Informatik B - Objektorientierte Programmierung in Java

Vorlesung 22: Netzwerkprogrammierung/ Kommunikation 1

Inhalt

- Netzwerkprogrammierung/ Kommunikation (Teil 1)
  - Grundlagen der Netzwerkprogrammierung
  - TCP/IP und UDP/IP
  - IP-Adressen
  - Client/Server
  - Package **`java.net`**: **`InetAddress`**
  - Sockets
  - Beispiele
  - Zugriff auf einen Webserver
  - EchoServer
- Siehe auch Krüger: Handbuch der Java-Programmierung, Kapitel 45; **`www.javabuch.de`**

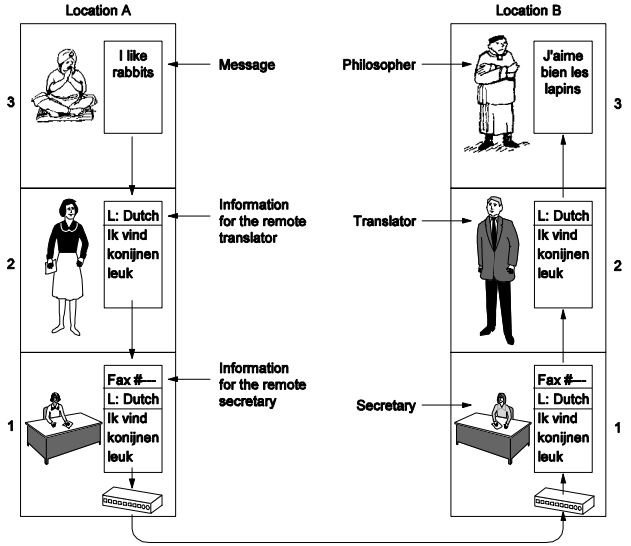
Was ist ein Netzwerk?

- Verbindung zwischen zwei oder mehr Computern, um Daten auszutauschen
- Beispiele für Netzwerk-Anwendungen
  - Entfernter Dateizugriff
  - Gemeinsame Benutzung eines Druckers
  - Unternehmensweiter Informationszugriff/-austausch
  - Email/Chat über das Internet
  - Remote login auf anderem Rechner
  - Clusterung von Computern zur Leistungssteigerung
  - Etc.

Protokolle

- Zum Datenaustausch müssen sich die Partner auf ein gemeinsames Protokoll geeinigt haben.
- Als *Protokoll* bezeichnet man die Menge aller Regeln, die notwendig sind, um einen kontrollierten und eindeutigen Verbindungsaufbau, Datenaustausch und Verbindungsabbau gewährleisten zu können. Es gibt sehr viele Protokolle mit sehr unterschiedlichen Ansprüchen.

Beispiel für Protokoll-Schichten



- Die Architektur Philosoph-Übersetzer-Sekretärin.

Netzwerkfähigkeiten von Java

- Basieren auf den Internet-Protokollen
  - TCP/IP
  - UDP/IP
- IP - Internet Protokoll
- TCP - Transmission Control Protocol
- UDP - User Datagramm Protocol

4-Schichten-Modell von TCP/IP

Schicht 4: Anwendungsschicht	z.B. FTP, HTTP, TELNET
Schicht 3: Transportschicht	TCP bzw. UDP
Schicht 2: Netzwerkschicht	IP
Schicht 1: Verbindungsschicht	z.B. Ethernet

Exkurs: Das ISO-OSI-Referenzmodell

- OSI – Open Systems Interconnection
- 7 Schichten
  - Die Bitübertragungsschicht (Physical Layer)
  - Die Sicherungsschicht (Data Link Layer)
  - Die Vermittlungsschicht (Network Layer)
  - Die Transportschicht (Transport Layer)
  - Die Sitzungsschicht (Session Layer)
  - Die Darstellungsschicht (Presentation Layer)
  - Die Anwendungsschicht (Application Layer)

