

# 2.7 SOCKET PROGRAMMIERUNG

---

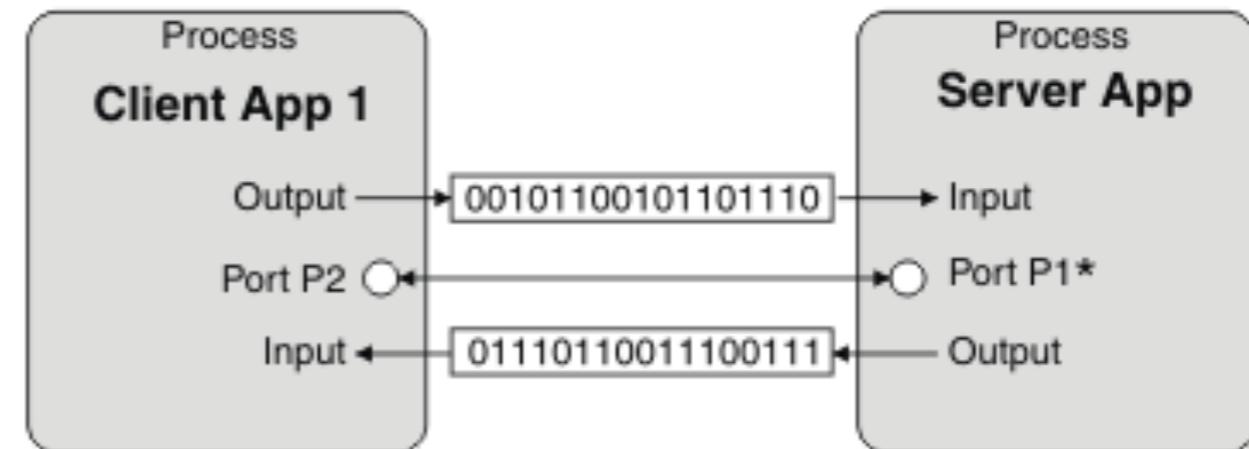
SEW 4

*DI Thomas Helml*

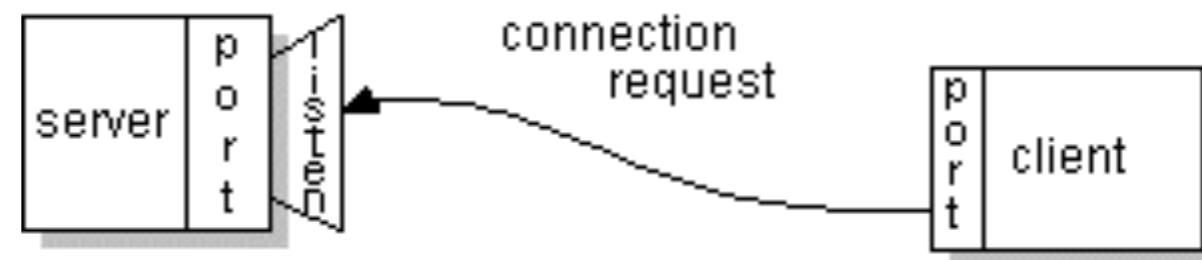


- Sockets
- Clientseite
- Serverseite
- Programmierbeispiele

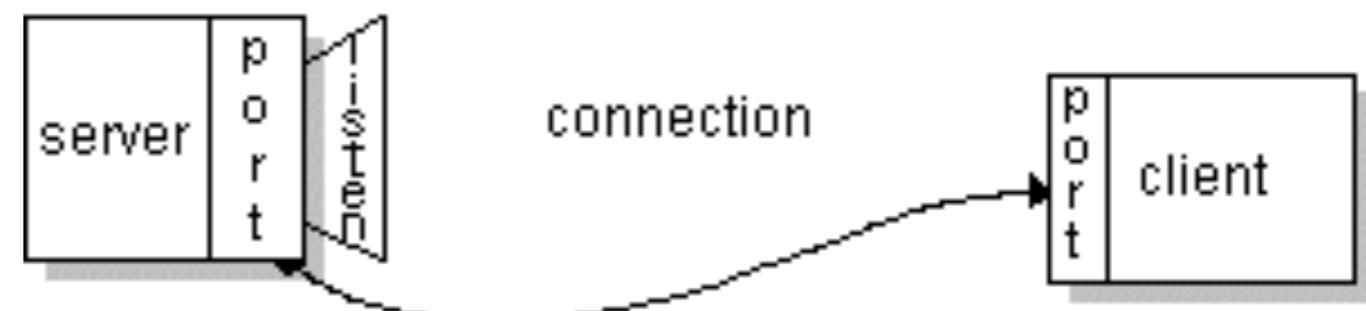
- Socket
  - repräsentiert Endpunkte einer Verbindung zw. 2 Hosts
  - vom OS zur Verfügung gestellt
  - bidirektional



