Advanced Object-oriented Programming

Lecture 19

Network Programming
A network consists of several computers connected so that data can be sent from one to another.

Network Programming covers most aspects of writing software for networks, from implementing low-level data-transfer protocols, to writing applications that provide access to data from across the whole network (e.g., a web-based front-end to a database).
Java provides classes that allow network applications to be built quickly and easily.

These classes provide methods for making and accepting connections across a network.

Before we look at these classes, and see how to use them, we’ll get an overview of the nuts and bolts of network connections.
Network Connections

When data is sent from one computer in a network to another,

\[
\begin{align*}
0110010101010101 \\
1010100101011011 \\
1110110010110000
\end{align*}
\]

some protocol is used, so that sender and receivers agree on how the data is to be interpreted.
Network Connections

When data is sent from one computer in a network to another,

\[ \begin{align*}
01100101 & \quad 01010101 \\
10101001 & \quad 01011011 \\
11101010 & \quad 10010000
\end{align*} \]

some protocol is used, so that sender and receivers agree on how the data is to be interpreted.
Protocols are important, whether the network is a small LAN, or a large network, such as the Internet.

Almost all communications between computers on the Internet use the Internet Protocol (IP) as a low-level data-transfer protocol.

In addition, the use of IP is usually controlled by the higher-level Transmission Control Protocol (TCP).

The combination of the two is usually referred to as TCP/IP.
When one (client) computer requests a file from a remote web server, data is sent in both directions:

- the **client computer sends** a request for a particular file;
- the **server sends** the contents of that file.
Suppose the client is requesting a file `index.html` from a server `www.liv.ac.uk`.

A connection is made between the client and the server.

Note that the connection may involve many intermediate ‘relay’ computers.
The HyperText Transfer Protocol (http)

Once the connection is established the request for the particular file is sent to the server. The request for the file is sent as a plain-text string:

```plaintext
GET /index.html HTTP/1.0
```

This third protocol (http) works at a higher level than TCP/IP. It’s concerned with what data is sent, rather than with how it’s sent.

For example, http defines the valid format for requests from clients:

- a valid http command (GET, POST, etc.), followed by
- a filestore address (e.g., /staff/department/index.html) followed by a version number (e.g., HTTP/1.0)
The HyperText Transfer Protocol (http)

Once the connection is established the request for the particular file is sent to the server. The request for the file is sent as a plain-text string:

GET /index.html HTTP/1.0

This *third* protocol (http) works at a higher level than TCP/IP. It’s concerned with *what* data is sent, rather than with *how* it’s sent.

For example, http defines the valid format for requests from clients:

- a valid http command (GET, POST, etc.), followed by
- a filestore address (e.g., /staff/department/index.html)
- followed by a version number (e.g., HTTP/1.0)
The http Protocol Response

The text sent from the server to the client might look like:

```
HTTP/1.1 200 OK
Date: Fri, 23 Oct 2009 15:18:18 GMT
Server: Apache/2.2.10 (Unix) ... 
X-Powered-By: PHP/5.2.6
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, ...
Pragma: no-cache
Content-Type: text/html
Set-Cookie: PHPSESSID=iu77ftkrco1538p3hpp1; path=/
Connection: close
```

so the browser knows how to ‘decode’ the following bytes
The http Protocol Response

The text sent from the server to the client might look like:

HTTP/1.1 200 OK
Date: Fri, 23 Oct 2009 15:18:18 GMT
Server: Apache/2.2.10 (Unix) . . .
X-Powered-By: PHP/5.2.6
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, . . .
Pragma: no-cache
Content-Type: text/html
Set-Cookie: PHPSESSID=iu77ftkrco1538p3hpp1; path=/
Connection: close

so the browser knows how to ‘decode’ the following bytes
This shows the text that is sent from client to server, and from server to client, but how is this text sent?

The **Internet Protocol** determines how data is sent at a low level.
The Internet Protocol

The Internet Protocol has just one function: to deliver data from one computer to another across the internet.

