

# CSSE 220 Day 29


Network I/O

Work on Spellchecker Project

# CSSE 220 Day 29

- ▶ Everything for the Mini-project is due at the beginning of your class time on Day 30. **No late days** may be used for this one.
- ▶ Writing up and turning in written problems is no longer required. But you should still do them at some point.
- ▶ The Digital Resource Center is looking for a student to do ANGEL support for faculty.
  - See Nancy Bauer in the DRC if you're interested

# Course Evaluations

- ▶ I will provide some class time on Thursday for filling out the evaluation forms
  - ▶ I recommend that you wait until then to do them, so you'll be able to comment on the full course, including your project experience.
- 

# Project presentation/demonstration

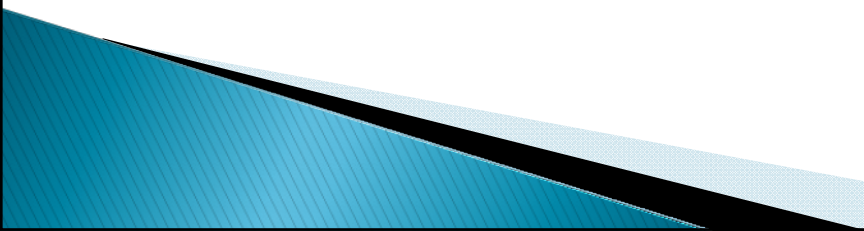
- ▶ Day 30 in class
- ▶ Informal and informational
- ▶ What does your program do? How does it do it
- ▶ Discuss your data Structures and algorithms.
  - **If you use an algorithm that you did not write, be sure that you can explain how it works.**
- ▶ Intended audience: Your classmates
  - Already know what the project is.
  - Already know Java
  - Already know the data structures we have studied.
- ▶ No more than 7 minutes, including Q&A time.
- ▶ **Just before your presentation, we will randomly choose which of your team members will present, so everyone should be prepared to do it.**
- ▶ **Commit an outline of your presentation to your team repository by 5:00 PM on Tuesday.**

# My schedule this week

	11 Monday	12 Tuesday	13 Wednesday	14 Thursday
8 am	CSSE220-01 O269 Anderson, Claude W	CSSE220-01 O269 Anderson, Claude W		CSSE220-01 O269 Anderson, Claude W
9 <sup>00</sup>			372 Project presentations GM room	
10 <sup>00</sup>	CSSE220-02 O269 Anderson, Claude W	CSSE220-02 O269 Anderson, Claude W		CSSE220-02 O269 Anderson, Claude W
11 <sup>00</sup>			Sr. ProjectExpo Union	
12 pm	craig Zilles UIUC curt's office	372 Scheme Grading; F21 ↻		CSSE department CSSE conference room ↻
1 <sup>00</sup>	CSSE Faculty Lunch with John Georgas			
2 <sup>00</sup>		PTRC Hadley ↻		PTRC Hadley ↻
3 <sup>00</sup>				
4 <sup>00</sup>	Meet with John Georgas			
5 <sup>00</sup>	John Georgas talk	Institute meeting	Special Faculty Meeting E-104	372 Rosie's List; F 210 ↻

- ▶ As always, you can find my up-to-date schedule online.

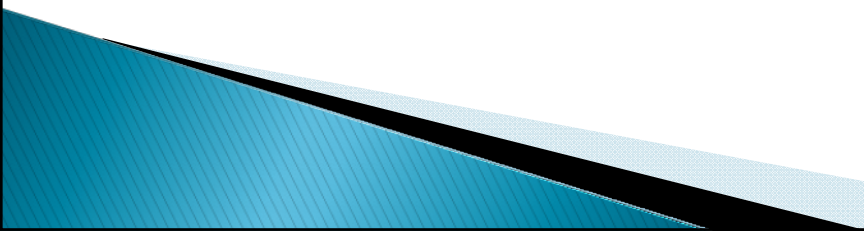
# Questions from students

- ▶ Spellchecker
  - ▶ Input and output
  - ▶ Networking
  - ▶ Anything else
- 

# Today's Agenda

- ▶ Random access files and serialization
  - ▶ Networking intro
  - ▶ Work on Spellchecker
- 

# Application Level Protocols

- ▶ TCP/IP mechanism establishes an Internet connection between two ports on two computers
  - ▶ Each Internet application has its own *application protocol*
  - ▶ This application protocol describes how data for that application are transmitted
- 