- Server:
  - üblicherweise eigener Rechner
  - Socket, der auf bestimmten Port gebunden ist
  - wartet auf eingehende Verbindungen



- Server:
  - alles ok -> Verbindung wird akzeptiert
  - Server bekommt neuen Socket für die akzeptierte Verbindung
  - „Ursprünglicher“ Socket lauscht weiter auf eingehende Verbindungen



- Client:
  - Socket erstellen - Verbindungsauftbau zu Server
  - wird Verbindung akzeptiert, so kann der Client über diesen Socket mit Server kommunizieren
  - Client+Server können jeweils am Socket lesen/schreiben

- Unterscheide Client- und Server-Seite!
- Client:
  - Klasse `java.net.Socket`
- Server:
  - Klasse `java.net.ServerSocket`

## 1. Socket erzeugen

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

## 2. Input stream & Output stream am Socket öffnen

```
InputStream getInputStream() throws IOException
```

```
OutputStream getOutputStream() throws IOException
```

## 3. Lesen/Schreiben in den Stream – abhängig vom Protokoll des Servers

## 4. Streams schließen

## 5. Socket schließen

```
void close() throws IOException
```

1. ServerSocket (Port!) öffnen (event. Timeout)

```
ServerSocket(int port) throws IOException
```

2. auf Verbindung warten (blockierend!) -> Client Socket

```
Socket accept() throws IOException
```