Data is packaged into chunks called ‘packets’; these packets are sent across the network; if any data is lost, this is noted but not acted on
The Internet Protocol

Packets are sent off to the recipient; they may be sent along different routes; they may be received in any order.
Data Loss and TCP

Losing occasional packets of data, or receiving them in the wrong order can be tolerated for some applications (e.g., audio/visual streams), but not for others (e.g., reading a web page).

TCP ensures that all data is received (e.g., by requesting lost packets to be re-sent).
Layers of Protocols

We’ve seen three different protocols each concerned with different aspects of communication between client and server:

- http
- TCP
- IP

Each works at a different level of abstraction.
Network Connections in Java

How can we write Java programs that communicate with other programs (processes) across a network?

As an example, we’ll write a program that connects to a web server, requests a web page, and prints out the result of that request to standard output (i.e., just the ‘raw’ HTML).
The `java.net.Socket` class represents network connections. An instance of the `Socket` class can be created by specifying the remote host, and the required program. The remote host name is a string, e.g., "www.liv.ac.uk". For our example, the program we need is the http server.
A computer typically runs many programs at the same time. Many of these are ‘system programs’ such as sendmail, crond, portmap, etc. (try ‘ps -aux’ under Linux).

In order to access a particular program on a particular computer, some sort of address system is needed.
Ports are abstract addresses for processes running on a computer.
A port is a number between 0 and 65,535.
Port numbers from 0 to 1,024 are reserved for common system processes.
For example, sendmail (SMTP) runs on port 25, web servers use port 80.
Creating Sockets in Java

We simply create an instance of `Socket`, and communicate via its I/O streams:

```java
Socket s = new Socket("www.liv.ac.uk", 80);
... s.getOutputStream();
... s.getInputStream();
```

The name of the remote machine to connect to
The port number of the webserver program on that machine
Creating Sockets in Java

We simply create an instance of `Socket`, and communicate via its I/O streams:

```java
Socket s = new Socket("www.liv.ac.uk", 80);
... s.getOutputStream();
... s.getInputStream();
```

The name of the remote machine to connect to
The port number of the webserver program on that machine
Creating Sockets in Java

We simply create an instance of `Socket`, and communicate via its I/O streams:

```
Socket s = new Socket("www.liv.ac.uk", 80);
... s.getOutputStream();
... s.getInputStream();
```

The name of the remote machine to connect to
The port number of the webserver program on that machine
Creating Sockets in Java

The Java programmer need only create the socket, get the I/O streams, and use them.

The details of connections and data-transmission protocols are all done in the implementation of the Socket class.
Example: Connecting to a Web Server

Our program will take command-line arguments for the remote host and the file to be requested.

```
example command-line
java WebConnector www.liv.ac.uk index.html
```

These arguments are supplied in the `String[] args` parameter to the `main` method.
Example: Connecting to a Web Server

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

public class WebConnector {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.out.println(...);
            System.exit(0);
        }
    }
}
```

For the `Socket` class
For the I/O classes
import java.net.*;
import java.io.*;

public class WebConnector {

    public static void main(String[] args)
        throws IOException {
        if (args.length != 2) {
            System.out.println(...);
            System.exit(0);
        }
    }

For the **Socket** class

For the I/O classes
Example: Connecting to a Web Server

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

public class WebConnector {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.out.println(...);
            System.exit(0);
        }
    }
}
```

For the Socket class
For the I/O classes
Example: Connecting to a Web Server

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

public class WebConnector {
    public static void main(String[] args)
        throws IOException {
        if (args.length != 2) {
            System.out.println(...);
            System.exit(0);
        }
```
Example: Connecting to a Web Server

Socket $s = \text{new } \text{Socket}(\text{args}[0], \text{80});$

This connects to the http server on the remote host.

May throw \textit{UnknownHostException} or \textit{IOException}: see the Socket API.
Example: Connecting to a Web Server

```
WebConnector#main(String[]), contd.