# Hypertext Transfer Protocol (HTTP)

- ▶ Application protocol used by the World Wide Web
- ▶ A web address is called a Uniform Resource Locator (URL)
- ▶ You type a URL into the address window of your browser
  - For example, `http://java.sun.com/index.html`

# Browser Steps: 1

1. Examines the part of the URL between the double slash and the first single slash
  - In this case: `java.sun.com`
  - This identifies the computer to which you want to connect
  - Because it contains letters, this part of the URL is a domain name, not an IP address
  - Browser sends request to a DNS server to obtain IP address for `java.sun.com`

# Browser Steps: 2

2. From the **http:** prefix, browser deduces that the protocol is HTTP
  - HTTP uses port 80 by default
3. It establishes a TCP/IP connection to port 80 at IP address obtained in step 1

# Browser Steps: 3

4. It deduces from the `/index.html` that you want to see the file `/index.html` and sends this request formatted as an HTTP command through the established connection

```
GET /index.html HTTP/1.0  
a blank line
```

# Browser Steps: 4

5. Web server running on computer whose IP Address was obtained above receives the request
  - It decodes the request
  - It fetches the file `/index.html`
  - It sends the file back to the browser on your computer

# Browser Steps: 5

6. The browser displays the contents of the file for you to see
  - Since this file is an HTML file, it translates the HTML codes into fonts, bullets, etc.
  - If the file contains images, it makes more GET requests through the same connection

# Telnet

- ▶ Telnet program allows you to
  - Type characters to send to a remote computer and
  - View the characters that the remote computer sends back
- ▶ It is a useful tool to establish test connections with servers
- ▶ You can imitate the browser connection by typing at the command line

```
telnet java.sun.com 80
```

