# Socket-Programmierung unter Java

## 1 - Grundlagen: Datenströme

# Datenströme

JAVA unterscheidet Streams und Reader/Writer

- Zur Dateneingabe: InputStream oder Reader
- Zur Datenausgabe: OutputStream oder Writer
- Verwende Reader und Writer für Textdaten, z.B. HTTP, Chat
- Verwende Streams für Binärdaten, z.B. Datei kopieren

# Binärdatenströme (1)

- FileInputStream

  Lesen aus einer Datei
- PushbackInputStream

  erlaubt es, bereits gelesene Daten wieder “zurückzuschieben, um sie erneut Lesen zu können
- DataInputStream

  machtvolle Klasse um alle Arten von Binärdaten
- ObjectInputStream

  erlaubt es Java-Objekte aus einem Strom zu lesen (? Serialization)
- ...

# Binärdatenströme (2)

- PrintStream

  erlaubt Schreiben von Text-Repräsentation von Standard-Datentypen
- FileOutputStream

  Schreibt in eine Datei
- PipedOutputStream

  Erlaubt es direkt in einen anderen InputStream zu schreiben
- ObjectOutputStream

  kann Java-Objekte in Strom schreiben

# Binärdatenströme (3)

- DataInputStream
  - `boolean readBoolean()`
  - `byte readByte()`
  - `char readChar()`
  - `double readDouble()`
  - `float readDouble()`
  - `int readInt()`
  - `String readUTF()`
  - `...`

# Binärdatenströme (4)

- DataOutputStream
  - `writeBoolean(boolean v)`
  - `writeByte(int b)`
  - `writeUTF(String s)`

    Unicode - 2 Byte pro Zeichen
  - `writeBytes(String s)`

    ASCII+ - 1 Byte pro Zeichen
  - `char readChar()`
  - `double readDouble()`
  - ...

# Textdatenströme (1)

- BufferedReader

  mit Zwischenpuffer zum effiziente, zeilenweise Einlesen
- CharArrayReader

  Lesen einzelner UniCode-Zeichen
- FileReader

  Lesen aus einer Datei
- InputStreamReader

  Bildet einen InputStream auf einen Reader ab
- StringReader

  Bildet einen String auf einen Reader ab

# Textdatenströme (2)

- PrintWriter

  erlaubt Schreiben von Text-Repräsentation von Standard-Datentypen
- FileWriter

  Schreibt in eine Datei
- OutputStreamWriter

  Bildet einen Writer auf einen OutputStream ab
- StringWriter

  Schreibt Daten in einen String
- ...

# Socket-Programmierung unter Java

## 2 - Strom- und Datagrammsockets

# Was sind Sockets?

- Sockets sind Endpunkte einer Kommunikation zwischen zwei Endpunkten (Anwendungen auf Rechnern).
- Sockets gehören zu einer Anwendung.
- Sockets werden an eine Portnummer gebunden.
- Sockets können an alle oder bestimmte IP-Adressen eines Rechners gebunden werden.
- Sockets gibt es für TCP und UDP-Kommunikation.

Unter Unix/Linux findet sich in `/etc/services` eine Auflistung von registrierten Portnummern.

