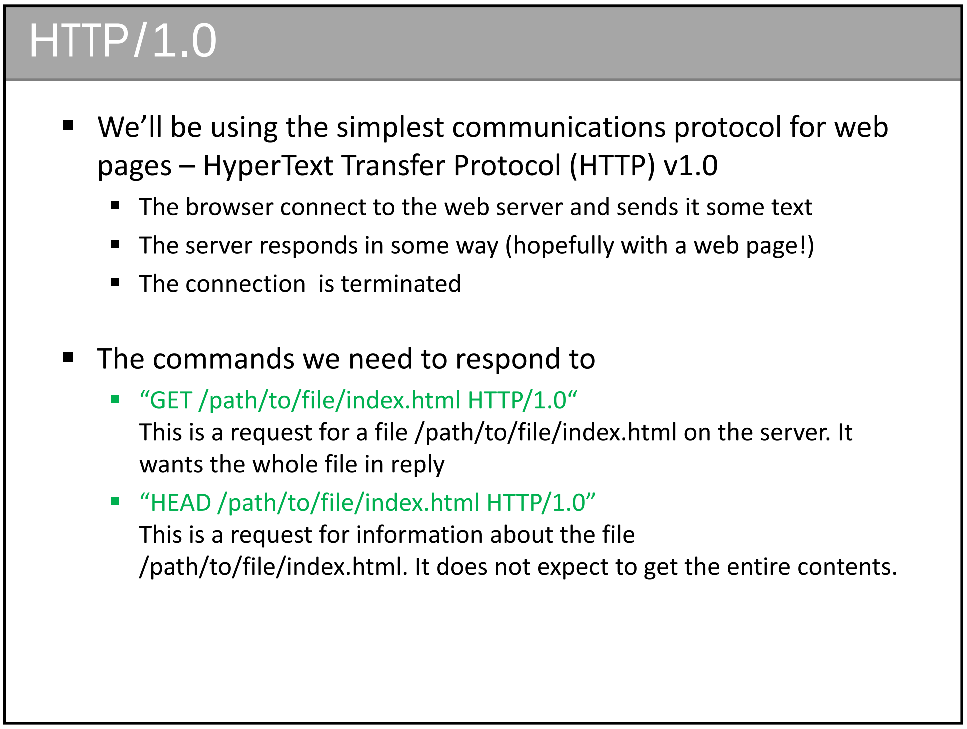
WebServer: Listening

```
private void listenForNewConnection() {
   ServerSocket serversocket = null;
   while (true) {
      try {
         serversocket = new ServerSocket(mPort);

         Socket connection = serversocket.accept();

         mConnection = new HTTPConnection(connection);
      }
       catch(IOException ioe) {
          // Something went wrong
       }
   }
}
```



The Connection Functionality

```
public void process() {
   try {
      // This just gets us something we can read from
      BufferedReader input = new BufferedReader(new
        InputStreamReader(mSocket.getInputStream()));

      // Wait for a message to come in
      String line = input.readLine();

      // handle the request
      handleRequest(line);

   }
   catch (IOException ioe) { }
   finally {
      try {   mSocket.close();   }
      catch(IOException ioe) {}
   }
}
```



HTTP/1.0

- We’ll be using the simplest communications protocol for web pages – HyperText Transfer Protocol (HTTP) v1.0
  - The browser connect to the web server and sends it some text
  - The server responds in some way (hopefully with a web page!)
  - The connection is terminated
- The commands we need to respond to
  - “GET /path/to/file/index.html HTTP/1.0“
    This is a request for a file /path/to/file/index.html on the server. It wants the whole file in reply
  - “HEAD /path/to/file/index.html HTTP/1.0”
    This is a request for information about the file /path/to/file/index.html. It does not expect to get the entire contents.

