

Skript Informatik B

Objektorientierte Programmierung in Java

Sommersemester 2011

- Teil 9 -

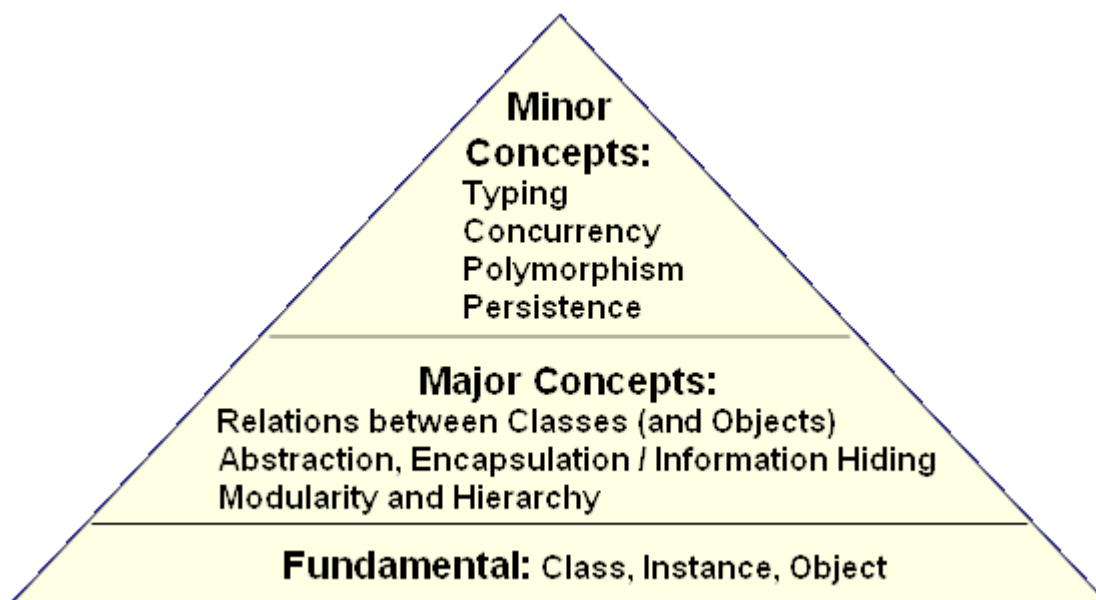
Inhalt

- 0 Einleitung
- 1 Grundlegende objektorientierte Konzepte (Fundamental Concepts)
- 2 Grundlagen der Software-Entwicklung
- 3 Wichtige objektorientierte Konzepte (Major Concepts)
- 4 Fehlerbehandlung
- 5 Generizität (Generics)
- 6 Polymorphie / Polymorphismus
- 7 Klassenbibliotheken (Java Collection Framework)
- 8 Persistenz
- 9 Nebenläufigkeit
- 10 Grafische Benutzeroberflächen (GUI)
- 11 Netzwerkprogrammierung

Kapitel 11:

Netzwerkprogrammierung

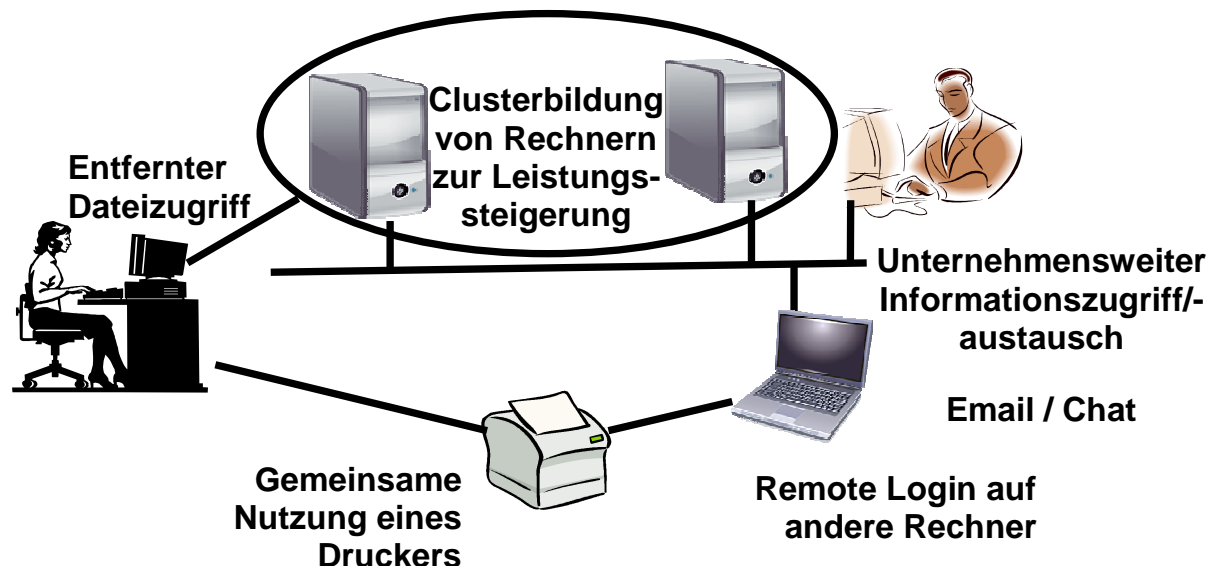
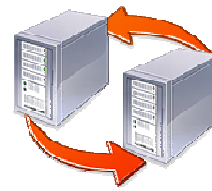
- 11.1 Netzwerke Grundlagen
- 11.2 Netzwerkprotokolle
- 11.3 Adressierung
- 11.4 Sockets
- 11.5 Client/Server
- 11.6 Implementierung in Java
- 11.7 Höhere Kommunikation