TCP und UDP

- TCP ist ein *verbindungsorientiertes* Protokoll, das auf der Basis von IP eine sichere und fehlerfreie Punkt-zu-Punkt-Verbindung realisiert. (Analogie: Telefonieren)
- Daneben gibt es ein weiteres Protokoll oberhalb von IP, das als *UDP* (*User Datagram Protocol*) bezeichnet wird. Im Gegensatz zu TCP ist UDP ein *verbindungsloses* Protokoll, bei dem die Anwendung selbst dafür sorgen muss, dass die Pakete überhaupt und in der richtigen Reihenfolge beim Empfänger ankommen. Sein Vorteil ist die größere Geschwindigkeit gegenüber TCP. In Java wird UDP durch die Klassen **`DatagramPacket`** und **`DatagramSocket`** abgebildet. (Analogie: Brief)

TCP

- ein Prozess auf einem Rechner wartet auf einen Kommunikationswunsch
- ein weiterer Prozess meldet seinen Kommunikationswunsch beim ersten Prozess an
- die Verbindung zwischen den beiden Partnern wird aufgebaut
- die Prozesse tauschen Nachrichten aus
- bei Übertragungsfehlern werden Nachrichten ohne eigene Aktion der Prozesse automatisch wieder angefordert
- am Ende der Kommunikation wird die Verbindung wieder abgebaut
- Die automatische Fehlerkorrektur vermehrt den Datenverkehr im Netz und kostet Bandbreite.

Adressierung von Daten

- Um Daten über ein Netzwerk zu transportieren, ist eine *Adressierung* dieser Daten notwendig. Der Absender muss angeben können, an wen die Daten zu senden sind, und der Empfänger muss erkennen können, von wem sie stammen.
- Die Adressierung in TCP/IP-Netzen erfolgt auf der IP-Ebene zur Zeit bei der Version IPv4 mit Hilfe einer 4 Byte = 32-Bit langen *IP-Adresse*.
- Zukünftig ist vorgesehen, dass bei IPv6 16 Byte für die Adresse zur Verfügung stehen.

IP-Adressen (IPv4)

- IP-Adressen sind 32-Bit-Zahlen, die normalerweise in Dezimaldarstellung mit Punkten geschrieben werden (Dotted Decimal Notation). In diesem Format werden alle 4 Byte dezimal geschrieben, von 0 bis 255. Die niedrigste IP-Adresse ist 0.0.0.0 und die höchste 255.255.255.255.
- Beispiel (Dotted Decimal und binär):
- 193.25.16.164
- **`11000001 00011001 00010000 10100100`**

Domain-Namen

- Während IP-Adressen für Computer sehr leicht zu verarbeiten sind, gilt das nicht unbedingt für die Menschen, die damit arbeiten müssen. Wer kennt schon die Telefonnummern und Ortsnetzkennzahlen von allen Leuten, mit denen er Telefonate zu führen pflegt? Um die Handhabung der IP-Adressen zu vereinfachen, wurde daher das *Domain Name System* eingeführt (kurz *DNS*), das numerischen IP-Adressen sprechende Namen wie **`www.fh-dortmund.de`**, **`www.gkrueger.com`** oder **`java.sun.com`** zuordnet.

Domain-Namen (2)

- Anstelle der IP-Adresse können bei den Anwendungsprotokollen nun wahlweise die symbolischen Namen verwendet werden. Sie werden mit Hilfe von *Name-Servern* in die zugehörige IP-Adresse übersetzt, bevor die Verbindung aufgebaut wird. Hinter ein und derselben IP-Adresse kann sich mehr als ein Name-Server-Eintrag befinden.

Client/Server

- Die Kommunikation zwischen zwei Rechnern läuft oft auf der Basis einer *Client-Server-Beziehung* ab. Dabei kommen den beteiligten Rechnern unterschiedliche Rollen zu:
  - Der Server stellt einen Dienst zur Verfügung, der von anderen Rechnern genutzt werden kann. Er läuft im Hintergrund und wartet darauf, dass ein anderer Rechner eine Verbindung zu ihm aufbaut. Der Server definiert das Protokoll, mit dessen Hilfe der Datenaustausch erfolgt.
  - Ein Client ist der Nutzer von Diensten eines oder mehrerer Server. Er kennt die verfügbaren Server und ihre Adressen und baut bei Bedarf die Verbindung zu ihnen auf. Der Client hält sich an das vom Server vorgegebene Protokoll, um die Daten auszutauschen.