# Telnet

- ▶ After Telnet starts, type the following without using backspace

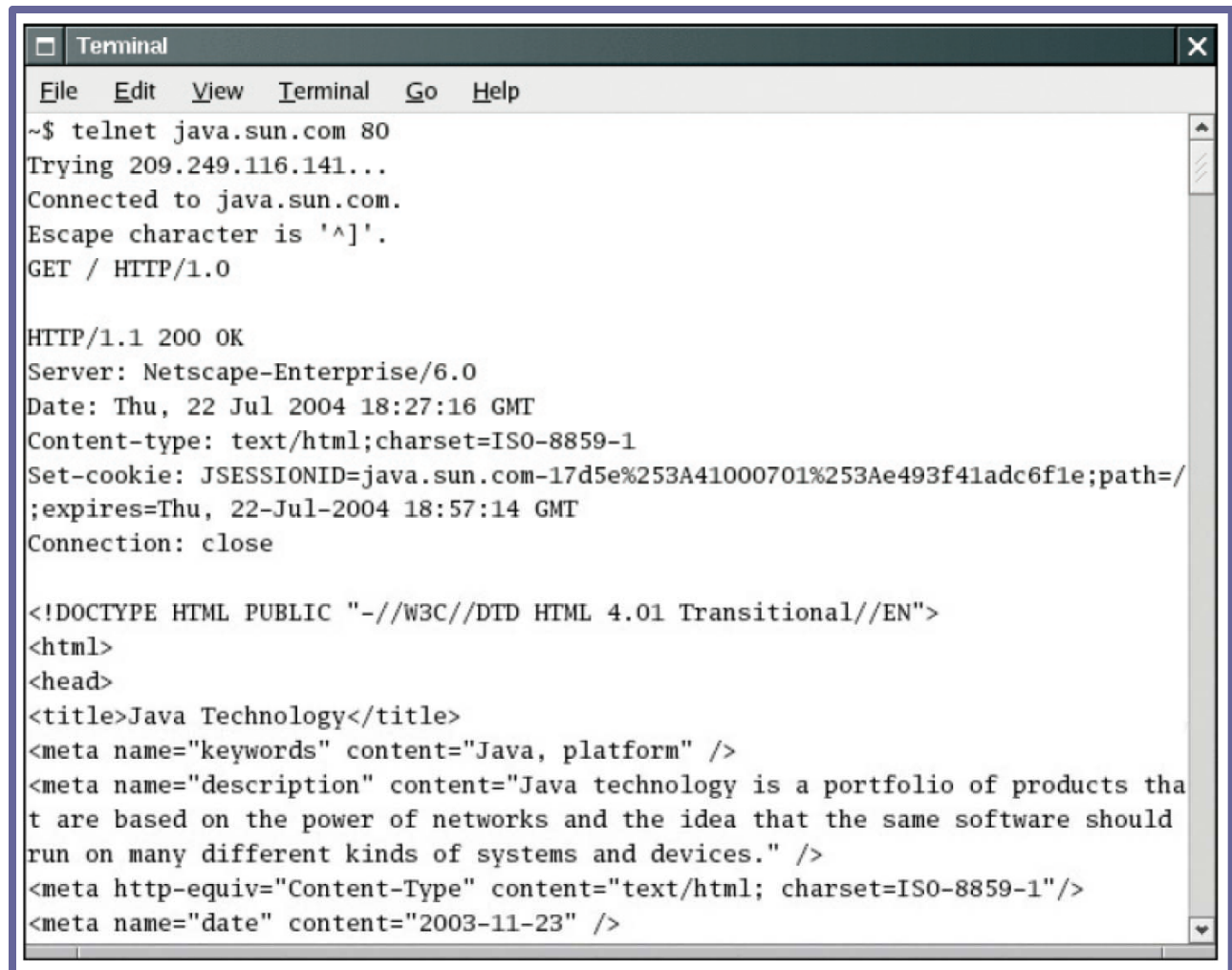
```
GET / HTTP/1.0
```

then press Enter twice

- ▶ The server responds to the request with the file
- ▶ Telnet is not a browser
- ▶ It does not understand HTML tags, so it just displays everything it was sent



# Web Server Response in Telnet



```
Terminal
File Edit View Terminal Go Help
~$ telnet java.sun.com 80
Trying 209.249.116.141...
Connected to java.sun.com.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Thu, 22 Jul 2004 18:27:16 GMT
Content-type: text/html;charset=ISO-8859-1
Set-cookie: JSESSIONID=java.sun.com-17d5e%253A41000701%253Ae493f41adc6f1e;path=/
;expires=Thu, 22-Jul-2004 18:57:14 GMT
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Java Technology</title>
<meta name="keywords" content="Java, platform" />
<meta name="description" content="Java technology is a portfolio of products that are based on the power of networks and the idea that the same software should run on many different kinds of systems and devices." />
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
<meta name="date" content="2003-11-23" />
```

# HTTP

- ▶ Do not confuse HTTP with HTML
- ▶ HTML is a *document format* that describes the structure of a document
- ▶ HTTP is a protocol that describes the command set for web server requests

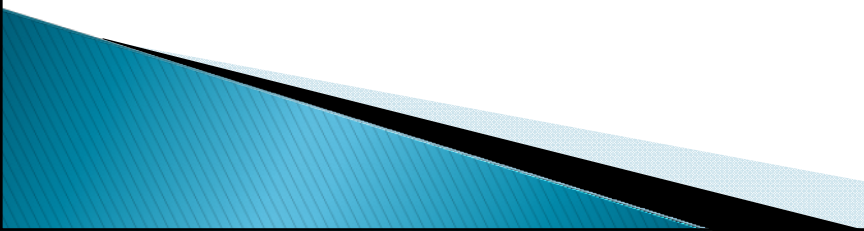
# HTTP

- ▶ Web browsers
  - Know how to display HTML documents
  - And how to issue HTTP commands
- ▶ Web servers
  - Know nothing about HTML
  - Merely understand HTTP and know how to fetch the requested items

# HTTP Commands

<b>Command</b>	<b>Meaning</b>
<b>GET</b>	<b>Return the requested item</b>
<b>HEAD</b>	<b>Request only the header information of an item</b>
<b>OPTIONS</b>	<b>Request communications option of an item</b>
<b>POST</b>	<b>Supply input to a server-side command and return the result</b>
<b>PUT</b>	<b>Store an item on the server</b>
<b>DELETE</b>	<b>Delete an item on the server</b>
<b>TRACE</b>	<b>Trace server communication</b>

# Application Level Protocols

- ▶ HTTP is one of many application protocols in use on the Internet
  - ▶ Another commonly used protocol is the Post Office Protocol (POP)
  - ▶ POP is used to download received messages from e-mail servers
  - ▶ To send messages, you use another protocol: Simple Mail Transfer Protocol (SMTP)
- 

# A Client Program – Sockets

- ▶ A socket is an object that encapsulates a TCP/IP connection
- ▶ There is a *socket* on both ends of a connection
- ▶ Syntax to create a socket in a Java program:

```
Socket s = new Socket(hostname, portnumber);
```

