Surfning

- Starta Firefox (eller Opera, eller Internet Explorer, eller ...).
- Skriv en URL i adressfältet: `http://www.w3.org/History/1989/proposal.html`.
- Webbläsaren kopplar upp sig mot webbservern på `www.w3.org` och skickar följande kommando:

```
GET /History/1989/proposal.html
```

- Webbservern hittar filen `proposal.html` och skickar tillbaka den till webbläsaren.
- Webbläsaren läser HTML-koden, tolkar kommandona (taggarna) och visar filen i fönstret.
- Samma saker händer om man inte själv skriver en URL utan klickar på en länk.

Webbklienter och webbservrar

Webbklienter (webbläsare) är komplicerade:

- ska kunna koppla upp sig mot servrar och skicka rätt HTTP-kommandon till dem,
- ska kunna tolka HTML-koden som kommer från servern och visa resultatet på skärmen: text av olika storlek, länkar, bilder, ...

Webbservrar är också komplicerade:

- ska tillåta att klienter kopplar upp sig,
- ska kunna skicka tillbaka filer när klienten begär det,
- behöver inte bry sig om innehållet i filerna.

En klient skriver man inte själv. En mycket förenklad server kan man skriva själv, så det gör vi. I Java.

Specialskrivna webbservrar

En "generell" webbserver har som huvuduppgift att skicka tillbaka dokument till klienten. Men man kan använda webben för att kommunicera med andra enheter, till exempel skrivare, switchar/routers, webbkameror, ...

Syftet kan vara att konfigurera enheten, övervaka arbetet, ...

Då har enheten en specialskriven webbserver som genererar dynamiska webbsidor.

Kommunikation med sockets

Två Unixprocesser som ska kommunicera med varandra gör det via *sockets*. Processerna kan finnas på olika datorer. En socket är "ena ändpunkten" av en kommunikationskanal. Man kan skriva på en socket eller läsa från en socket.

En server väntar på uppkopplingar från klienter genom att "lyssna" på en port med ett bestämt nummer. När en klient kopplar upp sig får man en socket som kan utnyttjas för kommunikationen. I Java:

```
ServerSocket server = new ServerSocket(portNbr); // create server
Socket connection = server.accept();             // listen
```

Klienter skapar sin egen socket genom att ange serverdatorns namn (eller IP-adress) och portnummer:

```
Socket connection = new Socket(computerName, portNbr);
```

På de följande sidorna finns dokumentation från `java.sun.com`.

ServerSocket — konstruktor

```
public ServerSocket(int port)
       throws IOException

  Creates a server socket, bound to the specified port. A port
  of 0 creates a socket on any free port.

  The maximum queue length for incoming connection indications
  (a request to connect) is set to 50. If a connection indication
  arrives when the queue is full, the connection is refused.

  Parameters:
      port - the port number, or 0 to use any free port.
  Throws:
      IOException - if an I/O error occurs when opening the socket.
      SecurityException - if a security manager exists and its
        checkListen method doesn't allow the operation.
```


ServerSocket — accept()

```
public Socket accept()
              throws IOException

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

  A new Socket is created and, if there is a security manager, ...

  Returns:
      the new Socket
  Throws:
      IOException - if an I/O error occurs when waiting for a
        connection.
      SecurityException - if a security manager exists and its
        checkListen method doesn't allow the operation.
      SocketTimeoutException - if a timeout was previously set
        with setSoTimeout and the timeout has been reached.
      IllegalBlockingModeException - if this socket has an
        associated channel, ...
```


Socket — konstruktor

```
public Socket(String host,
              int port)
       throws UnknownHostException, IOException

  Creates a stream socket and connects it to the specified port
  number on the named host.

  If the specified host is null it is the equivalent of specifying
  the address as InetAddress.getByName(null). In other words, it
  is equivalent to specifying an address of the loopback interface.

  Parameters:
      host - the host name, or null for the loopback address.
      port - the port number.
  Throws:
      UnknownHostException - if the IP address of the host could
        not be determined.
      IOException - if an I/O error occurs when creating the
        socket.
```


Strömmar för att läsa och skriva

I Unix läser man data från en input-ström och skriver på en output-ström. I Java motsvaras strömmarna av objekt av klasserna `InputStream` respektive `PrintWriter` (eller `PrintStream`). Strömmarna kan kopplas till filer, sockets, tangentbord, datorskärm, ... (`System.in` är ett `InputStream`-objekt, `System.out` är ett `PrintStream`-objekt.)