# Was sind Sockets?

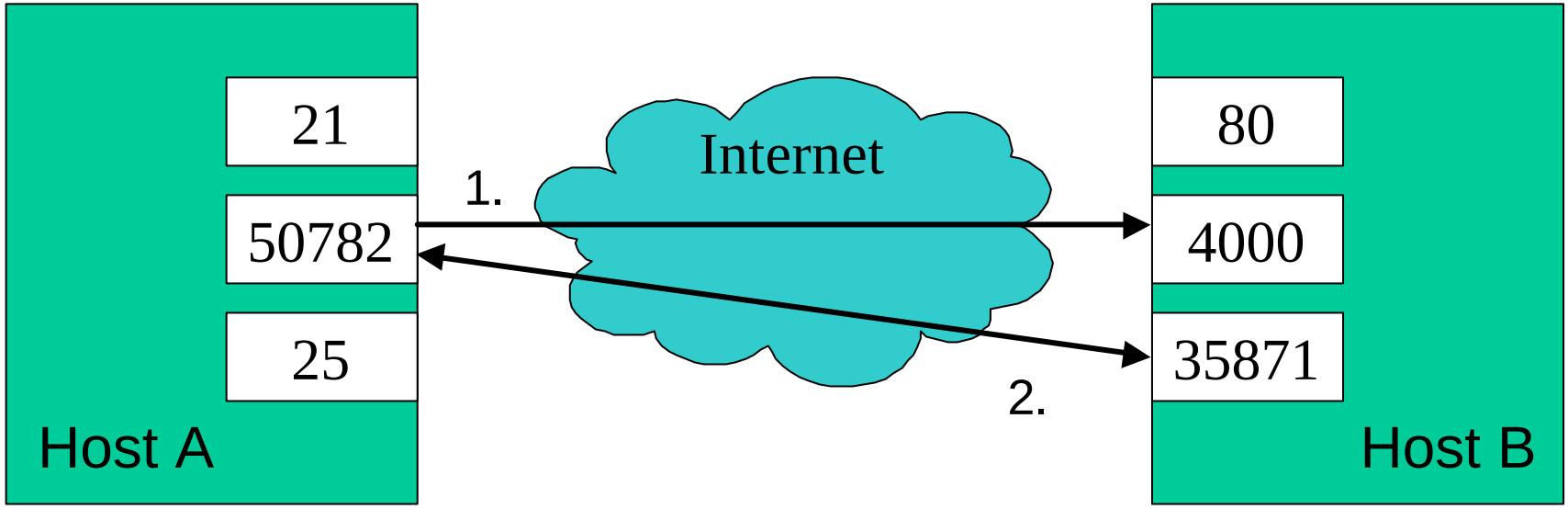
- vgl. mit realen Adressen:
  - IP-Adresse ? Ort + Straße + Hausnummer
  - Port ? Klingelknopf
  - Anwendung ? Wohnung
- bekannte Portnummern:
  - ftp: 21/tcp
  - ssh: 22/tcp, 22/udp
  - telnet: 23/tcp
  - http: 80/tcp
  - dns: 53/tcp, 53/udp

# TCP / UDP

- Zwei Internet-Transportprotokolle: TCP und UDP
- UDP
  - Übertragung von Paketen (Datagrammen)
  - keine Garantie für Reihenfolge oder Vollständigkeit der Pakete beim Empfänger
- TCP
  - Übertragung eines Bytestroms (intern Pakete)
  - Reihenfolge und Vollständigkeit garantiert

# grober Ablauf (TCP)

1. A erstellt einen lokalen Socket auf beliebigen Port und stellt eine Verbindung zu B auf Port 4000 her.
2. B nimmt die Verbindung entgegen, erzeugt einen Socket mit bel. Portnummer und übergibt Verbindung.



# java.net - InetAddress

- Klasse `java.net.InetAddress`

  Zum Zugriff auf IP (v4) Adressen unter JAVA

  (IPv6 angeblich ab Herbst 2001)

```
String host = “willy”;
InetAddress server = null;
try {
  server = InetAddress.getByName(host);
} catch (UnknownHostException uhe) {
  System.out.println(“Ein Server namens “+host+
                     “ ist unbekannt.”);
}
```