Socket s = new Socket(args[0], 80);
```

This connects to the http server on the remote host.

May throw UnknownHostException or IOException: see the Socket API.
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

Socket s = new Socket(args[0], 80);

This connects to the http server on the remote host.

May throw UnknownHostException or IOException: see the Socket API.
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

Socket s = new Socket(args[0], 80);

This connects to the http server on the remote host.

May throw UnknownHostException or IOException: see the Socket API.
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

```java
InputStream in = s.getInputStream();
BufferedReader br = new BufferedReader
(new InputStreamReader(in));

OutputStream out = s.getOutputStream();
PrintWriter pw = new PrintWriter
(new OutputStreamWriter(out));
```

To read data from the server
To send data to the server
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

```java
InputStream in = s.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader(in));

OutputStream out = s.getOutputStream();
PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
```

To read data **from** the server
To send data **to** the server
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

```java
InputStream in = s.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader(in));
OutputStream out = s.getOutputStream();
PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
```

To read data **from** the server
To send data **to** the server
Example: Connecting to a Web Server

```java
WebConnector#main(String[]), contd.

InputStream in = s.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader(in));
OutputStream out = s.getOutputStream();
PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
```

To read data **from** the server
To send data **to** the server
Example: Connecting to a Web Server

```java
pw.print("GET /" + args[1]
         + " HTTP/1.0
         n
         n");
pw.flush();
```

For example, this sends the request:

```
GET /index.html HTTP/1.0
```

The `PrintWriter.flush()` method forces any bytes in the buffer to be written to the output stream.
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

```java
pw.print("GET /" + args[1] + " HTTP/1.0\n\n");
pw.flush();
```

For example, this sends the request:

```
GET /index.html HTTP/1.0

```

The PrintWriter.flush() method forces any bytes in the buffer to be written to the output stream.
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

```java
pw.print("GET /" + args[1] + " HTTP/1.0\n\n");
pw.flush();
```

For example, this sends the request:

```
GET /index.html HTTP/1.0\n\n```

The PrintWriter.flush() method forces any bytes in the buffer to be written to the output stream.
String line;
while((line = br.readLine()) != null) {
    System.out.println(line);
}
br.close();
pw.close();
s.close();
}

Shutting down the streams and socket

Example: Connecting to a Web Server
Example: Connecting to a Web Server

WebConnector#main(String[]), contd.

String line;
while((line = br.readLine()) != null) {
    System.out.println(line);
}

br.close();
pw.close();
s.close();

Shutting down the streams and socket
Client-Server Applications

A server application is a program running on a port of a computer, that will accept connections across a network.

A client application is a program that connects to a server application.
Client-Server Applications

A server application is a program running on a port of a computer, that will accept connections across a network.

A client application is a program that connects to a server application.
Client-Server Applications

A **server application** is a program running on a port of a computer, that will accept connections across a network.

A **client application** is a program that connects to a server application.
Client-Server Applications

Examples include:

- **Web servers**: clients might be browsers
- **Mail servers**: clients might be mail readers

But note the ‘might be’ — any program, written in any language, can connect to an arbitrary port

(you used to be able to have a lot of fun connecting to the sendmail port of machines such as whitehouse.gov)
Client-Server Applications

Examples include:

- **Web servers**: clients *might be* browsers
- **Mail servers**: clients *might be* mail readers

But note the ‘might be’ — any program, written in any language, can connect to an arbitrary port.