Beispiel: WWW

- Ein typisches Beispiel für eine Client-Server-Verbindung ist der Seitenabruf im World Wide Web. Der Browser fungiert als Client, der nach Aufforderung durch den Anwender eine Verbindung zum Web-Server aufbaut und eine Seite anfordert. Diese wird vom Server von seiner Festplatte geladen oder dynamisch generiert und an den Browser übertragen. Dieser analysiert die Seite und stellt sie auf dem Bildschirm dar. Enthält die Seite Image-, Applet- oder Frame-Dateien, werden sie in gleicher Weise beim Server abgerufen und in die Seite integriert.

Ports

- Auf einem Host laufen meist unterschiedliche Serveranwendungen, die noch dazu von mehreren Clients gleichzeitig benutzt werden können. Um die Server voneinander unterscheiden zu können, gibt es ein weiteres Adressmerkmal, die *Port-Nummer*. Sie wird oberhalb von IP auf der Ebene des Transportprotokolls definiert (also in TCP bzw. UDP) und gibt die Server-Anwendung an, mit der ein Client kommunizieren will. Port-Nummern sind positive Ganzzahlen im Bereich von 0 bis 65535. Port-Nummern im Bereich von 0 bis 1023 sind für Anwendungen mit Superuser-Rechten reserviert. Jeder Servertyp hat seine eigene Port-Nummer, viele davon sind zu Quasi-Standards geworden. So läuft beispielsweise ein SMTP-Server meist auf Port 25, ein FTP-Server auf Port 21 und ein HTTP-Server auf Port 80.
- Beispiel: **`www.fh-dortmund.de:80`**

Adressierung

- des Rechners durch IP-Adresse
- eines Prozesses auf einem Rechner durch Port-Nummer(n)
- dazu muss der Prozess eine oder mehrere Port-Nummern vom Betriebssystem beantragen
- ein Client, der mit einem Server kommunizieren will, muss somit IP-Adresse und Port-Nummer des Server-Rechners und -Prozesses kennen
- der Server kann seinerseits IP-Adresse und Port-Nummer des Clients am Datenpaket (UDP) oder Verbindungsaufbau (TCP) erkennen

Package `java.net`

- **`java.net`** enthält die Unterstützung von Java für die Kommunikation mittels TCP/IP.
- Kommunikation bildet ferner die Grundlage für
  - das Nachladen von Java-Klassen,
  - Client/Server-Applikationen,
  - RMI (Remote Method Invokation),
  - verteilte Anwendungen.
- Zur Adressierung von Rechnern im Netz wird die Klasse **`InetAddress`** des Pakets **`java.net`** verwendet.

Die Klasse `InetAddress`

- Ein **`InetAddress`**-Objekt enthält sowohl eine IP-Adresse als auch den symbolischen Namen des jeweiligen Rechners. Die beiden Bestandteile können mit den Methoden **`getHostName`** und **`getHostAddress`** abgefragt werden.
  - **`String getHostName()`** liest aus einem Objekt der Klasse **`InetAddress`** den zugehörigen Rechernamen
  - **`String getHostAddress()`** liest aus einem Objekt der Klasse **`InetAddress`** die zugehörige Rechner-IP-Adresse
- Mit Hilfe von **`getAddress`** kann die IP-Adresse auch direkt als **`byte`**-Array mit vier Elementen beschafft werden:
  - **`byte[] getAddress()`**

Die Klasse `InetAddress` (2)

Um ein **`InetAddress`**-Objekt zu generieren:

- **`static InetAddress getLocalHost()`** liefert IP-Adresse des eigenen Rechners
- **`static InetAddress getByName(String hostname)`** und
- **`static InetAddress[] getAllByName(String hostname)`** ermittelt eine/alle IP-Adressen des Rechners, dessen Name in der String-Variablen **`hostname`** enthalten ist.
- Exception: **`UnknownHostException`**

Beispiel: `InternetAdresse.java`

```
import java.net.*;
public class InternetAdresse {
  public static void main( String args[] ) {
    try {
      InetAddress iaddr = InetAddress.getByName(
                              "www.fh-dortmund.de" );
      System.out.println( iaddr.getHostName() + " : " +
                iaddr.getHostAddress() );
    }
    catch (UnknownHostException e)
      { System.err.println( "Host nicht bekannt" ); }
  }
}
```

Ausgabe der IP-Adresse: 193.25.16.164

Zum Testen: Localhost

- **`localhost`** ist eine Pseudo-Adresse für den eigenen Host. Sie ermöglicht das Testen von Netzwerkanwendungen, auch wenn keine wirkliche Netzwerkverbindung besteht. IP-Adresse des localhost: 127.0.0.1
- Testen Sie das Programm **`InternetAdresse.java`** mit „localhost“.