3. Kommunizieren über Client Socket (Thread?)

4. Input stream & Output stream am Socket öffnen

```
InputStream getInputStream() throws IOException
```

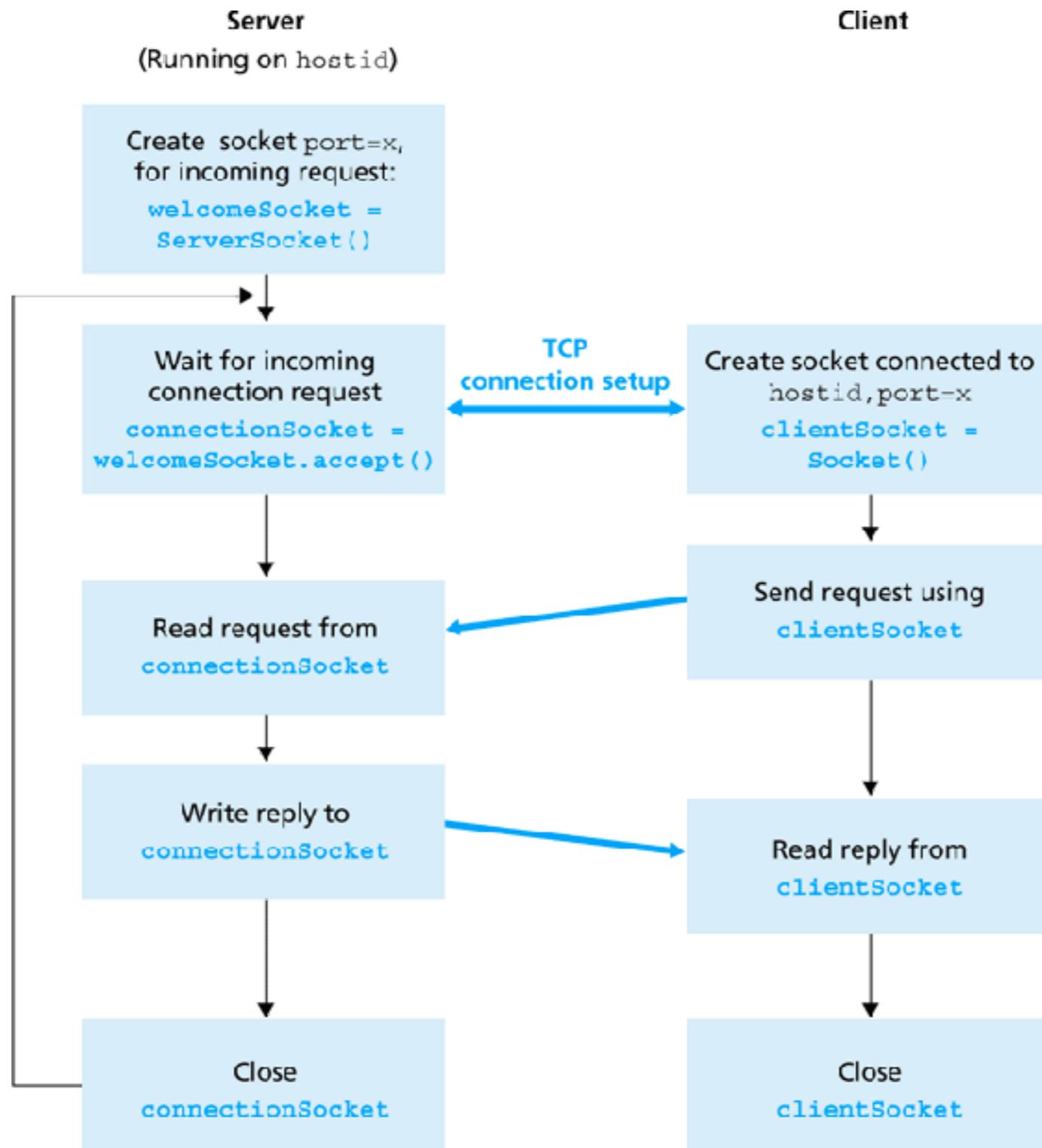
```
OutputStream getOutputStream() throws IOException
```

5. Lesen/Schreiben in den Stream

6. Streams schließen

7. Socket schließen

```
void close() throws IOException
```



Ausgabe-Streams		Eingabe-Streams	
Byte-Streams	Character-Streams	Byte-Streams	Character-Streams
OutputStream	Writer	InputStream	Reader
FileOutputStream	FileWriter	FileInputStream	FileReader
BufferedOutputStream	BufferedWriter	BufferedInputStream	BufferedReader
ByteArrayOutputStream	CharArrayWriter	ByteArrayInputStream	CharArrayReader
FilterOutputStream	FilterWriter	FilterInputStream	FilterReader
PipedOutputStream	PipedWriter	PipedInputStream	PipedReader
PrintStream	PrintWriter		
		PushbackInputStream	PushbackReader

- DataInputStream + DataOutputStream
  - für Senden/Empfangen von Basisdatentypen!
  - writeInt, writeDouble, ...
  - readInt, readDouble, ...
- ObjectInputStream + ObjectOutputStream
  - für Senden/Empfangen von Objekten
  - readObject
  - writeObject

# AUFGABE 1

---

- Server für Multiplikationen
  - Client baut Verbindung auf
  - und schickt 2 Zahlen
  - Server multipliziert die Werte und schickt Ergebnis retour