[Boo94]

11.1 Netzwerke Grundlagen

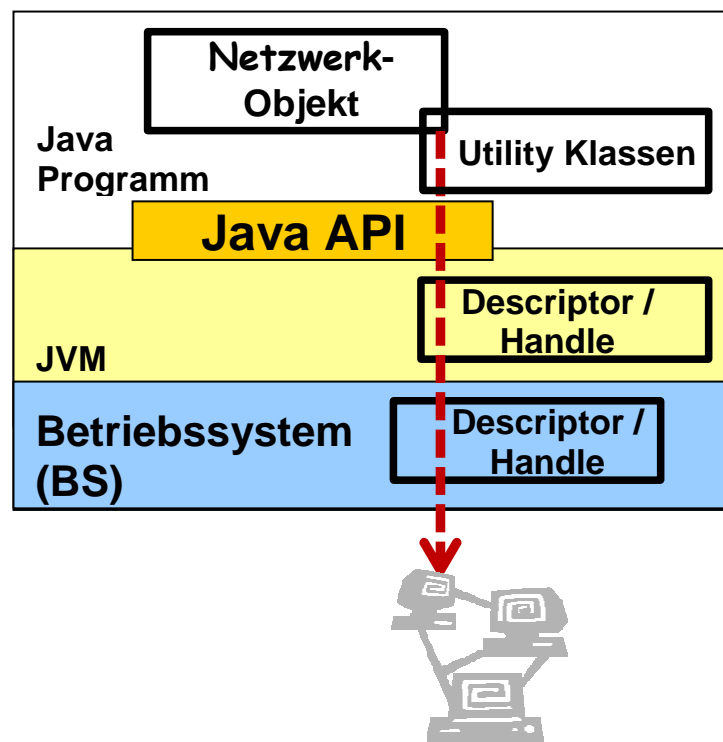
- Netzwerk: Verbindungen zwischen Geräten, um Daten auszutauschen
- Verschiedene Netzwerktopologien: Die Geräte im Netzwerk können nach verschiedenen Mustern „verkabelt“ sein (z.B. Bus, Ring, Stern).
- Einsatzbeispiele:

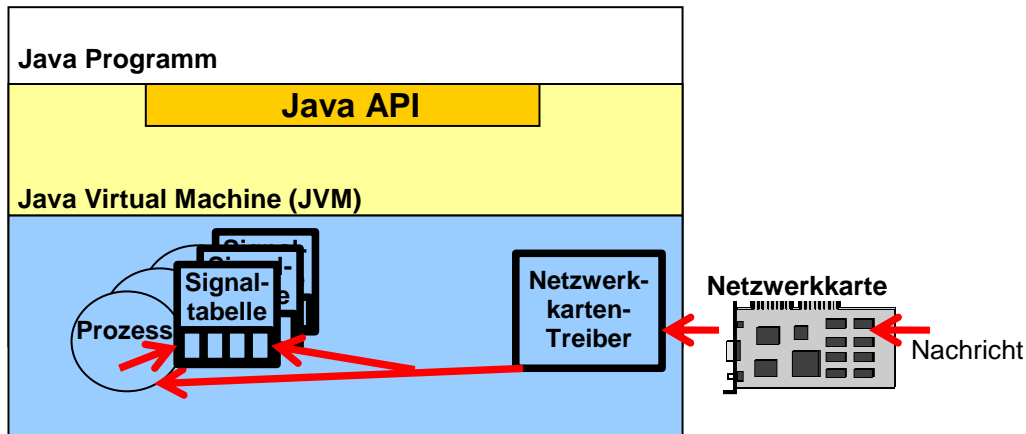


Java Sicht auf das Netzwerk:

Java unterstützt die Netzwerkprogrammierung mit eigenen Bibliotheken.

- Package `java.net`: Klassenbibliothek zum Verbindungsaufbau und zur Kommunikation über Netze.
- Plattformunabhängigkeit: Ein Netzwerkprogramm in Java muss keine Betriebssystem- und Hardware-Details kennen. Für jede Plattform sieht das Programm aus Entwicklersicht gleich aus.





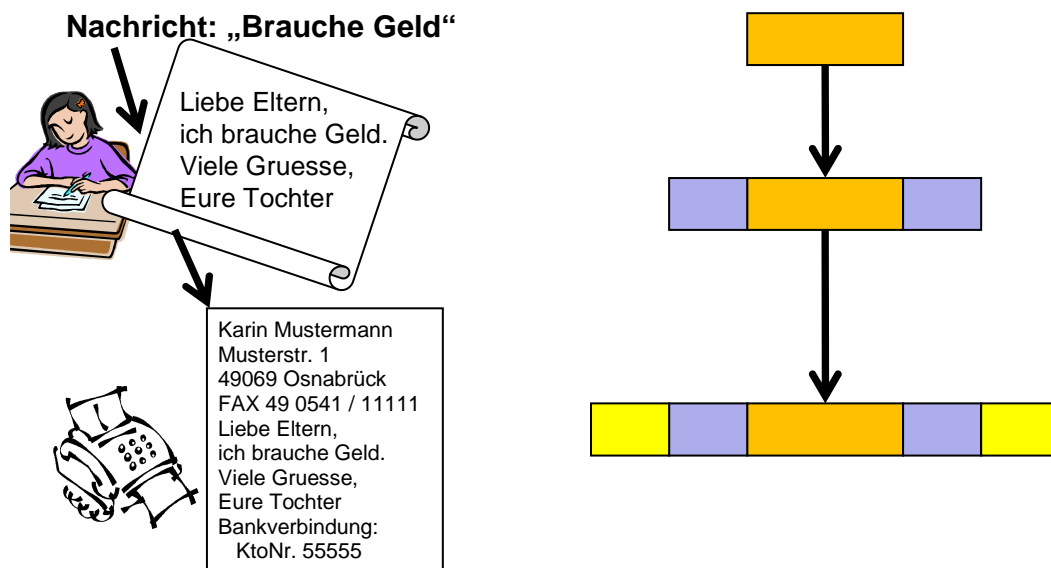
Kommt über eine Netzwerkkarte eine Information an, so wird sie zunächst von der Treibersoftware im Betriebssystem aufgefangen. Die Nachricht wird auf Basis der Daten der Nachricht einem Prozess zugeordnet. In der dem Prozess zugeordneten Signaltabelle setzt die Treibersoftware ein Signal. Der Prozess sieht das Signal und nimmt die gesamte Nachricht von der Treibersoftware entgegen. Anschließend ist der Prozess für die Weiterverarbeitung der Nachricht zuständig.

Zwei Fragestellungen müssen wir uns auf allen Ebenen (also auch auf der Programmierenebene) stellen:

- 1) Wie wird die Nachricht richtig verstanden?
- 2) Wie findet die Nachricht den richtigen Adressaten?

11.2 Netzwerkprotokolle

- Zum Informationsaustausch müssen sich die Partner auf ein gemeinsames Protokoll einigen.
- Protokoll (engl. Protocol): Ein Protokoll ist die Menge aller Regeln für den Verbindungsaufbau, Datenaustausch und Verbindungsabbau.
- Die Regeln legen das Format, den Inhalt, die Bedeutung und die Reihenfolge der gesendeten Nachrichten fest.
- Vergleichbar mit der zwischenmenschlichen Kommunikation.