Definition: Socket

- Als *Socket* bezeichnet man eine Programmierschnittstelle zur Kommunikation zweier Rechner in einem TCP/IP-Netz. Sockets wurden Anfang der achtziger Jahre für die Programmiersprache C entwickelt und mit Berkeley UNIX 4.1/4.2 allgemein eingeführt. Das Übertragen von Daten über eine TCP-Socket-Verbindung ähnelt dem Zugriff auf eine Datei:
  - Zunächst wird eine Verbindung aufgebaut.
  - Dann werden Daten gelesen und/oder geschrieben.
  - Schließlich wird die Verbindung wieder abgebaut.

Client-Socket (TCP)

- Während die Socket-Programmierung in C eine etwas mühsame Angelegenheit war, ist es in Java recht einfach geworden. Im wesentlichen sind dazu die beiden Klassen **`Socket`** und **`ServerSocket`** erforderlich. Sie repräsentieren Sockets aus der Sicht einer Client- bzw. Server-Anwendung. Nachfolgend wollen wir uns zunächst mit den Client-Sockets beschäftigen, die Klasse **`ServerSocket`** wird danach behandelt.

Aufbau einer einfachen Socket-Verbindung

- Die Klasse **`Socket`** besitzt verschiedene Konstruktoren, mit denen ein neuer Socket erzeugt werden kann. Die wichtigsten von ihnen sind:
  - **`public Socket(String host, int port) throws UnknownHostException, IOException`**
  - **`public Socket(InetAddress address, int port) throws IOException`**
- der Client versucht damit, eine Verbindung zum angegebenen Rechner und Port aufzubauen, der Thread wird so lange blockiert, bis eine Verbindung zustande kommt oder ein Timeout eintritt.

Aufbau einer Socket-Verbindung (2)

- Beide Konstruktoren erwarten als erstes Argument die Übergabe des Hostnamens, zu dem eine Verbindung aufgebaut werden soll. Dieser kann entweder als Domainname in Form eines Strings oder als Objekt des Typs **`InetAddress`** übergeben werden. Soll eine Adresse mehrfach verwendet werden, ist es besser, die zweite Variante zu verwenden. In diesem Fall kann das übergebene **`InetAddress`**-Objekt wiederverwendet werden, und die Adressauflösung muss nur einmal erfolgen. Wenn der Socket nicht geöffnet werden konnte, gibt es eine Ausnahme des Typs **`IOException`** bzw. **`UnknownHostException`** (wenn das angegebene Zielsystem nicht angesprochen werden konnte).

Aufbau einer Socket-Verbindung (3)

- Nachdem die Socket-Verbindung erfolgreich aufgebaut wurde, kann mit den beiden Methoden **`getInputStream`** und **`getOutputStream`** je ein Stream zum Empfangen und Versenden von Daten beschafft werden:
  - **`public InputStream getInputStream() throws IOException`**
  - **`public OutputStream getOutputStream() throws IOException`**
- Diese Streams können entweder direkt verwendet oder mit Hilfe der Filterstreams in einen bequemer zu verwendenden Streamtyp geschachtelt werden. Nach Ende der Kommunikation sollten sowohl die Eingabe- und Ausgabestreams als auch der Socket selbst mit **`close`** geschlossen werden.

Beispiel (Teil 1)

- Es soll der String „Hello, world!“ vom Client-Prozess zum Server-Prozess übertragen und dort ausgegeben werden.
- Festlegung von Client und Server auf gemeinsamen Port:

  localhost:4711

Beispiel (Teil 2): `Client.java`

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

public class Client {

   public static void main( String argv[] ) {
      try {
       Socket s = new Socket( "localhost", 4711 );
       OutputStream o = s.getOutputStream();
       OutputStreamWriter out = new OutputStreamWriter(o);
       out.write( "Hello, world!" );
       out.close();
       o.close();
       s.close();
      }
      // bitte umblättern
```


Beispiel (Teil 2): `Client.java`