# A Client Program – Sockets

- ▶ Code to connect to the HTTP port of server,

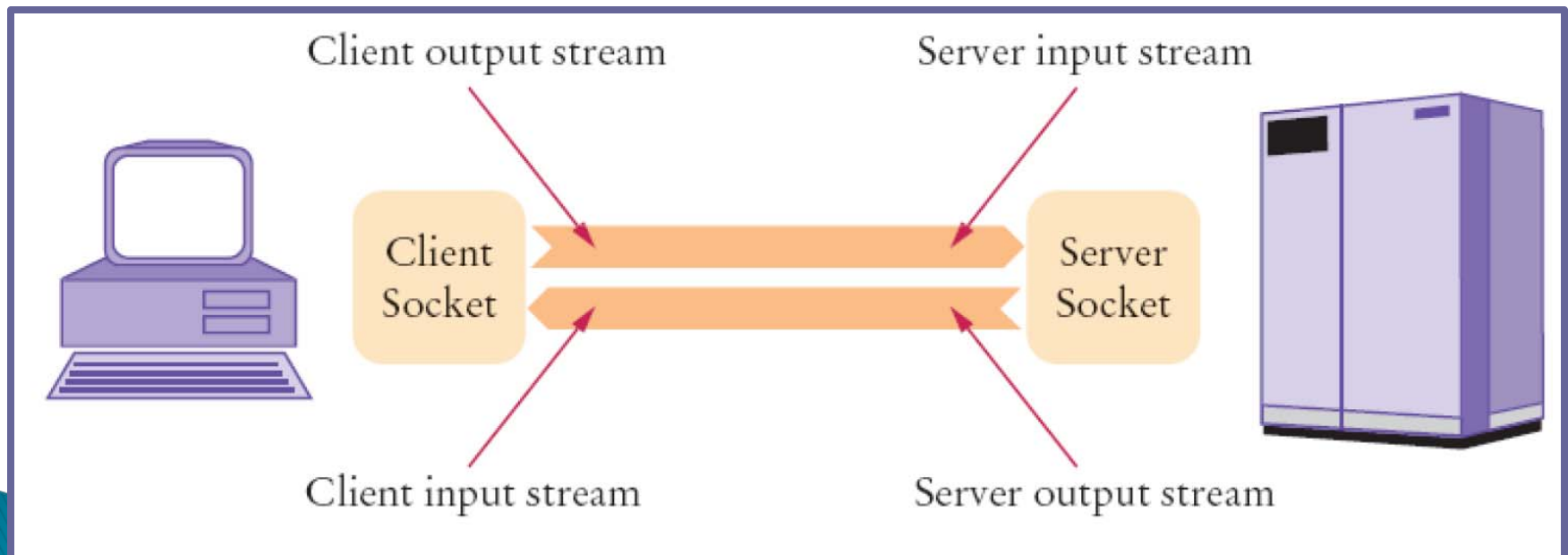
```
java.sun.com final int HTTP_PORT = 80;  
Socket s = new Socket("java.sun.com", HTTP_PORT);
```

- ▶ If it can't find the host, the Socket constructor throws an `UnknownHostException`

# Client Program – Input & Output Streams

- ▶ Use the input and output streams attached to the socket to communicate with the other endpoint
- ▶ Code to obtain the input and output streams

```
InputStream instream = s.getInputStream();  
OutputStream ostream = s.getOutputStream();
```



*Continued*



# Client Program – Input & Output Streams

- ▶ When you send data to `ostream`, the socket forwards them to the server
- ▶ The socket catches the server's response and you can read it through `istream`
- ▶ When you are done communicating with the server, close the socket

```
s.close();
```

# Client Program – Scanners and Writers

- ▶ `InputStream` and `OutputStream` send and receive bytes
- ▶ To send and receive text, use a scanner and a writer

```
Scanner in = new Scanner(instream);  
PrintWriter out = new PrintWriter(outstream);
```

# A Client Program – Scanners and Writers

- ▶ A `PrintWriter` *buffers* the characters and only sends when the buffer is full
  - Buffering increases performance
- ▶ When sending a command, you want the whole command to be sent now
  - *Flush* the buffer manually:

```
out.print(command);  
out.flush();
```

# A Client Program – WebGet

- ▶ This program lets you retrieve any item from a web server
- ▶ You specify the host and item from the command line
- ▶ For example:

```
java WebGet java.sun.com /
```

The "/" denotes the root page of the web server that listens to port 80 of

**java.sun.com**

*Continued*

# A Client Program – WebGet

- ▶ **WebGet:**
  - Establishes a connection to the host
  - Sends a **GET** command to the host
  - Receives input from the server until the server closes its connection

# File WebGet.java

```
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
```

```
/**
```

```
    This program demonstrates how to use a socket to communicate
    with a web server. Supply the name of the host and the
    resource on the command-line, for example
    java WebGet java.sun.com index.html
```

```
*/
```

```
public class WebGet
{
    public static void main(String[] args) throws IOException
    {
```