The HTTP Header

- Everything we send must be prefixed with a nice ‘header’ that describes the actual data. This is just a string of text for HTTP:

HTTP/1.0	*The protocol we’re using*
200 OK	*The message type*
Connection: close	*We intend to close the connection*
Server: SomeServerName	*The name of the server*
Content-Type: text/html	*The data type (we only do HTML)*
<data if any>	*Usually the web page*

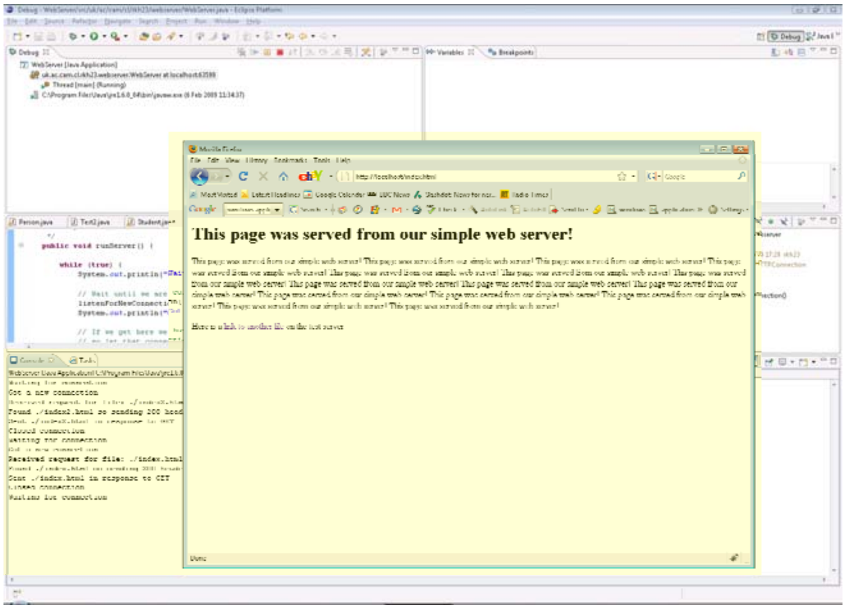
- This is done in the method **sendHeader()**

Sending the File

```
DataInputStream input=null;
try {
   FileInputStream f = new FileInputStream(file);
   input = new DataInputStream(f);
}

...

DataOutputStream output  = new
 DataOutputStream(mSocket.getOutputStream());


   while (true) {
      int b = input.read();
      if (b == -1) {
         break; //end of file
      }
      output.write(b);
   }
```

- Open the file for input
- Get the socket’s output
- Read each byte in from the file
- Send each byte out to the socket

Does it work?



This page was served from our simple web server!

This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server! This page was served from our simple web server!

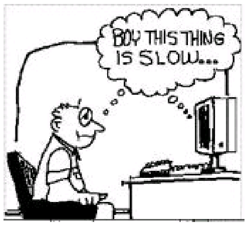
Here is a link to another file on the test server

```
    */
    public void runServer() {

        while (true) {
            System.out.println("Wai

            // Wait until we are
            listenForNewConnecti
            System.out.println("

            // If we get here we
            // so let that conne
```

```
Waiting for connection
Got a new connection
Received request for file: ./index2.html
Found ./index2.html so sending 200 head
Sent ./index2.html in response to GET
Closed connection
Waiting for connection
Got a new connection
Received request for file: ./index.html
Found ./index.html so sending 200 head
Sent ./index.html in response to GET
Closed connection
Waiting for connection
```

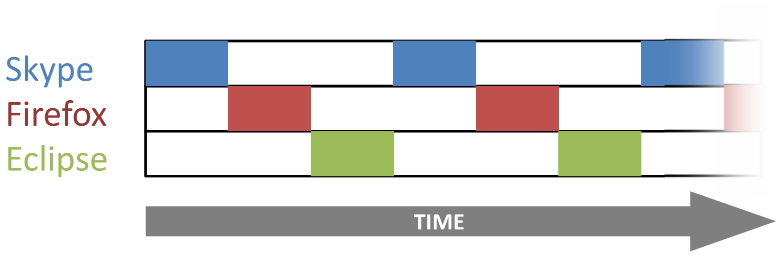
But...

- While we are dealing with a new connection
  - The **SocketServer** will queue up any other incoming connections (to a point)
  - But we process them one at a time
- Two problems here
  - The **SocketServer** queue might get full and we might lose connections
  - If the current request takes a long time to process, the queued requests will be stuck waiting. Thus our web server will seem really sluggish
- What we want to do is have the server process requests in *parallel* and not in *serial*...