# java.net - Exceptions

- Klasse `BindException`

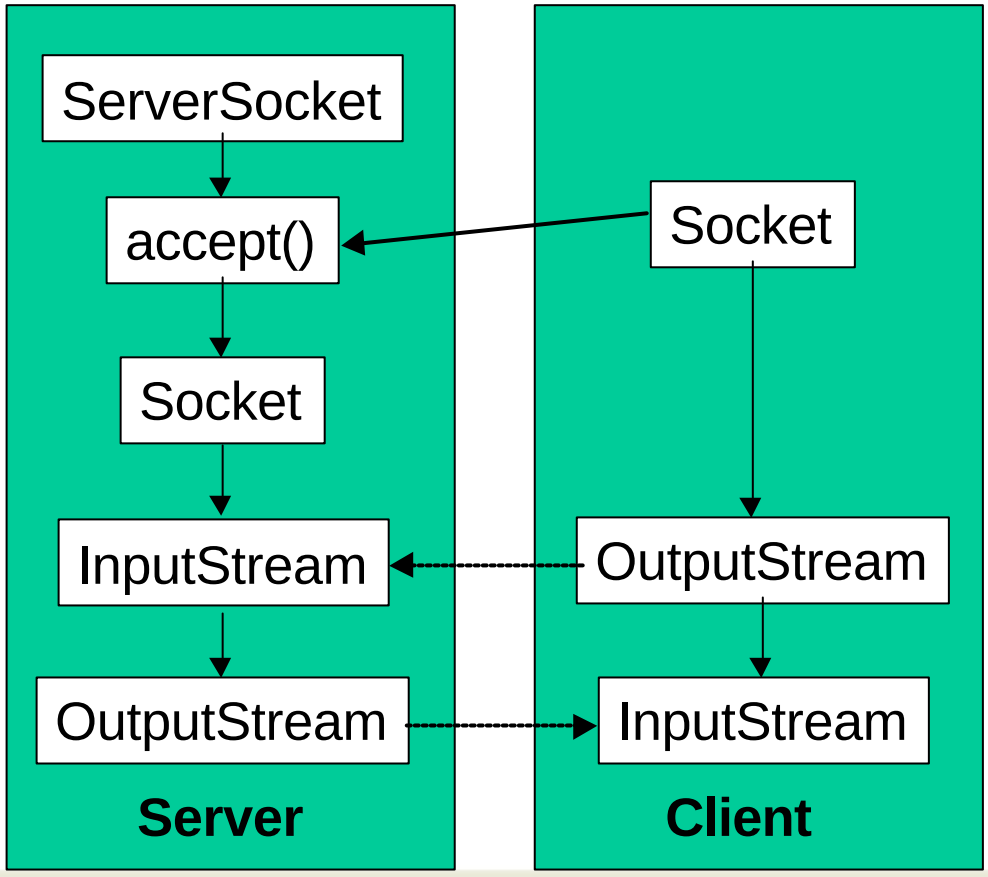
  Socket konnte nicht reserviert werden
- Klasse `UnknownHostException`

  IP-Adresse zu Hostnamen konnte nicht ermittelt werden
- Klasse `NoRouteToHostException`