*Continued*

# File WebGet.java

```
// Get command-line arguments
```

```
String host;
```

```
String resource;
```

```
if (args.length == 2)
```

```
{
```

```
    host = args[0];
```

```
    resource = args[1];
```

```
}
```

```
else
```

```
{
```

```
    System.out.println("Getting / from java.sun.com");
```

```
    host = "java.sun.com";
```

```
    resource = "/";
```

```
}
```

*Continued*

# File WebGet.java

```
// Open socket
```

```
final int HTTP_PORT = 80;  
Socket s = new Socket(host, HTTP_PORT);
```

```
// Get streams
```

```
InputStream instream = s.getInputStream();  
OutputStream ostream = s.getOutputStream();
```

```
// Turn streams into scanners and writers
```

```
Scanner in = new Scanner(instream);  
PrintWriter out = new PrintWriter(ostream);
```



# File WebGet.java

```
// Send command
```

```
String command = "GET " + resource + " HTTP/1.0\n\n";  
out.print(command);  
out.flush();
```

```
// Read server response
```

```
while (in.hasNextLine())  
{  
    String input = in.nextLine();  
    System.out.println(input);  
}
```

```
// Always close the socket at the end
```

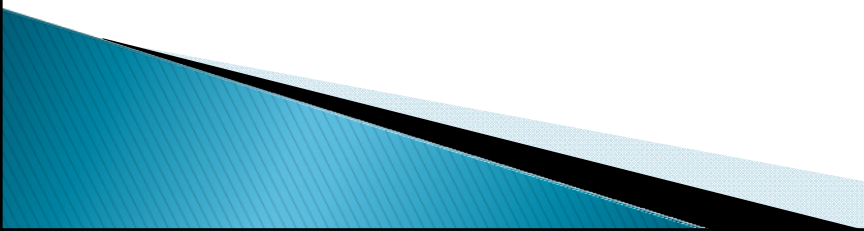
```
s.close();
```

```
}
```

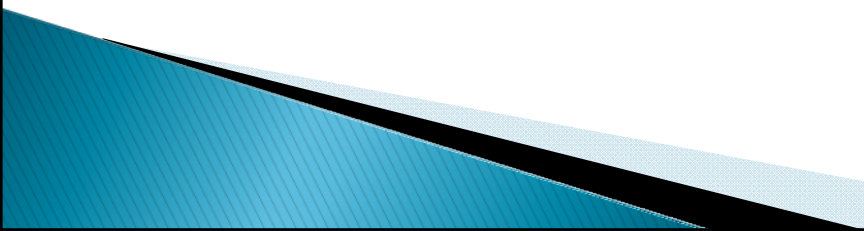
```
}
```

Continued

# Server and Client Example

- ▶ We probably won't get to the final example in class.
  - ▶ I have attempted to give sufficient explanation so you can get it by reading the slides.
  - ▶ Please study this examples and ask questions in the Day 30 class if there are things you do not understand.
- 

# A Server Program

- ▶ Sample server program: enables clients to manage bank accounts in a bank
  - ▶ When you develop a server application, you need to come up with an application-level protocol
  - ▶ The client can use this protocol to interact with the server
  - ▶ A simple bank access protocol is shown on the next slide
- 

# Simple Bank Access Protocol

<b>Client Request</b>	<b>Server Response</b>	<b>Meaning</b>
BALANCE $n$	$n$ and the balance	Get the balance of account $n$
DEPOSIT $n$ $a$	$n$ and the new balance	Deposit amount $a$ into account $n$
WITHDRAW $n$ $a$	$n$ and the new balance	Withdraw amount $a$ from account $n$
QUIT	none	Quit the connection

For this simple example, account numbers will be 0–9.

# A Server Program

- ▶ The server waits for clients to connect on a certain port
  - We choose **8888**
- ▶ To listen for incoming connections, use a server socket
- ▶ To construct a server socket, provide the port number.

```
ServerSocket server = new ServerSocket(8888);
```

- ▶ The ServerSocket is not the actual socket that the server will use to talk to the client, but merely a means of "listening" for a client that wants to connect to the server.

# A Server Program

- ▶ Use the **accept** method to wait for client connection and obtain a socket
- ▶ Note that once the client connects, the server has an ordinary socket (its half of the connection to the client).

```
Socket s = server.accept();  
BankService service = new BankService(s, bank);
```

# A Server Program – BankService

- ▶ **BankService** carries out the service
  - Implements the **Runnable** interface
  - Its **run** method will be executed in a separate thread that serves each client connection.
  - In this way, multiple clients can be connected at the same time.