```
// Fortsetzung

      catch ( UnknownHostException e ) {
         System.err.println( "Client: Unknown Host " +
                                        e.getMessage() );
         System.exit( 1 );
      }
      catch ( IOException e ) {
         System.err.println( "Client: IO-Fehler " +
                                        e.getMessage() );
         System.exit( 1 );
      }
   }
}
```


Die Klasse `ServerSockets` (1)

- Bisher haben wir uns mit dem Entwurf von *Netzwerk-Clients* beschäftigt. Nun wollen wir uns das passende Gegenstück ansehen, uns also mit der Entwicklung von *Servern* beschäftigen. Glücklicherweise ist auch das in Java recht einfach. Der wesentliche Unterschied liegt in der Art des Verbindungsaufbaus, für den es eine spezielle Klasse **`ServerSocket`** gibt. Diese Klasse stellt Methoden zur Verfügung, um auf einen eingehenden Verbindungswunsch zu warten und nach erfolgtem Verbindungsaufbau einen **`Socket`** zur Kommunikation mit dem Client zurückzugeben. Bei der Klasse **`ServerSocket`** sind im wesentlichen der Konstruktor und die Methode **`accept`** von Interesse:

Die Klasse ServerSockets (2)

```
public ServerSocket(int port) throws IOException
public Socket accept() throws IOException
```

- Der Konstruktor erzeugt einen **ServerSocket** für einen bestimmten Port, also einen bestimmten Typ von Serveranwendung. Anschließend wird die Methode **accept** aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten. **accept** blockiert so lange, bis sich ein Client bei der Serveranwendung anmeldet (also einen Verbindungsaufbau zu unserem Host unter der Portnummer, die im Konstruktor angegeben wurde, initiiert). Ist der Verbindungsaufbau erfolgreich, liefert **accept** ein **Socket**-Objekt, das wie bei einer Client-Anwendung zur Kommunikation mit der Gegenseite verwendet werden kann. Anschließend steht der **ServerSocket** für einen weiteren Verbindungsaufbau zur Verfügung oder kann mit **close** geschlossen werden.

Beispiel (Teil 3): Server.java

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

public class Server {
   public static void main( String argv[] ) {
     int input;
     try {
       ServerSocket ss = new ServerSocket( 4711 );
       Socket       s  = ss.accept();
       InputStream  i  = s.getInputStream();
       InputStreamReader in = new InputStreamReader(i);
       while ((input = in.read()) != -1)
       {
         System.out.print ( (char)input );
       }
       // bitte umblättern
```


Beispiel (Teil 3): Server.java

```
// Fortsetzung

         in.close();
         i.close();
         s.close();
         ss.close();
      }
      catch ( IOException e ) {
         System.err.println( "Client: IO-Fehler " +
                                        e.getMessage() );
         System.exit( 1 );
      }
   }
}
```


Zusammenfassung des Beispiels

- Server meldet sich auf Port 4711 an, wartet auf Verbindungsaufbau (**ServerSocket**)
- Client meldet Verbindungswunsch an (**Socket**)
- Server akzeptiert Verbindungswunsch (**accept**), erhält damit zugleich einen Socket für die Verbindung
- Client und Server warten bei **Socket** bzw. **accept**, bis Verbindung zustande gekommen ist
- durch die Sockets kann auf einen **Input-** bzw. **OutputStream** zugegriffen werden, darüber kann dann die Kommunikation abgewickelt werden
- Am Ende werden die Ressourcen Socket, ServerSocket und die Streams wieder freigegeben.

Beispiel: Zugriff auf einen Web-Server über das HTTP-Protokoll

- Die Kommunikation mit einem Web-Server erfolgt über das HTTP-Protokoll. Ein Web-Server läuft normalerweise auf TCP-Port 80 und kann wie jeder andere Server über einen Client-Socket angesprochen werden.

Bsp: Zugriff auf einen Web-Server über das HTTP-Protokoll (2)

- Wir wollen an dieser Stelle nicht auf Details eingehen, sondern nur die einfachste und wichtigste Anwendung eines Web-Servers zeigen, nämlich das Übertragen einer Seite. Ein Web-Server ist in seinen Grundfunktionen ein recht einfaches Programm, dessen Hauptaufgabe darin besteht, angeforderte Seiten an seine Clients zu versenden. Kompliziert wird er vor allem durch die Vielzahl der mittlerweile eingebauten Zusatzfunktionen, wie beispielsweise Logging, Server-Scripting, Server-Side-Includes, Security- und Tuning-Features usw.