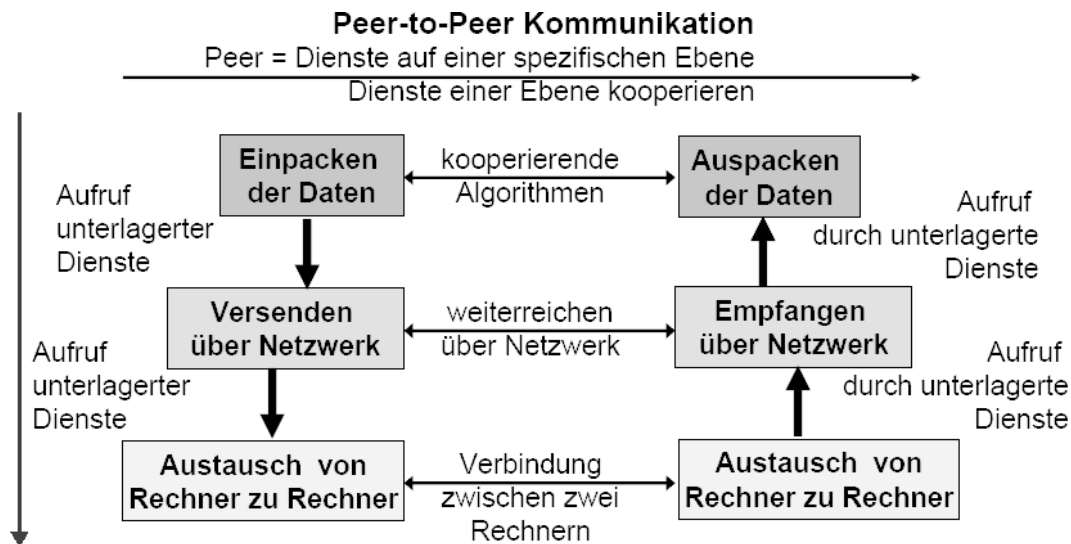
Eine Nachricht wird über mehrere Schichten geleitet.

Sie erhält so in jeder Schicht einen weiteren Kopf und Fuß. Auf jeder Ebene ist die Einhaltung bestimmter Konventionen notwendig.

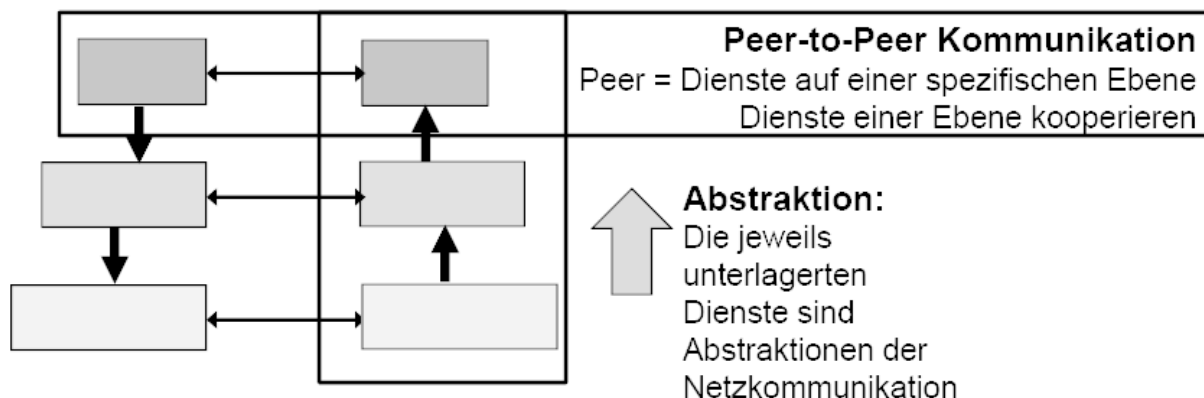
Beispiel: Ein Brief erhält in der obersten Ebene die Anrede („Liebe Eltern“) und einen Gruß („Viele Grüße“). Dieser Kopf und Fuß ist eine Konvention, die grundsätzlich nichts mit der übermittelten Information zu tun hat. In der nächsten Ebene wird der Brief weiter verpackt. Während Anrede und Gruß für die Eltern bestimmt war, muss nun ein Kopf und Fuß hinzu, damit das Faxgerät den Brief richtig ausliefern kann.