(you used to be able to have a lot of fun connecting to the sendmail port of machines such as whitehouse.gov)
Client-Server Applications

Examples include:

- **Web servers**: clients might be browsers
- **Mail servers**: clients might be mail readers

But note the ‘might be’ — any program, written in any language, can connect to an arbitrary port

(you used to be able to have a lot of fun connecting to the sendmail port of machines such as whitehouse.gov)
Creating Server Applications in Java

A server application can be created with the java.net.ServerSocket class:

This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

This is what a ServerSocket does
Nonsense — this is what the programmer makes happen
Creating Server Applications in Java

A server application can be created with the `java.net.ServerSocket` class:

From the not-very-well-written `ServerSocket` API

This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

This is what a `ServerSocket` does

Nonsense — this is what the *programmer* makes happen
A server application can be created with the `java.net.ServerSocket` class:

from the not-very-well-written ServerSocket API

This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

This is what a ServerSocket does

Nonsense — this is what the programmer makes happen
Creating Server Applications in Java

A server application can be created with the java.net.ServerSocket class:

from the not-very-well-written ServerSocket API

This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

This is what a ServerSocket does

Nonsense — this is what the programmer makes happen
Creating Server Applications in Java

A server application can be created with the `java.net.ServerSocket` class:

This class implements server sockets. A server socket *waits for requests* to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

*This is what a ServerSocket does*

Nonsense — this is what the *programmer* makes happen
A server needs a port number, so that its clients can connect to it.

Recall that IP addresses give a way of reaching a particular machine (computer).
Port numbers are ‘addresses’ for particular programs running on that machine.

The constructor

```
ServerSocket(int port)
```

in class ServerSocket

creates a server socket on the specified port number.

The port number should be greater than 1024!
A server needs a port number, so that its clients can connect to it.

Recall that IP addresses give a way of reaching a particular machine (computer)

Port numbers are ‘addresses’ for particular programs running on that machine

The constructor

```
ServerSocket(int port)
```

creates a server socket on the specified port number.

The port number should be greater than 1024!
A server needs a port number, so that its clients can connect to it.

Recall that IP addresses give a way of reaching a particular machine (computer)
Port numbers are ‘addresses’ for particular programs running on that machine

The constructor

ServerSocket(int port)

creates a server socket on the specified port number.

The port number should be greater than 1024!
A server needs a port number, so that its clients can connect to it.

Recall that IP addresses give a way of reaching a particular machine (computer).
Port numbers are ‘addresses’ for particular programs running on that machine.

The constructor

```
ServerSocket(int port)
```

creates a server socket on the specified port number.

The port number should be greater than 1024!
A server needs a port number, so that its clients can connect to it.

Recall that IP addresses give a way of reaching a particular machine (computer).

Port numbers are ‘addresses’ for particular programs running on that machine.

The constructor

ServerSocket(int port)

creates a server socket on the specified port number.

The port number should be greater than 1024!
A server needs a port number, so that its clients can connect to it.

Recall that IP addresses give a way of reaching a particular machine (computer)
Port numbers are ‘addresses’ for particular programs running on that machine

The constructor

```java
ServerSocket(int port)
```

creates a server socket on the specified port number.

The port number should be greater than 1024!
The main work of the server socket is done by:

```java
public Socket accept() throws IOException {
    ...
}
```

**API: ServerSocket#accept(int)**

Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

This method returns a `Socket`, which we know how to work with — get the I/O streams ...
The main work of the server socket is done by:

```java
public Socket accept() throws IOException {
    ...
}
```

**API: ServerSocket#accept(int)**

Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

This method returns a `Socket`, which we know how to work with — get the I/O streams …
The **ServerSocket** Class

The main work of the server socket is done by:

```java
public Socket accept() throws IOException {
    ...
}
```

**API: ServerSocket#accept(int)**

Listens for a connection to be made to this socket and **accepts it**. The method blocks until a **connection is made**.

This method returns a **Socket**, which we know how to work with — get the I/O streams . . .
Creating Server Applications in Java

Creating a server application involves:

```java
ServerSocket ss = new ServerSocket(8018);
Socket session = ss.accept();
// handle the request
```

‘Handling the request’ depends upon the desired functionality of the server — this is what the programmer must do.
Creating Server Applications in Java

Creating a server application involves:

```
ServerSocket ss = new ServerSocket(8018);
Socket session = ss.accept();
// handle the request
```

‘Handling the request’ depends upon the desired functionality of the server — this is what the programmer must do.
Example: Echo Server

As an example, we’ll create a server that simply echoes every string sent to it, and exits when a client sends it the string "BYE".