Bsp: Zugriff auf einen Web-Server über das HTTP-Protokoll (3)

- Fordert ein Anwender in seinem Web-Browser eine Seite an, so wird diese Anfrage vom Browser als **`GET`**-Transaktion an den Server geschickt.
- Um beispielsweise die Seite **`http://www.inf.fh-dortmund.de/index.html`** zu laden, wird folgendes Kommando als String an den Server **`www.inf.fh-dortmund.de`** gesendet:

```
GET /index.html
```


Bsp: Zugriff auf einen Web-Server über das HTTP-Protokoll (4)

- Der erste Teil gibt den Kommandonamen an, dann folgt die gewünschte Datei. Die Zeile muss mit einer CRLF-Sequenz abgeschlossen werden, ein einfaches '**`\n`**' reicht nicht aus. Der Server versucht nun die angegebene Datei zu laden und überträgt sie an den Client. Ist der Client ein Web-Browser, wird er den darin befindlichen HTML-Code interpretieren und auf dem Bildschirm anzeigen. Befinden sich in der Seite Verweise auf Images, Applets oder Frames, so fordert der Browser die fehlenden Dateien in weiteren **`GET`**-Transaktionen von deren Servern ab.

Bsp: Zugriff auf einen Web-Server über das HTTP-Protokoll (5)

- Die Struktur des **`GET`**-Kommandos wurde mit der Einführung von **HTTP 1.0** etwas erweitert. Zusätzlich wird nun am Ende der Zeile eine Versionskennung mitgeschickt. Es folgt eine leere Zeile (also ein alleinstehendes CRLF), um das Kommandoende anzuzeigen. **HTTP 1.0** ist weit verbreitet, und das obige Kommando würde von den meisten Browsern in folgender Form gesendet werden (jede der beiden Zeilen muss mit CRLF abgeschlossen werden):

```
GET /index.html HTTP/1.0
<Leerzeile>
```


Bsp: Zugriff auf einen Web-Server über das HTTP-Protokoll (6)

- Wird **HTTP/1.0** verwendet, ist auch die Antwort des Servers etwas komplexer. Anstatt lediglich den Inhalt der Datei zu senden, liefert der Server seinerseits einige Header-Zeilen mit Zusatzinformationen, wie beispielsweise den Server-Typ, das Datum der letzten Änderung oder den MIME-Typ der Datei (MIME=Multi-Purpose Internet Mail Extensions). Auch hier ist jede Header-Zeile mit einem CRLF abgeschlossen, und nach der letzten Header-Zeile folgt eine Leerzeile. Erst dann beginnt der eigentliche Dateiinhalt.

Bsp: Zugriff auf einen Web-Server über das HTTP-Protokoll (7)

- Das Programm **`WebClient.java`** kann dazu verwendet werden, eine Datei von einem Web-Server zu laden. Es wird mit einem Host- und einem Dateinamen als Argument aufgerufen und lädt die Seite vom angegebenen Server. Das Ergebnis wird (mit allen Header-Zeilen) auf dem Bildschirm angezeigt.

- Beispiel-Aufruf:

```
java WebClient www.inf.fh-dortmund.de /index.html
```


Bsp: Zugriff auf einen Web-Server über das HTTP-Protokoll (8)

```
class WebClient {
...
   try {
      Socket sock = new Socket(args[0], 80);
      OutputStream out = sock.getOutputStream();
      InputStream in = sock.getInputStream();
      //GET-Kommando senden
      String s = "GET "+args[1]+"HTTP/1.0"+"\r\n\r\n";
      out.write(s.getBytes());
      //Ausgabe lesen und anzeigen
      int len;
      byte[] b = new byte[100];
      while ((len = in.read(b)) != -1) {
        System.out.write(b, 0, len);
      }
```

Echo-Service

- Im folgenden erstellen wir ein Programm, das eine Verbindung zum *ECHO-Service* auf Port 2007 herstellt. Das Programm liest so lange die Eingaben des Anwenders und sendet sie an den Server, bis End-of-File gelesen wird. Ein EOF kann erzeugt werden durch Eingabe von Crtl-Z Return (Windows) oder Crtl-D (UNIX). Der Server liest die Daten zeilenweise und sendet sie unverändert an unser Programm zurück, von dem sie auf dem Bildschirm ausgegeben werden. Um Lese- und Schreibzugriffe zu entkoppeln, verwendet das Programm einen separaten Thread, der die eingehenden Daten liest und auf dem Bildschirm ausgibt. Dieser läuft unabhängig vom Vordergrund-Thread, in dem die Benutzereingaben abgefragt und an den Server gesendet werden.