*Continued*

# A Server Program – BankService

- ▶ **run** gets a scanner and writer from the socket, then calls **doService**, which reads and executes the client's commands:

```
public void doService() throws IOException {
    while (true) {
        if (!in.hasNext())
            return;
        String command = in.next();
        if (command.equals("QUIT"))
            return;
        executeCommand(command);
    }
}
```



# A Server Program – `executeCommand`

- ▶ Processes a single command
- ▶ If the command is **DEPOSIT**, it carries out the deposit

```
int account = in.nextInt();  
double amount = in.nextDouble();  
bank.deposit(account, amount);
```

- ▶ **WITHDRAW** is handled in the same way

*Continued*

# A Server Program - executeCommand

---

- ▶ After each command, the account number and new balance are sent to the client:

```
out.println(account + " " + bank.getBalance(account));
```

# A Server Program

- ▶ `doService` returns to the `run` method if the client closed the connection or the command equals `QUIT`
- ▶ Then `run` closes the socket and exits
- ▶ How can we support multiple simultaneous clients?
  - Spawn a new thread whenever a client connects
  - Each thread is responsible for serving one client

# A Server Program – Threads

- ▶ **BankService** implements **Runnable**; so, it can start a thread using **start()** (from the class **Thread**) .
- ▶ The new thread communicates with the client, so that the original thread can listen for another client connection.
- ▶ The new thread dies when the client quits or disconnects so that the **run** method exits