Koppla strömmar till sockets och läs/skriv:

```
Socket connection = ...;
InputStream in = connection.getInputStream();
PrintWriter out = new PrintWriter(connection.getOutputStream());
char ch = (char) in.read(); // read one character
out.println("Hello");       // write a string
```

Felhantering med exceptions

Fel som Javasystemet upptäcker rapporteras med `Exception`-objekt, ”undantagsobjekt”. Du har sett flera exempel: `FileNotFoundException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.

Man tar hand om undantag med `try catch`. Efter `catch`-blocken kan man skriva ett `finally`-block: satserna i det blocket utförs alltid, oavsett vad som inträffat tidigare.

```
try {
    // ... satser som kan ge XXXException
} catch (XXXException e) {
    // ... satser som tar hand om felet
} finally {
    // ... satser som alltid utförs
}
```


En enkel server

På nästa bild finns klassen `HelloWorldServer`. Den beskriver en server som:

- Lyssnar på port 8089.
- I all oändlighet:
  - väntar (med `accept()`) på att en klient ska koppla upp sig,
  - svarar med ”Hello, world!”.

HelloWorldServer (1)

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

class HelloWorldServer {
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(8089);
            while (true) {
                // ... vänta på uppkoppling, svara
                // ... med "Hello, world!" (nästa bild)
            }
        } catch (IOException e) {
            System.err.println("Could not start server. " +
                               "Port occupied");
        }
    }
}
```


HelloWorldServer (2)

```
ServerSocket server = new ServerSocket(8089);
while (true) {
    Socket connection = null;
    try {
        connection = server.accept();
        PrintWriter out = new PrintWriter
            (connection.getOutputStream());
        out.println("Hello, world!");
        out.close();
    } catch (IOException e) {
    } finally {
        if (connection != null) {
            connection.close();
        }
    }
}
```

En enkel klient

På nästa bild finns klassen `SimpleClient`. Den beskriver en klient som:

- kopplar upp sig mot servern som lyssnar på port 8089 på den dator där klienten körs (`localhost`),
- läser alla tecken som skickas från servern och skriver ut dem på `System.out`.

SimpleClient

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

class SimpleClient {
    public static void main(String[] args) {
        try {
            Socket connection = new Socket("localhost", 8089);
            InputStream in = connection.getInputStream();
            int chCode = in.read();
            while (chCode >= 0) {
                System.out.print((char) chCode);
                chCode = in.read();
            }
            in.close();
            connection.close();
        } catch (IOException e) {
            System.err.println("Could not connect to server");
        }
    }
}
```


SimpleWebServer2

Samma som `HelloWorldServer`, men följer HTTP-protokollet:

- Svarar alltid med en rubrik enligt HTTP-protokollet, sedan med HTML-kod (90 tecken):

```
HTTP/1.0 200 OK
Content-length: 90
Content-type: text/html

<html>
<head></head>
<body>
<H1><CENTER>Hello, world!</CENTER></H1>
</body>
</html>
```


SimpleWebServer3

Nästan samma som `SimpleWebServer2`:

- Inleder med att läsa tecknen i kommandot som klienten skickar till servern (`GET filnamn` och så vidare), skriver ut dem i kommandofönstret.
- Skickar sedan tillbaka rubriken och HTML-koden för `Hello, world!`

SimpleWebServer4

Nästan samma som `SimpleWebServer3`:

- Inleder också med att läsa kommandot som klienten skickar till servern, men plockar ut de första orden (strängarna `cmd` och `fileName`) och skriver ut dem i kommandofönstret. Orden identifieras med hjälp av ett `Scanner`-objekt.
- Skickar sedan tillbaka `Hello, world!`

SimpleWebServer5

Nästan samma som `SimpleWebServer4`:

- Letar reda på filen som efterfrågas i `GET`-kommandot och skickar tillbaka den till klienten. Alltså en "riktig" server!
- Använder Javaklassen `File` för att representera filen och en `FileInputStream` för att läsa från filen.

SimpleWebServer5 → Apache

Man måste bygga ut `SimpleWebServer5` innan man kan börja sälja klassen:

- fler kommandon (HTTP innehåller mycket mera än `GET`),
- felhantering,
- stöd för CGI, PHP, Servlets, ...,
- effektivitet (trådar, parallella processer).

Om programmering ...

- Programmeringen här är nästan trivial, bara några `while`-satser.
- Nästan all funktionalitet som vi behövde finns i färdigskrivna klasser.
- Vi har utnyttjat följande klasser:

```
ServerSocket, Socket, PrintWriter, IOException,
InputStream, Scanner, String, File, FileInputStream
```

- Mycket "riktig" programmering är av denna typ: leta i dokumentationen efter klasser som man kan utnyttja, läs på nätet, läs i böcker, ...