Im Nachrichtenaustausch in einem Netzwerk haben wir viele solche Schichten. Jede Schicht übernimmt eine andere Aufgabe. Jede Schicht abstrahiert so von den darunterliegenden Details.

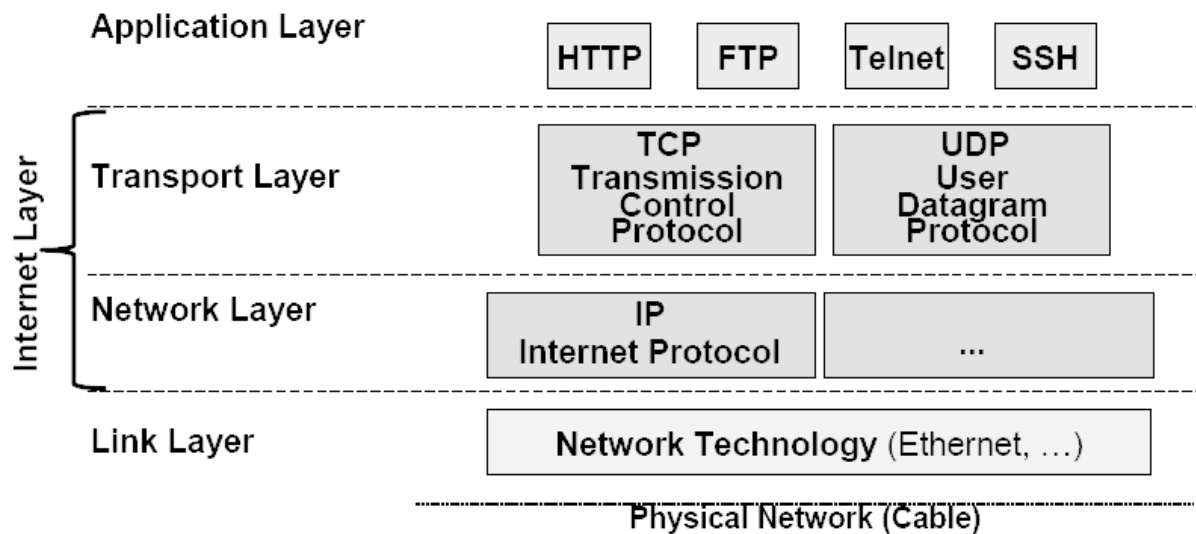
In einer bestimmten Schicht erscheint die Netzwerkkommunikation wie eine Kommunikation mit einem Partner auf der gleichen (Abstraktions-)Ebene (Peer-to-Peer).



Zwei Sichten: horizontal und vertikal



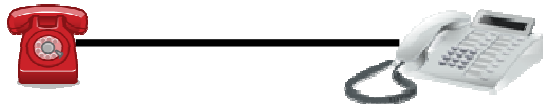
- Netzwerke werden durch geschichtete Protokolle (*Protocol Stack*) beschrieben.
 ⇒ Ein Modell der Netzwerkkommunikation definiert den Aufbau eines Protocol Stacks.
- Es gibt viele Modelle und noch mehr Protokolle.
- Bekanntes Referenzmodell:
 Open System Interconnection Reference Model (OSI) of the International Standards Organization (ISO) (definiert Ebenen, deren Dienste und deren Interaktionen)
 ISO/OSI ist ein Referenzmodell, da es keine Implementierung vorgibt.
- Konkretes Modell: TCP/IP (entwickelt vom US Department of Defense)
 TCP/IP ist parallel zum ISO/OSI-Referenzmodell entstanden.

Beispiel: TCP/IP (“das Netzwerkmodell des Internets”)**Schichten des TCP/IP Protocol Stacks:**

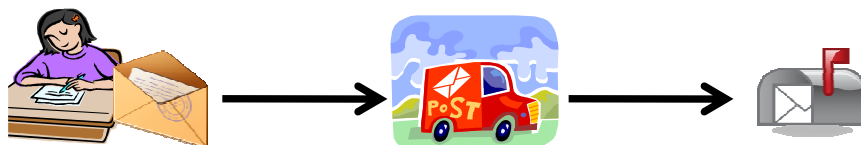
- **Application Layer (Anwendungsschicht):** Diese oberste Schicht umfasst alle Anwendungen, d.h. was der Anwender letztlich sieht (z.B. SSH, mail, Dateimanager, usw.).
- **Transport Layer (Transportschicht):** Sie sorgt für eine zuverlässige Verbindung (Adressierung, Ansprechen von Datenendgeräten, Organisation der Auslieferung).
- **Network Layer (Netzwerkschicht, Vermittlungsschicht):** In dieser Schicht werden Aufgaben wie Routing (Wegefindung), Verbindungsaufbau, -abbau, Zuordnung von Netzwerkadressen erledigt.
Bemerkung: Der Weg zweier Pakete durch das Netzwerk mit gleichem Bestimmungsort muss nicht gleich sein.
- **Link Layer (Sicherungsschicht und physikalische Schicht):** Diese Schicht enthält alle Fähigkeiten zum physikalischen Ansprechen der Kabel; hier findet die Bitübertragung statt.