- Echo Server
  - Server öffnet Socket für Verbindungsanfragen
  - Ausgabe der lokalen Socket Adresse
  - eingehende Verbindungen werden in EIGENEM Thread mit neuem Socket bearbeitet
  - Client nimmt Verbindung auf
  - Ausgabe der entfernten Socket Adresse
  - Client schickt Textnachricht
  - Server schickt „Echo“ retour an Client

```
// we use a cached thread pool for performance
// ClientHandler will not be created only once!
ExecutorService executorService = Executors.newCachedThreadPool();

// open server socket
try (ServerSocket serverSocket = new ServerSocket(port)){

    while (true) {
        // listening for new clients -> get a client socket
        Socket client = serverSocket.accept();

        // start thread for THIS new connection (client socket),
        // where communication is handled
        executorService.execute(new ClientHandler(client));
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

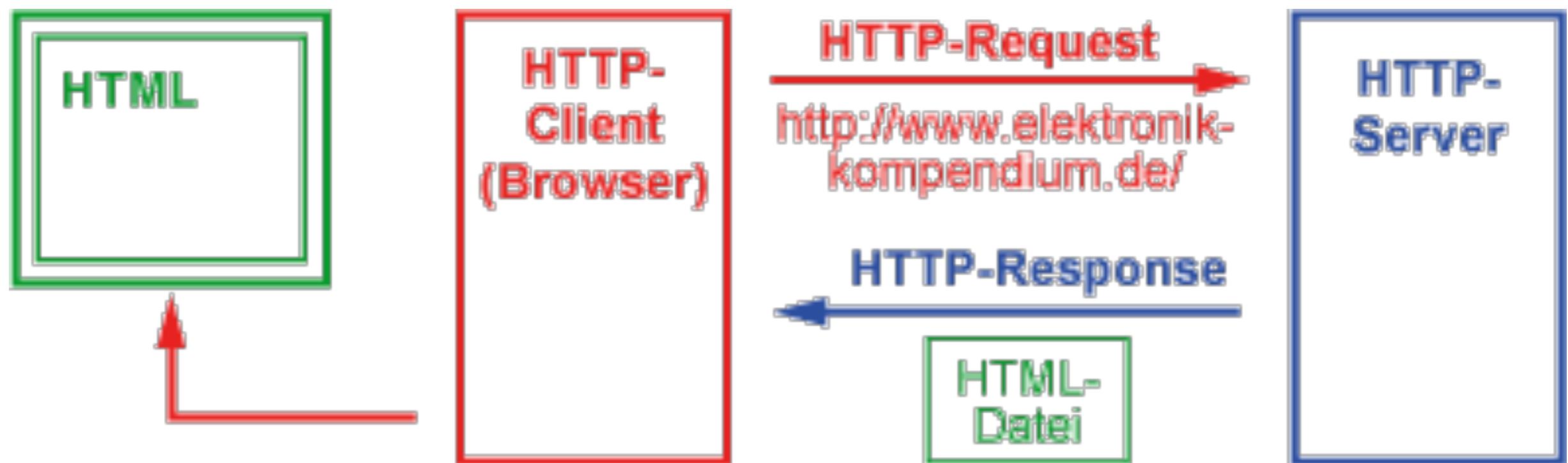
```
public class ClientHandler implements Runnable {  
  
    private Socket client;  
  
    public ClientHandler(Socket client) {  
        this.client = client;  
    }  
  
    @Override  
    public void run() {  
        // get input + output streams for sending/receiving,  
        // then communicate...  
  
        ...  
        finally {  
            try {  
                client.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

- Echo Server
  - ergänze Aufgabe 2 um eine Multithreaded-Variante
  - benutze ThreadPools

## ► HTTP Client/Server

1. Server wartet auf eingehende HTTP Anfragen
2. Client erzeugt einen URL: `http://...`
3. Client versucht, TCP Verbindungen zum Server aufzubauen
4. Server akzeptiert Verbindungswunsch
5. Client schickt Nachricht (HTTP-Anfrage) und fordert Ressource mit spezifizierter URL an
6. Server verarbeitet Anfrage (generiert u.U. HTML Seite)
7. Server schickt Rückantwort (HTTP-Antwort) mit Ressource oder Fehlercode
8. Client verarbeitet Antwort
9. Client und/oder Server schließen TCP Verbindung

# AUFGABE 3



# AUFGABE 3

## ► HTTP Request

Methode	URL	Version	CR	LF
---------	-----	---------	----	----

Parametername :	Wert	CR	LF
-----------------	------	----	----

⋮

Parametername :	Wert	CR	LF
-----------------	------	----	----

CR	LF
----	----

Sendedaten bei Post-Methode

Connection: close

Trotz HTTP/1.1 soll die Verbindung nach dem Reply geschlossen werden

User-agent: xxx

Gibt den Namen des verwendeten Browsers und seine Versionsnummer an

Accept: text/html...