  Die angegebene IP-Adresse kann nicht erreicht werden
- Klasse `SocketException`

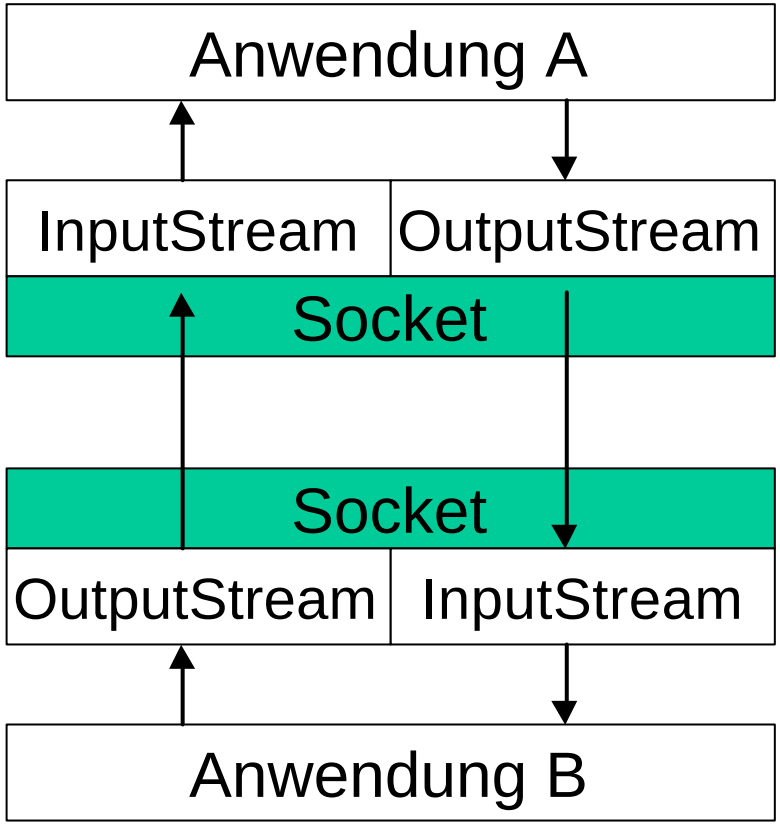
  Ein Fehler im unterliegenden Protokoll (z.B. TCP) ist aufgetreten.

# Verwendung von Stream-Sockets




- ServerSocket wird an Port gebunden
- accept() wartet auf eingehende Verbindungen - liefert Socket
- Socket liefert Input- und OutputStream

# Verwendung von Stream-Sockets (2)




- Aus Sicht des JAVA-Programmes wird die gesamte Kommunikation über Stream-Objekte abgewickelt.
- Socket-Objekt bietet Methoden zum Zugriff auf Stream-Objekte
  `getInputStream()`
  `getOutputStream()`

# Stream-Sockets (Server)

(Achtung: In den Beispielen werden keine Exceptions abgefangen)

Warte auf Port 4000 auf eingehende Verbindungen

```
ServerSocket incom = new ServerSocket(4000);
Socket con = incom.accept();
```

Initialisiere Datenströme

```
InputStream in = con.getInputStream();
OutputStream out = con.getOutputStream();
```

# Stream-Sockets (Client)

Baue Verbindung zu Server willy auf Port 4000 auf.

```
try {
  Socket con = new Socket(“willy”, 4000);
  InputStream in = con.getInputStream();
  OutputStream out = con.getOutputStream();
} catch (UnknownHostException uhe) {
  System.err.println(“No such host”);
} catch (IOException ioe) {
  System.err.println(“Could not connect.”);
}
```

# Stream-Sockets - Nützliches

- `getLocalAddress()` und `getLocalPort()` liefern IP-Adresse und Portnummer des lokalen Sockets
- `getInetAddress()` und `getPort()` liefern dieselben Daten über den entfernten Socket
- TCP-Sockets zum Verarbeiten von Text lassen sich mittels `telnet` testen (`telnet <server> <port>`)