(This example is adapted from Horstman & Cornell, *Core Java, 2 vols.*, Prentice Hall, 2002, 2001.)
public static void main(String[] args) {
    try {
        ServerSocket ss = new ServerSocket(8189);
        Socket incoming = ss.accept();
    } catch (Exception e) {
        // Handle exceptions
    }
}

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

class EchoServer

import Socket and ServerSocket classes
the server will just be a program to accept and handle connections
see the API for the Exceptions that could be thrown
Example: Echo Server

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

public class EchoServer {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(8189);
            Socket incoming = ss.accept();
        }
    }
}
```

import `Socket` and `ServerSocket` classes

the server will just be a program to accept and handle connections

see the API for the `Exceptions` that could be thrown
Example: Echo Server

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

public class EchoServer {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(8189);
            Socket incoming = ss.accept();
        }
    }
}
```

import `Socket` and `ServerSocket` classes
the server will just be a program to accept and handle connections
see the API for the `Exceptions` that could be thrown
Example: Echo Server

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

class EchoServer {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(8189);
            Socket incoming = ss.accept();
        } catch (Exception e) {
            // Handle exceptions
        }
    }
}
```

The server will just be a program to accept and handle connections. See the API for the Exceptions that could be thrown.
import java.io.*;
import java.net.*;

class EchoServer {

    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(8189);
            Socket incoming = ss.accept();
        }
    }

import Socket and ServerSocket classes
the server will just be a program to accept and handle connections
see the API for the Exceptions that could be thrown
Example: Echo Server

As before, get the I/O streams.
In this case, we’re going to read and write strings.

```java
BufferedReader in =
    new BufferedReader(
        new InputStreamReader(
            incoming.getInputStream()));

PrintWriter out =
    new PrintWriter(
        new OutputStreamWriter(
            incoming.getOutputStream()));
```
Example: Echo Server

EchoServer#main(String[]), contd.

```java
BufferedReader in =
    new BufferedReader(
        new InputStreamReader(
            incoming.getInputStream()));

PrintWriter out =
    new PrintWriter(
        new OutputStreamWriter(
            incoming.getOutputStream()));
```

As before, get the I/O streams.
In this case, we’re going to read and write strings.
Example: Echo Server

```java
out.println("Hello! " +
    "Enter BYE to exit.");

// make sure this all gets sent
out.flush();
```

Sending to the remote client.

NB: this is a ‘baby’ protocol. Real protocols do not prompt the remote client — because the remote client is usually a program that has been written with the application-level protocol in mind.
Example: Echo Server

```java
EchoServer#main(String[]), contd.

out.println("Hello! " + "Enter BYE to exit.");

// make sure this all gets sent
out.flush();
```

Sending to the remote client

NB: this is a ‘baby’ protocol. Real protocols do not prompt the remote client — because the remote client is usually a program that has been written with the application-level protocol in mind.
Example: Echo Server

```java
out.println("Hello! " +
    "Enter BYE to exit.");

// make sure this all gets sent
out.flush();
```

Sending to the remote client

NB: this is a ‘baby’ protocol. **Real** protocols do *not* prompt the remote client — because the remote client is usually a program that has been written with the application-level protocol in mind.
Example: Echo Server

```java
String line;  // input from remote client
boolean done = false;

while (!done) {
    line = in.readLine();
    if ((line == null) ||
        (line.trim().equals("BYE")))
        done = true;
    else
        out.println("Echo: " + line);
        out.flush();
}
```

Exit when client closes connection or sends “BYE”
Example: Echo Server