Protokolle können unter anderem folgendermaßen unterschieden werden:

- 1) Verbindungsorientierte Protokolle (connection oriented), z.B. TCP



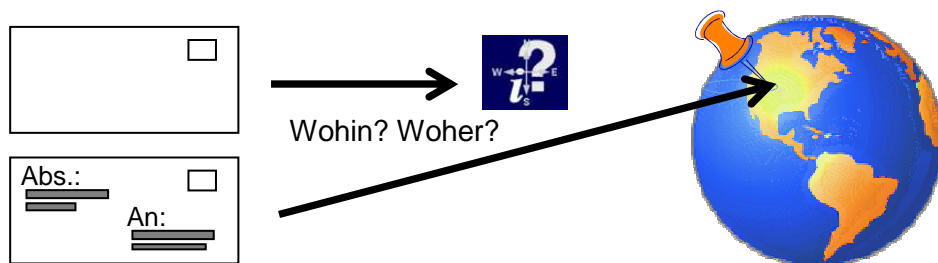
- 2) Verbindungslose Protokolle (connectionless), z.B. IP, UDP



Wichtige, konkrete Protokolle, die innerhalb des TCP/IP Protocol Stacks zum Einsatz kommen:

- TCP (Transmission Control Protocol)
 - Verbindungsorientiertes Protokoll: virtueller Kanal zwischen zwei Endpunkten einer Netzwerkverbindung.
 - Sichere und fehlerfreie Punkt-zu-Punkt-Verbindung.
 - Automatische Fehlerkorrektur durch automatische Neuansforderung im Fall von Übertragungsfehlern.
- UDP (User Datagram Protocol)
 - Verbindungsloses, unsicheres Protokoll.
 - Die Anwendung muss selbst dafür sorgen, dass die einzelnen Datagram-Pakete überhaupt und in der richtigen Reihenfolge beim Empfänger ankommen.
 - Größere Geschwindigkeit als mit TCP (da weniger überprüft wird).

Transport von Daten per Post:



Der Transport von Daten über das Netzwerk benötigt analog zur Post:

- die Adresse des Geräts, das die Daten versendet hat (Sender),
- die Adresse des Geräts, das die Daten bekommen soll (Ziel-/Empfängeradresse).

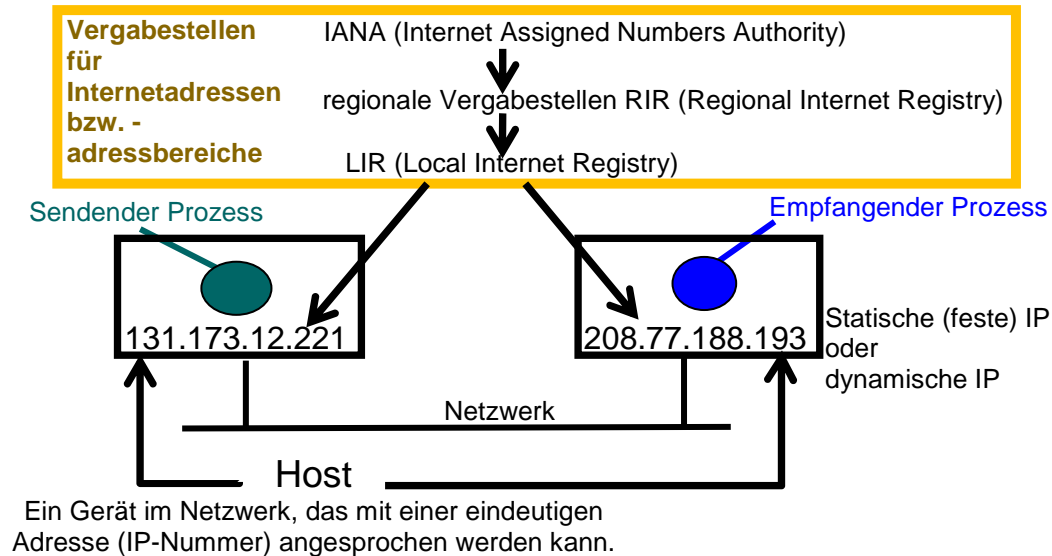
Adressformat in TCP/IP-Netzen auf der IP-Ebene:

- Heute: IPv4 (32-Bit IP-Adresse) \Rightarrow pro km² Erdoberfläche: 8,4 Adressen
- In Zukunft: IPv6 (128 Bit) \Rightarrow pro mm² Erdoberfläche >650 Billionen Adressen

11.3 Adressierung

IP-Adressvergabe:

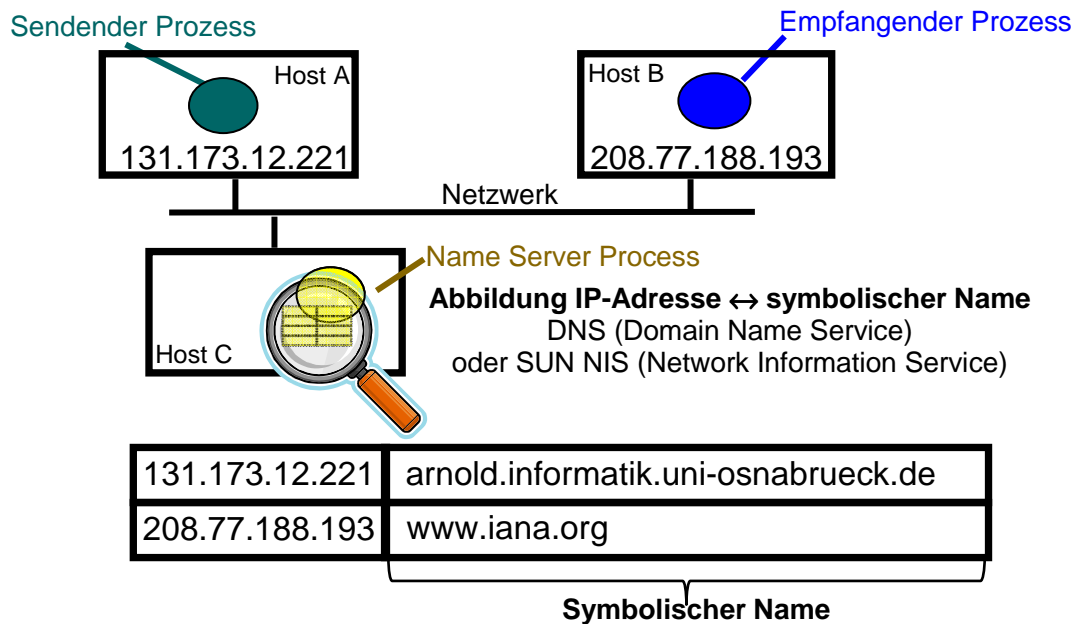
IP-Adressen werden von dafür autorisierten Institutionen zentral vergeben:



Localhost: Pseudo-Adresse für den eigenen Host (IP-Adresse: 127.0.0.1).

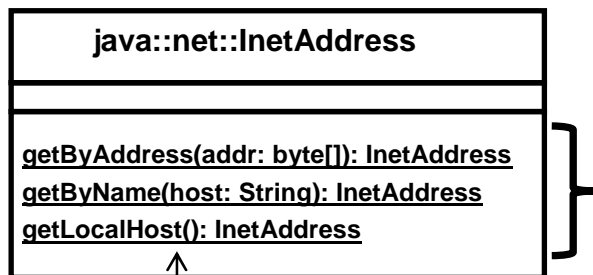
Namensdienst:

IP-Adressen werden mit Hilfe eines Namensdiensts (Name Server) auf symbolische Namen abgebildet. Der Namensdienst ist ein Prozess, der im Netzwerk läuft.