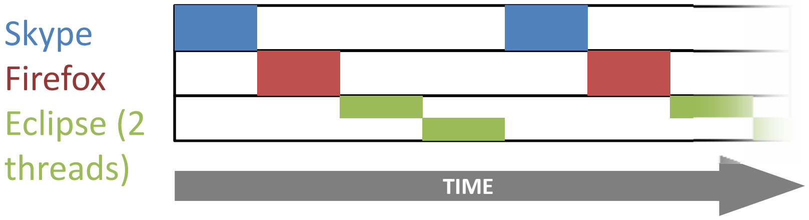
Multiple Processes (for the NSTs)

- A single processor can only do one thing at a time
- So how does it manage to run multiple applications simultaneously (word, skype, eclipse, etc)?
- Answer: it **fakes** it.
  - It rapidly shifts between programs, allowing them to run for very small amounts of time (milliseconds)
  - To us, everything seems to run simultaneously



Threading (for the NSTs)

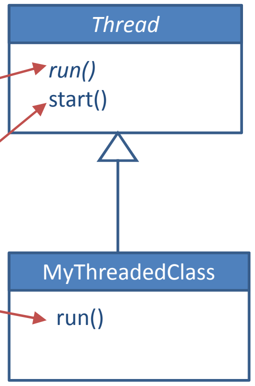
- When a single application wants to run multiple things at the same time, it creates a new ***thread*** which is treated in exactly the same way as a new process
- Why are threads useful?
  - Allow one web page to load while you scroll through another
  - Allow you to calculate results and still process input (for example, the cancel button!!)



Threads in Java

- Really easy to create
- Extend from java.lang.Thread
- Implement a method to run

public void run()
Override this abstract method. When started, the new thread treats this as the new **main()** method i.e. It’s the entry point for the thread

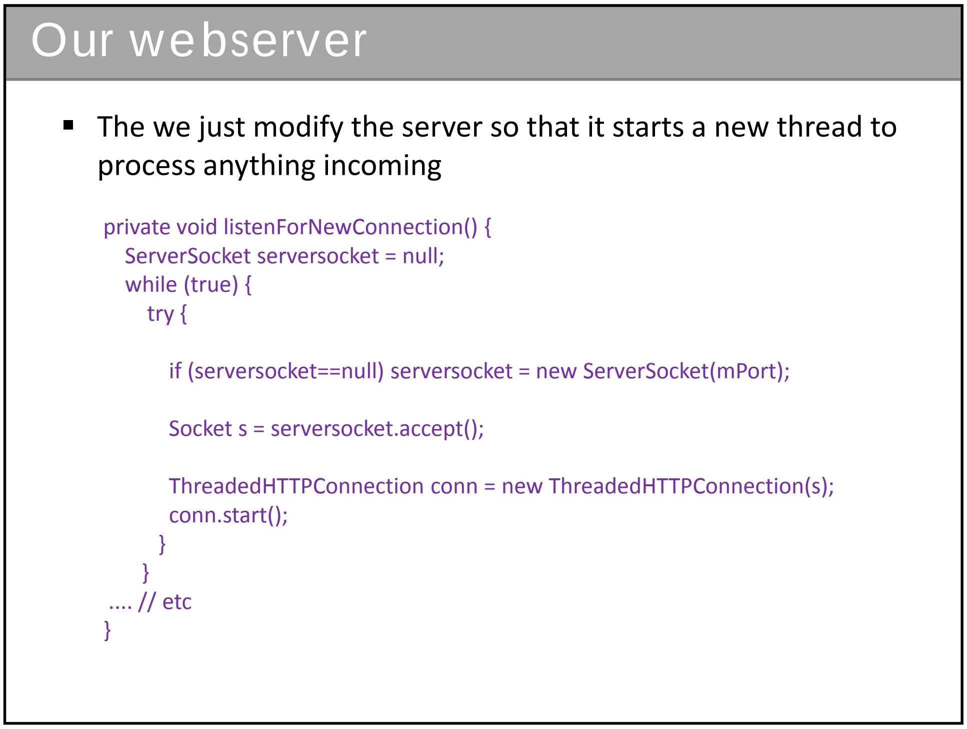


public void start()
This method does all the clever stuff for us. It starts the new thread and runs the **run()** method

Our webserver

- We can make HTTPConnection extend from Thread
  - We don’t have to change any existing code, just add a run() method that calls that existing code!