Gibt an, welche Dateiformate der Browser als Antwort akzeptiert

Accept-language: de

Gibt die bevorzugte Sprache des Client an. Der Server kann darauf mit einer Seite in der richtigen Sprache reagieren

Cookie: xxxx

Der Client hatte zuvor ein Cookie vom Server erhalten. Er sendet es jetzt mit jedem weiteren Request an diesen Server mit. So kann der Server einzelne Requests immer sicher einem bestimmten Kunden zuordnen, z.B. für einen Warenkorb.

Authorization: xxx

Sobald der Server für eine Seite einen Benutzernamen und Passwort verlangt, werden diese mit jedem folgenden Request an diesen Server mitgeschickt.

# AUFGABE 3

## ► Response

Version	Statusnr	Statustext	CR	LF
---------	----------	------------	----	----

Parametername	:	Wert	CR	LF
---------------	---	------	----	----

⋮

Parametername	:	Wert	CR	LF
---------------	---	------	----	----

CR	LF
----	----

Daten des abgerufenen Dokuments, z.B. HTML-Seite, GIF-Bild...
--

200 OK

Alles klar, die angeforderte Datei ist im Paket enthalten

301 Moved  
Permanently

Objekt wurde verschoben. Die neue Adresse steht im Parameterfeld "Location:". Der Client lädt diese automatisch

400 Bad Request

Der Server hat den HTTP-Request nicht verstanden

401 Authorization  
Required

Der Zugriff auf die Seite erfordert die Freischaltung über einen Benutzernamen und ein Passwort. Der Browser erfragt dies vom Anwender und sendet den HTTP-Request mit diesen Angaben erneut zum Server. Jeder weitere ab diesem Zeitpunkt zum gleichen Server gesendete HTTP-Request enthält die Autorisierungsdaten ebenfalls

404 Not Found

Das angeforderte Dokument existiert nicht

Request:

```
get /index.php HTTP/1.1
Host: localhost:8888
```

...

Response:

```
HTTP/1.1 200 OK
Date: Thu, 12 Nov 2015 07:27:35 GMT
Server: Apache
X-Powered-By: PHP/5.6.10
Content-Length: 1850
Content-Type: text/html; charset=UTF-8
```

- Test mit Telnet:

```
telnet www.htr-wels.at 80
Trying 10.34.57.251...
Connected to www.htr-wels.at.
Escape character is '^]'.
```

**GET / HTTP/1.1**  
**HOST: www.htr-wels.at**

*HTTP Request*

---

```
HTTP/1.1 302 Found
Date: Mon, 12 Nov 2018 10:33:31 GMT
Server: Apache/2.4.18 (Ubuntu)
Location: https://www.htr-wels.at/
Content-Length: 289
Content-Type: text/html; charset=iso-8859-1
```

*HTTP Response*

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://www.htr-wels.at/">here</a>.</p>
<hr>
<address>Apache/2.4.18 (Ubuntu) Server at www.htr-wels.at Port 80</address>
</body></html>
```

---

Connection closed by foreign host.

- Implementiere einen HTTP Server
  - Primitiver Webserver, der nur HTTP Methode GET versteht, Query Strings aber nicht verarbeitet
  - Status 200, 400, 404, 501
  - Test mit Webbrowser
  - Konstruktor bekommt:
    - Wurzel des Webverzeichnis, die alle abrufbaren Ressourcen enthält (Verzeichnis www)
    - Portnummer

- Client
  - Einfacher Webclient, der nur die HTTP Methode GET versteht
  - versucht Index Datei auszulesen
  - gibt Kommunikation in der Konsole aus

- URL
- URLConnection
- InetAddress (Inet4Address + Inet6Address)
- NetworkInterface

- <https://docs.oracle.com/javase/tutorial/networking/sockets/index.html>
- <http://phoenix.goucher.edu/~kelliher/s2011/cs325/>
- HERDT Buch Java 8 Fortgeschrittene Programmierung