*Continued*

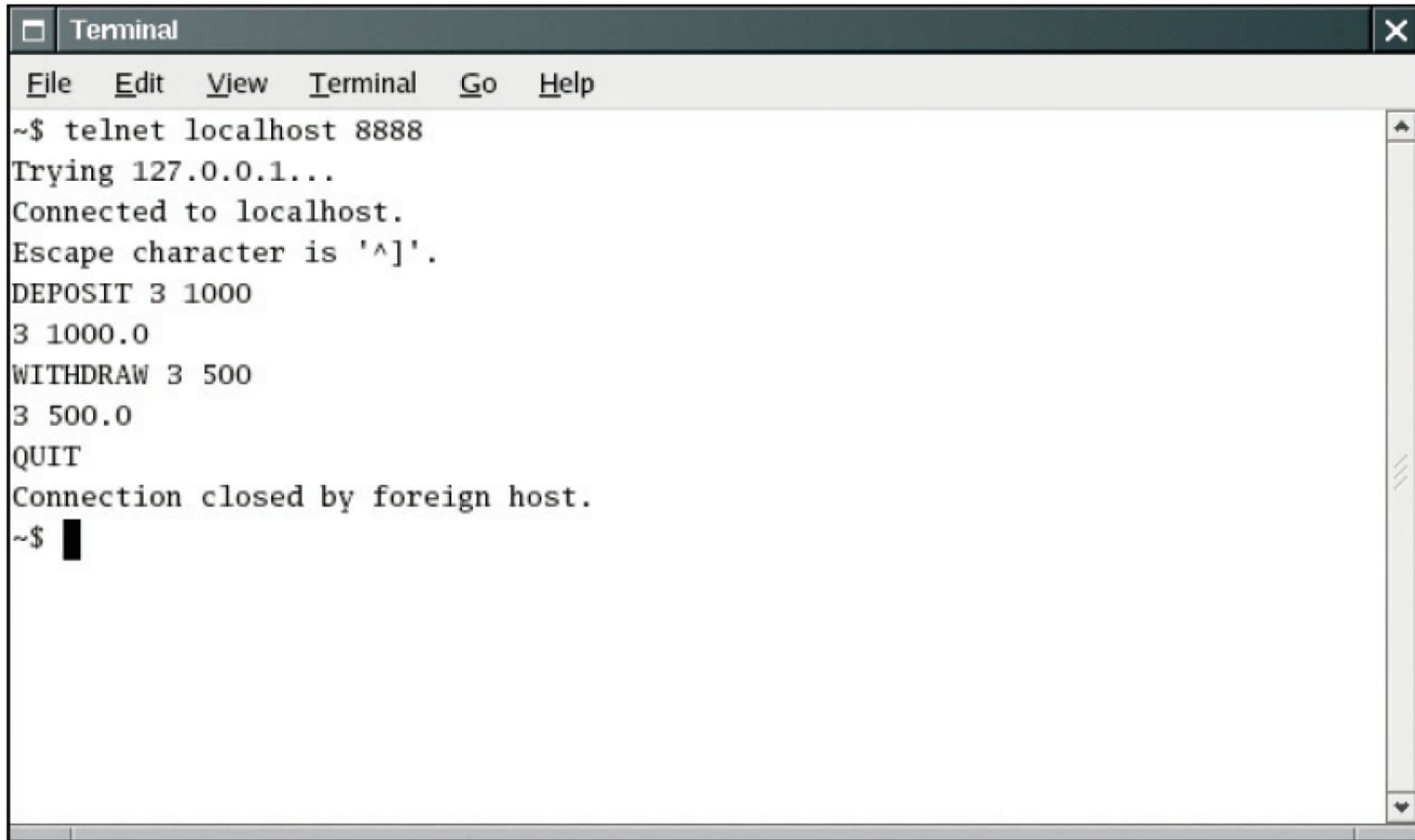
# A Server Program – Threads

- ▶ In the meantime, **BankServer** loops back to accept the next connection

```
while (true) {  
    Socket s = server.accept();  
    BankService service = new BankService(s, bank);  
    Thread t = new Thread(service);  
    t.start();  
}
```

- ▶ The server program never stops
- ▶ When you are done running the server, you need to kill it

# Using the Telnet Program to Connect to the Server

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Go", and "Help". The terminal output shows a telnet session to localhost on port 8888. The user enters "telnet localhost 8888", and the terminal shows "Trying 127.0.0.1...", "Connected to localhost.", and "Escape character is '^]'. The user then enters "DEPOSIT 3 1000", "3 1000.0", "WITHDRAW 3 500", "3 500.0", and "QUIT". The terminal shows "Connection closed by foreign host." and returns to the prompt "~\$".

```
Terminal
File Edit View Terminal Go Help
~$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
DEPOSIT 3 1000
3 1000.0
WITHDRAW 3 500
3 500.0
QUIT
Connection closed by foreign host.
~$
```

# File BankServer.java

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

// A server that executes the Simple Bank Access Protocol.
public class BankServer {
    public static void main(String[] args) throws IOException {
        final int ACCOUNTS_LENGTH = 10;
        Bank bank = new Bank(ACCOUNTS_LENGTH);
        final int SBAP_PORT = 8888;
        ServerSocket server = new ServerSocket(SBAP_PORT);
        System.out.println("Waiting for clients to connect...");

        while (true) {
            Socket s = server.accept();
            System.out.println("Client connected.");
            BankService service = new BankService(s, bank);
            Thread t = new Thread(service);
            t.start();
        }
    }
}
```

Start a new thread to handle this client connection so that the server can continue to accept other connections.

# File BankService.java

```
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

// Executes Simple Bank Access Protocol commands from a socket.

public class BankService implements Runnable {
    private Socket s;
    private Scanner in;
    private PrintWriter out;
    private Bank bank;

    /**
     Construct a service object that processes commands from a socket for a bank.
     @param aSocket the socket
     @param aBank the bank
    */
    public BankService(Socket aSocket, Bank aBank) {
        s = aSocket;
        bank = aBank;
    }
}
```



# BankService **run** method

```
public void run() {  
    try {  
        try {  
            in = new Scanner(s.getInputStream());  
            out = new PrintWriter(s.getOutputStream());  
            doService();  
        } finally {  
            s.close();  
        }  
    } catch (IOException exception) {  
        exception.printStackTrace();  
    }  
}
```

- ▶ Establish the stream connections to the client, and let **doService** do the actual work.

# BankService doService method

```
/**
    Executes all commands until the QUIT command or the
    end of input.
 */
public void doService() throws IOException
{
    while (true)
    {
        if (!in.hasNext()) return;
        String command = in.next();
        if (command.equals("QUIT")) return;
        else executeCommand(command);
    }
}
```

- ▶ Read and Execute commands until the command is "QUIT" or the client breaks the connection.

# BankService executeCommand method

```
public void executeCommand(String command){
    int account = in.nextInt();
    if (command.equals("DEPOSIT")){
        double amount = in.nextDouble();
        bank.deposit(account, amount);
    }
    else if (command.equals("WITHDRAW")) {
        double amount = in.nextDouble();
        bank.withdraw(account, amount);
    }
    else if (!command.equals("BALANCE")) {
        out.println("Invalid command");
        out.flush();
        return;
    }
    out.println(account + " " + bank.getBalance(account));
    out.flush();
}
```

- ▶ The call to **flush()** is necessary because a **PrintWriter**'s output is buffered.

# File Bank.java

## ▶ Field and constructor declarations