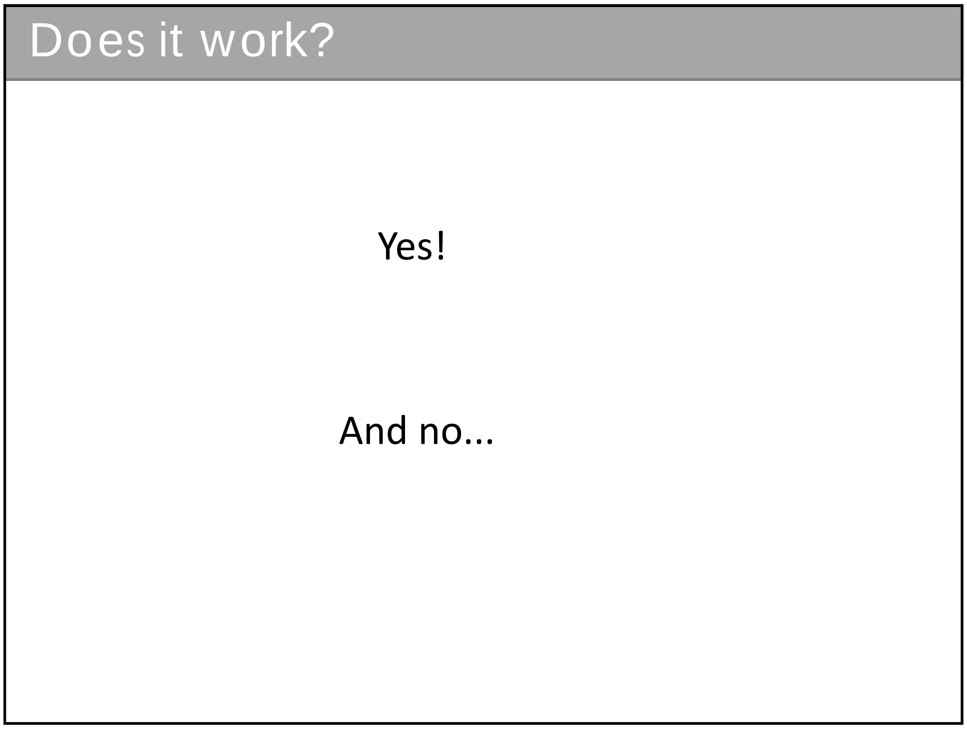
```
public class ThreadedHTTPConnection extends Thread {
  ....
  public void run() {
    this.process();
  }
  ....
}
```



Our webserver

- The we just modify the server so that it starts a new thread to process anything incoming

```
private void listenForNewConnection() {
   ServerSocket serversocket = null;
   while (true) {
      try {

         if (serversocket==null) serversocket = new ServerSocket(mPort);

         Socket s = serversocket.accept();

         ThreadedHTTPConnection conn = new ThreadedHTTPConnection(s);
         conn.start();
        }
      }
 .... // etc
}
```



Does it work?

Yes!

And no...

Close...

- When multiple clients requests come in simultaneously for different files, they are handled ‘simultaneously’
  - If we stress test it, there will come a point where it can’t handle the rate of requests, but...
- The problem comes when two clients want the same file
  - The second request ends up waiting until the first one has finished with the file!
  - There are obviously ways around this, but that’s a whole lecture course in itself!

Why Threads Suck...

- It is cool being able to run things *concurrently*
- But it gets really complex when:
  - We need to share information between threads
  - We need to share resources (files etc) between threads
  - One thread depends on another in any way

Why Threads Suck…

- This is an area known as *concurrency* control
- Really quite interesting to study that just becomes even more relevant as we start getting multi-processor chips
- Catches out a huge number of programmers and is the source of many bugs
- The simplest solution is rarely the most efficient!

- For those of you doing CST next year, you'll get to study it in gory detail!

For This Course

- You don't need to know about concurrency programming in any detail
- I expect you to:
  - Have a general idea of what a thread is
  - Know has to make your own class that can run in a thread