Beispiel: `EchoClient.java`

- EchoClient.java
- Es wird zunächst ein Socket zu dem als Argument angegebenen Host geöffnet. Das Programm beschafft dann Ein- und Ausgabestreams zum Senden und Empfangen von Daten. Der Aufruf von **`setSoTimeout`** gibt die maximale Wartezeit bei einem lesenden Zugriff auf den Socket an. Wenn bei einem **`read`** auf den **`InputStream`** nach Ablauf dieser Zeit noch keine Daten empfangen wurden, terminiert die Methode mit einer **`InterruptedIOException`**. Nun erzeugt das Programm den Lesethread und übergibt ihm den Eingabestream. In der nun folgenden Schleife (Zeile 27) werden so lange Eingabezeilen gelesen und an den Server gesendet, bis End-of-File gelesen wird.

Beispiel: Echo-Client (2)

- Die Klasse **`OutputThread`** implementiert den Thread zum Lesen und Ausgeben der Daten. Da die Methode **`stop`** der Klasse **`Thread`** im JDK 1.2 als deprecated markiert wurde, müssen wir mit Hilfe der Variable **`stoprequested`** etwas mehr Aufwand treiben, um den Thread beenden zu können. **`stoprequested`** steht normalerweise auf **`false`** und wird beim Beenden des Programms durch Aufruf von **`requestStop`** auf **`true`** gesetzt. In der Hauptschleife des Threads wird diese Variable periodisch abgefragt, um die Schleife bei Bedarf abbrechen zu können (Zeile 81).

Beispiel: Echo-Client (3)

- Problematisch bei dieser Technik ist lediglich, dass der Aufruf von **`read`** normalerweise so lange blockiert, bis weitere Zeichen verfügbar sind. Steht das Programm also in Zeile 83, so hat ein Aufruf **`requestStop`** zunächst keine Wirkung. Da das Hauptprogramm in Zeile 48 die Streams und den Socket schließt, würde es zu einer **`SocketException`** kommen. Unser Programm verhindert das durch den Aufruf von **`setSoTimeout`** in Zeile 19. Dadurch wird ein Aufruf von **`read`** nach spätestens 300 ms. mit einer **`InterruptedIOException`** beendet. Diese Ausnahme wird in Zeile 87 abgefangen, um anschließend vor dem nächsten Schleifendurchlauf die Variable **`stoprequested`** mit der **`synchronized`** Methode **`stopRequested`** erneut abzufragen.

Timeout

- Das Warten auf ein einkommendes Datagrampaket (UDP) sowie auf das Zustandekommen einer Verbindung und das Warten auf Daten (TCP) kann durch ein Timeout begrenzt werden:
- **`setSoTimeout( millisecs )`**
- beendet die Blockade nach der angegebenen Anzahl von Millisekunden, wenn in dieser Zeit kein Datenpaket ankommt oder keine Verbindung zustande kommt.
- Damit kann ein generelles Blockieren unterbunden werden.
- Wird für **`millisecs`** der Wert 0 angegeben, so soll wirklich "endlos" gewartet werden.

Beispiel: simpler Echo-Server

- SimpleEchoServer.java
- Wir wollen uns die Konstruktion von Servern an einem Beispiel ansehen. Dazu soll ein einfacher ECHO-Server geschrieben werden, der auf Port 2007 auf Verbindungswünsche wartet. Alle eingehenden Daten sollen unverändert an den Client zurückgeschickt werden. Zur Kontrolle sollen sie ebenfalls auf die Konsole ausgegeben werden:

Beispiel: simpler Echo-Server (2)

- Wird der Server gestartet, kann mit dem EchoClient auf den Server zugegriffen werden:

```
java EchoClient localhost
```