```
**  
  A bank consisting of multiple bank accounts.  
*/  
public class Bank {  
  
    private BankAccount[] accounts;  
  
    /**  
     Constructs a bank account with a given number of accounts.  
     @param size the number of accounts  
    */  
    public Bank(int size) {  
        accounts = new BankAccount[size];  
        for (int i = 0; i < accounts.length; i++)  
            accounts[i] = new BankAccount();  
    }  
}
```

# File Bank.java

## ▶ Field and constructor declarations

```
**
```

```
    A bank consisting of multiple bank accounts.
```

```
*/
```

```
public class Bank {
```

```
    private BankAccount[] accounts;
```

```
/**
```

```
    Constructs a bank account with a given number of accounts.
```

```
    @param size the number of accounts
```

```
*/
```

```
public Bank(int size) {
```

```
    accounts = new BankAccount[size];
```

```
    for (int i = 0; i < accounts.length; i++)
```

```
        accounts[i] = new BankAccount();
```

```
}
```

# Bank transaction methods

```
public void deposit(int accountNumber, double amount) {  
    BankAccount account = accounts[accountNumber];  
    account.deposit(amount);  
}
```

```
public void withdraw(int accountNumber, double amount) {  
    BankAccount account = accounts[accountNumber];  
    account.withdraw(amount);  
}
```

```
public double getBalance(int accountNumber) {  
    BankAccount account = accounts[accountNumber];  
    return account.getBalance();  
}
```

# BankClient first part (setup)

```
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class BankClient
{
    public static void main(String[] args) throws IOException
    {
        final int SBAP_PORT = 8888;
        Socket s = new Socket("localhost", SBAP_PORT);
        InputStream instream = s.getInputStream();
        OutputStream outstream = s.getOutputStream();
        Scanner in = new Scanner(instream);
        PrintWriter out = new PrintWriter(outstream);
```

# BankClient second part (commands)

```
String command = "DEPOSIT 3 1000\n";
System.out.print("Sending: " + command);
out.print(command);
out.flush();
String response = in.nextLine();
System.out.println("Receiving: " + response);

command = "WITHDRAW 3 500\n";
System.out.print("Sending: " + command);
out.print(command);
out.flush();
response = in.nextLine();
System.out.println("Receiving: " + response);

command = "QUIT\n";
System.out.print("Sending: " + command);
out.print(command);
out.flush();

s.close();
}
}
```



# BankAccount first part (setup)

The concurrency Lock is to make sure that two clients do not try to change the balance on this account simultaneously.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * A bank account has a balance that can be changed by
 * deposits and withdrawals.
 */
public class BankAccount {

    private double balance;
    private Lock balanceChangeLock;

    /**
     * Constructs a bank account with a zero balance.
     */
    public BankAccount() {
        balance = 0;
        balanceChangeLock = new ReentrantLock();
    }

    /**
     * Constructs a bank account with a given balance.
     * @param initialBalance the initial balance
     */
    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }
}
```

# BankAccount transactions

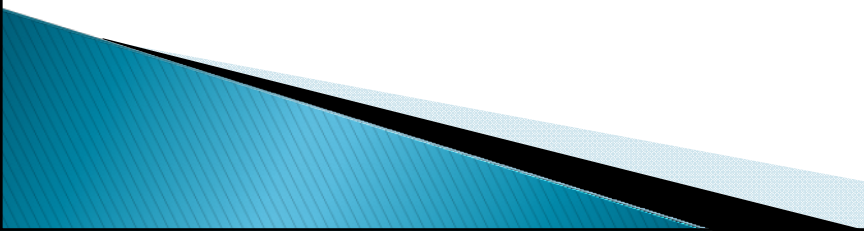
```
public void deposit(double amount) {
    balanceChangeLock.lock();
    try {
        double newBalance = balance + amount;
        balance = newBalance;
    } finally {
        balanceChangeLock.unlock();
    }
}
```

```
public void withdraw(double amount) {
    balanceChangeLock.lock();
    try {
        double newBalance = balance - amount;
        balance = newBalance;
    } finally {
        balanceChangeLock.unlock();
    }
}
```

```
public double getBalance() {
    return balance;
}
```

# Self Check

---

5. Why didn't we choose port 80 for the bank server?
  6. Can you read data from a server socket?
- 

# Answers

---

5. Port 80 is the standard port for HTTP. If a web server is running on the same computer, then one can't open a server socket on an open port.
  6. No, a server socket just waits for a connection and yields a regular `Socket` object when a client has connected. You use that socket object to read the data that the client sends.
- 