```
String line;  // input from remote client
boolean done = false;

while (!done) {
    line = in.readLine();
    if ((line == null) ||
        (line.trim().equals("BYE")))
        done = true;
    else
        out.println("Echo: " + line);
    out.flush();
}
```

Exit when client closes connection or sends “BYE”
Example: Echo Server

String line;  // input from remote client
boolean done = false;

while (!done) {
    line = in.readLine();
    if ((line == null) ||
        (line.trim().equals("BYE")))
        done = true;
    else
        out.println("Echo: " + line);
        out.flush();
}

Exit when client closes connection or sends “BYE”
Example: Echo Server

```java
String line;  // input from remote client
boolean done = false;

while (!done) {
    line = in.readLine();
    if ((line == null) ||
        (line.trim().equals("BYE")))
        done = true;
    else
        out.println("Echo: "+ line);
        out.flush();
}
```

Exit when client closes connection or sends “BYE”
Example: Echo Server

String line;  // input from remote client
boolean done = false;

while (!done) {
    line = in.readLine();
    if ((line == null) ||
        (line.trim().equals("BYE")))
        done = true;
    else
        out.println(\"Echo: \" + line);
        out.flush();
}
Example: Echo Server

```
Incoming.close();
ss.close();
```
Example: Echo Server

```
EchoServer#main(String[]), contd.

    incoming.close();
    ss.close();

    } catch (Exception e) {
        e.printStackTrace();
    }

}
```

close the Socket

close the ServerSocket (shut down the server)

Wimping out
Example: Echo Server

```java
EchoServer#main(String[]), contd.

    incoming.close();
    ss.close();
    } catch (Exception e) { 
        e.printStackTrace();
    }
}

close the Socket
close the ServerSocket (shut down the server)
Wimping out
```
Error-handling

A server is a program that is typically run on a particular machine, on a particular port.

The program is started and left to run (and all it does is:

- wait for a connection
- accept connection;
- handle request;
- repeat)

If something goes wrong, it might be because

- there is a fault with one connection, or
- there is a local fault with the server

Try-catch blocks should handle the different Exceptions that might be thrown in appropriate ways.
Example: Echo Server

This example sets up a server on port 8189 of the computer on which the Java program is run.

To interact with it, we could write a client application along the lines of the example in the previous lecture.

Or, we could use \texttt{telnet}:

1. start the server:
   \begin{verbatim}
   java EchoServer
   \end{verbatim}

2. open a shell, possibly on a different machine

3. connect to the server:
   \begin{verbatim}
   telnet localhost 8189
   \end{verbatim}
Handling Multiple Sessions

The example handles one session, and then exits.

We could make the server keep on accepting requests by:

```java
in main()

ServerSocket ss = new ServerSocket(8189);
while (true) {
    Socket session = ss.accept();
    // handle session as before
}
```

But there are a couple of things wrong with that . . .
Handling Multiple Sessions

The example handles one session, and then exits.

We could make the server keep on accepting requests by:

```java
in main()