# Text via StreamSockets

Text-Kommunikation mittels Socket

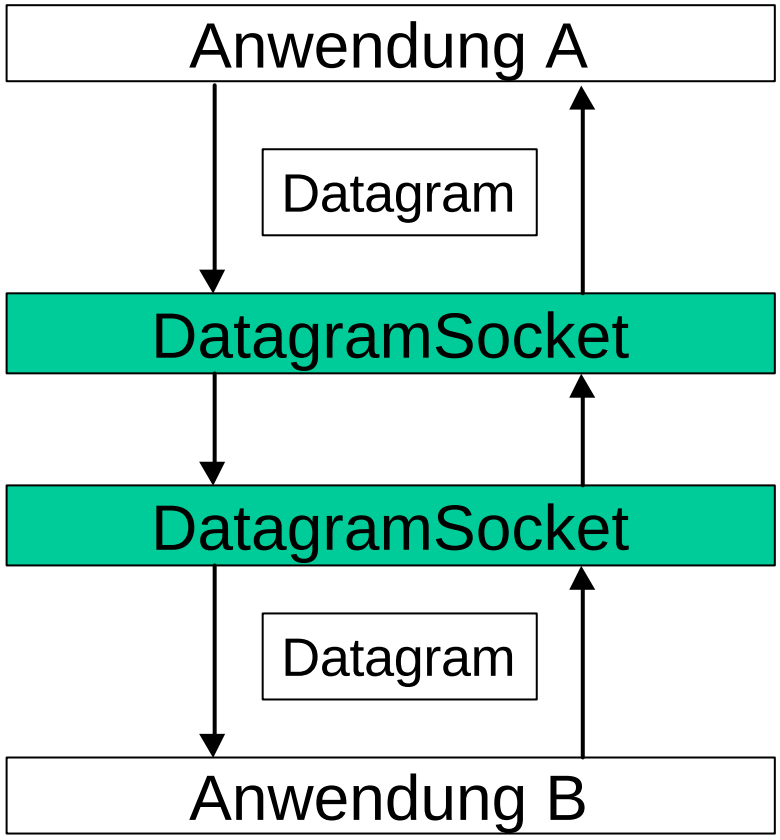
```
BufferedReader in = new BufferedReader(new
  InputStreamReader(sock.getInputStream());
PrintWriter out = new PrintWriter(new
  OutputStreamWriter(sock.getOutputStream());
while (true) {
  String line = in.readLine();
  out.println("Du schriebst: "+line);
}
```

# Nützliches: Textzerlegung

Die Klasse java.util.StringTokenizer erlaubt das Zerlegen eines Strings anhand beliebiger Trennzeichen (z.B. Space, um einzelne Wörter zu erhalten).

```
String line = “Dies ist nur ein Beispiel”;
StringTokenizer tok = new
  StringTokenizer(line, “ “);
while (tok.hasMoreTokens()) {
  System.out.println(“--> “+tok.nextToken());
}
```

# Verwendung von Datagram-Sockets




- DatagramSockets haben Methoden zum Senden und Empfangen von Datagrammen
  `write(Datagram pack)`
  `receive(Datagram pack)`
- Datagram-Objekt enthält den zu schreibenden oder zu füllenden Bytebuffer
- Bei receive() muß vorher Platz für Daten reserviert werden

# DatagramSocket (Sender)

Erzeuge lokalen Socket auf Port 4000

```
DatagramSocket sock = new DatagramSocket(4000);
```

Sende Daten an willy Port 5000

```
InetAddress dest =
  InetAddress.getByName("willy");
byte[] myData = new byte[4096];
DatagramPacket dp = new DatagramPacket(myData,
  myData.length, dest, 5000);
sock.send(dp);
```

# DatagramSocket (Empfänger)

Erzeuge lokalen Socket auf Port 5000

```
DatagramSocket sock = new DatagramSocket(4000);
```

Erwarte Daten

```
byte[] myData = new byte[4096];
DatagramPacket dp = new DatagramPacket(myData,
  myData.length);
sock.receive(dp);  // blockiert
```

Daten stehen anschließend in `myData`, die Anzahl der tatsächlich empfangenen Bytes in `dp.getLength()`.

# DatagramSockets - Nützliches

- Bei empfangenen DatagramPackets kann man mit `getAddress()` und `getPort()` die IP-Adresse und Portnummer des Senders ermitteln.
- Variante `MulticastSocket`:
  - Erlaubt Gruppenkommunikation
  - Alle Empfänger lauschen auf speziellen IP-Adressen
  - Wenn einer an diese Adresse sendet, empfangen es alle
  - Mehr in "*Rechnernetze 2*"

# Übungsaufgaben

- Schreibe einen TCP Echo-Server (sendet die Daten, die er empfängt an der Absender zurück) für Texteingaben und teste diesen mit Telnet.
- Schreibe ein Server, der via UDP mehrere Bytes auf einmal empfängt, jedes einzeln bearbeitet (z.B. durch 2 teilt) und das Ergebnis wieder zurücksendet.