ServerSocket ss = new ServerSocket(8189);
while (true) {
    Socket session = ss.accept();
    // handle session as before
}
```

But there are a couple of things wrong with that . . .
Any exceptions that are thrown (e.g., an IOException while reading input from a client in one session) will cause the server to shut down — but a server should be robust: able to carry on in such cases.

How do we stop (shut down) the server?

Ideally, our server should be able to handle more than one session at a time — we will look at concurrency next week.
Any exceptions that are thrown (e.g., an IOException while reading input from a client in one session) will cause the server to shut down — but a server should be *robust*: able to carry on in such cases.

How do we stop (shut down) the server?

Ideally, our server should be able to handle more than one session at a time — we will look at concurrency next week.
Any exceptions that are thrown (e.g., an IOException while reading input from a client in one session) will cause the server to shut down — but a server should be *robust*: able to carry on in such cases.

How do we stop (shut down) the server?

Ideally, our server should be able to handle more than one session at a time — we will look at concurrency next week.
Robustness

The ServerSocket constructor and the accept() method both throw IOExceptions if there is an I/O error while setting up the ServerSocket, or while waiting for a connection; these errors are fatal, and should be handled by shutting the server down (as in the code we’ve just seen).

Reading input from the client can also throw IOExceptions if there is an I/O error while reading input from the client. Such errors are not fatal for the server, but they mean that the connection to the client has been lost. In this case, our program should simply terminate the session, and allow the server to continue with the next session for the next client that connects.
Robustness

The ServerSocket constructor and the accept() method both throw IOExceptions if there is an I/O error while setting up the ServerSocket, or while waiting for a connection; these errors are fatal, and should be handled by shutting the server down (as in the code we’ve just seen).

Reading input from the client can also throw IOExceptions if there is an I/O error while reading input from the client. Such errors are not fatal for the server, but they mean that the connection to the client has been lost. In this case, our program should simply terminate the session, and allow the server to continue with the next session for the next client that connects.
Catching IOExceptions within a Session

We introduce a new method, handleSession(), that will catch any IOExceptions from client input by simply terminating, allowing the server to get back to accepting connections.

```java
in main()

    ServerSocket ss = new ServerSocket(8189);
    while (true) {
        Socket session = ss.accept();
        handleSession(session);
    }
```

Note that the method takes the Socket instance as parameter; this represents the connection to one client.
Catching IOExceptions within a Session

We introduce a new method, handleSession(), that will catch any IOExceptions from client input by simply terminating, allowing the server to get back to accepting connections.

```java
ServerSocket ss = new ServerSocket(8189);
while (true) {
    Socket session = ss.accept();
    handleSession(session);
}
```

Note that the method takes the Socket instance as parameter; this represents the connection to one client.
Catching IOExceptions within a Session

We introduce a new method, handleSession(), that will catch any IOExceptions from client input by simply terminating, allowing the server to get back to accepting connections.

```java
in main()

ServerSocket ss = new ServerSocket(8189);
while (true) {
    Socket session = ss.accept();
    handleSession(session);
}
```

Note that the method takes the Socket instance as parameter; this represents the connection to one client.
handleSession(Socket)

static void handleSession(Socket client) {
    try {
        BufferedReader in =
            new BufferedReader(
                new InputStreamReader(
                    client.getInputStream()));
        // etc.
    } catch (IOException e) {
        System.err.println(e.getMessage());
    } finally {
        // close the socket
    }
}
The method is static because it’s called from a static context (main()).

Any IOExceptions thrown by setting up the I/O Streams, or reading from the input stream is caught within the method. The catch-block prints out the error message; the finally block closes the socket connection; and the terminates.

Flow of control then goes back to the while-loop in main().
Shutting the Server Down

We *start* the server by running the Java interpreter:

```
java EchoServer
```

We can stop the server by stopping the Java interpreter: type Ctrl-C at the command line.

This makes many programmers uneasy. (Surely we need to close the ServerSocket, to free up the port?!)

Typing Ctrl-C is fine, as the Java interpreter will close any ServerSocket-, Socket-, File-, etc., connections before it shuts down.

But still . . . .
Shutting the Server Down

We *start* the server by running the Java interpreter:

```java
java EchoServer
```

We can stop the server by stopping the Java interpreter: type Ctrl-C at the command line.

This makes many programmers uneasy. (Surely we need to close the ServerSocket, to free up the port?!)

Typing Ctrl-C is fine, as the Java interpreter will close any ServerSocket-, Socket-, File-, etc., connections before it shuts down. But still . . . .
We start the server by running the Java interpreter:

```
java EchoServer
```

We can stop the server by stopping the Java interpreter: type Ctrl-C at the command line.

This makes many programmers uneasy. (Surely we need to close the ServerSocket, to free up the port?!)

Typing Ctrl-C is fine, as the Java interpreter will close any ServerSocket-, Socket-, File-, etc., connections before it shuts down.

But still ....
**Shutting the Server Down**

We *start* the server by running the Java interpreter:

```bash
java EchoServer
```

We can stop the server by stopping the Java interpreter: type Ctrl-C at the command line.

This makes many programmers uneasy. (Surely we need to close the ServerSocket, to free up the port?!) Typing Ctrl-C is fine, as the Java interpreter will close any ServerSocket-, Socket-, File-, etc., connections before it shuts down. But still ....
Shutting the Server Down

We *start* the server by running the Java interpreter:

```
java EchoServer
```

We can stop the server by stopping the Java interpreter: type Ctrl-C at the command line.

This makes many programmers uneasy. (Surely we need to close the ServerSocket, to free up the port?!) Typing Ctrl-C is fine, as the Java interpreter will close any ServerSocket-, Socket-, File-, etc., connections before it shuts down.

But still . . . .
Stopping the Server

We could call the `ServerSocket.close()` method to close the ServerSocket and shut down the server.

The only problem is, the server is busy running its `while(true)`-loop.

A common solution is to include a secret ‘shut down’ command in the application-level protocol.

The EchoServer protocol is very simple: accept and echo user-input until the user types ‘BYE’.

We’ll let a user shut the server down by typing in the code-word ‘0q’.
Stopping the Server

We could call the `ServerSocket.close()` method to close the ServerSocket and shut down the server.

The only problem is, the server is busy running its `while(true)`-loop.

A common solution is to include a secret ‘shut down’ command in the application-level protocol.

The EchoServer protocol is very simple: accept and echo user-input until the user types ‘BYE’.

We’ll let a user shut the server down by typing in the code-word ‘0q’.
Stopping the Server

We could call the `ServerSocket.close()` method to close the ServerSocket and shut down the server.

The only problem is, the server is busy running its `while(true)`-loop.

A common solution is to include a secret ‘shut down’ command in the application-level protocol.

The EchoServer protocol is very simple: accept and echo user-input until the user types ‘BYE’.
We’ll let a user shut the server down by typing in the code-word ‘0q’.
Stopping the Server

We could call the `ServerSocket.close()` method to close the ServerSocket and shut down the server.

The only problem is, the server is busy running its `while(true)`-loop.

A common solution is to include a secret ‘shut down’ command in the application-level protocol.

The EchoServer protocol is very simple: accept and echo user-input until the user types ‘BYE’.

We’ll let a user shut the server down by typing in the code-word ‘0q’.
Shutting the Server Down

in handleSession()

if ((line == null) ||
    (line.trim().equals("BYE"))) {
    // end session
} else if (line.equals("0q")) {
    shutDown();
} else // etc.
Changes to EchoServer

in class EchoServer

private static ServerSocket ss;

public static shutDown() {
    try {
        ss.close();
    } catch (Exception e) {
        System.err.println("problem ...");
        System.err.println(e.getMessage());
    }
}

And don’t declare ss as local in main()!
Shutting Down

After calling shutDown(), the ServerSocket should be closed. The accept() method in the while(true)-loop will throw an IOException when it is next called, and the main method will end.
The java.sun tutorials recommend ending socket ‘sessions’ as follows:

- close the I/O streams
- close the socket

and give example code:

```java
in.close();
out.close();
mySocket.close();
```
Closing Sockets

The java.sun tutorials recommend ending socket ‘sessions’ as follows:

- close the I/O streams
- close the socket

and give example code:

```java
in.close();
out.close();
mySocket.close();
```
What’s wrong with that?

In the current implementation of java.net, closing the input/output streams of a socket sets that socket to null, and the example code throws a NullPointerException.

This behaviour is undocumented, and therefore possibly liable to change.
try {
    in.close();
} catch (IOException e) {
}

if (out != null)
    out.close();
}

if (client != null)
    try {
        client.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
Summary

- Sockets
- Ports
- Class **Socket**
- I/O streams
- Servers

Next:

Concurrency