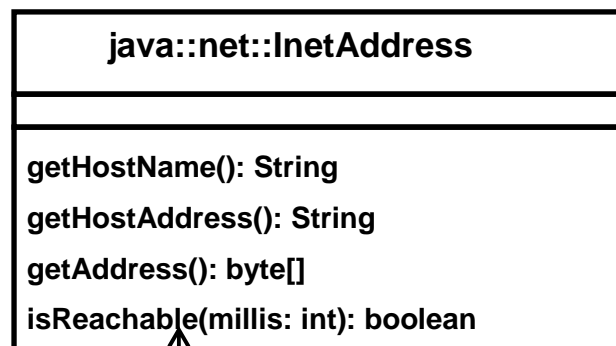
Hilfsmittel in Java zur Adressierung von Hosts im Netz:

- Package `java.net`
- Java-Repräsentation einer Adresse mit `InetAddress`-Objekten.



Liefert die lokale Internetadresse

- Instanzen für IP-Adressobjekte erhält man nur durch Anfrage über statische Methoden.
Es gibt keine Konstruktoren.
- `InetAddress` verwaltet IP-Adresse (IPv4/IPv6) und Hostname.



Verwaltet IP-Adresse (IPv4/IPv6) und Hostname (Abfragen beider Bestandteile möglich)

Überprüft, ob das Gerät, das durch das `InetAddress`-Objekt repräsentiert ist, erreichbar ist:

- Aussenden eines Requests (dt. Anfrage)
- Erfolgt innerhalb des angegebenen Zeitraumes keine Antwort, wird vermutet, dass das Gerät nicht erreichbar ist.

Java-Beispiel zur Adressierung von Hosts im Internet:

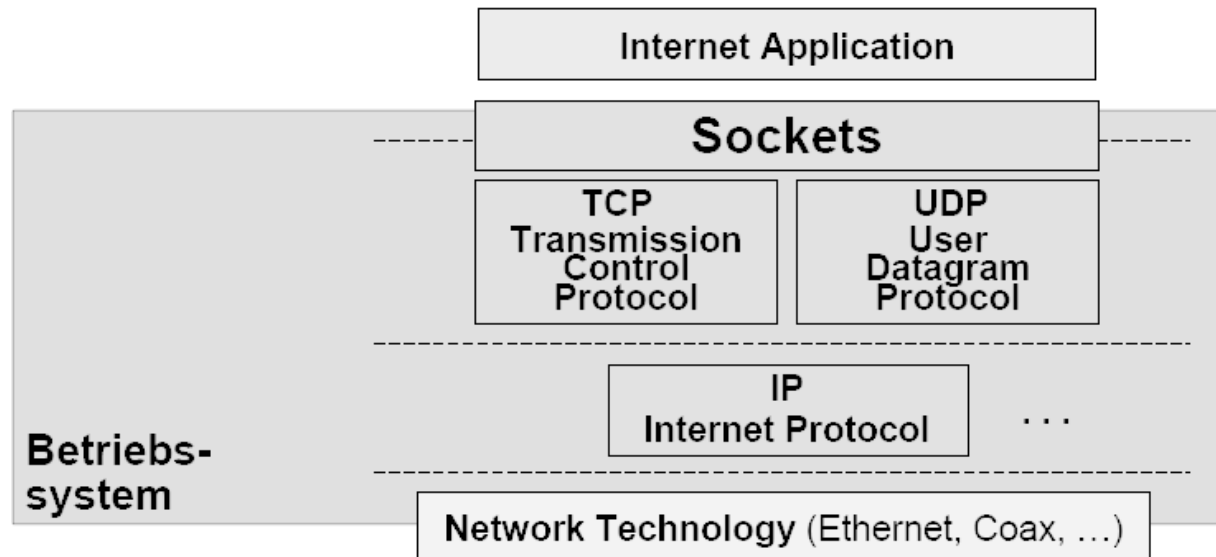
```

import java.net.*;
public class InetAddress {
    public static void main(String args[]) {
        try {
            InetAddress iaddr =
                InetAddress.getByHost("www.uni-osnabrueck.de");
            System.out.println(iaddr.getHostName() + "; "
                               + iaddr.getHostAddress());
        }
        catch (UnknownHostException e) {
            System.err.println("no such host");
        }
    }
}
  
```

UnknownHostException!

11.4 Sockets

Socket(s): Ein Socket ist eine Programmierschnittstelle zur Kommunikation zweier Anwendungen über das Netzwerk (auf der Basis von TCP/IP).



Sockets ermöglichen Programmen (bzw. den zur Programmausführungszeit zugehörigen Prozessen) den Zugriff auf das Netzwerk

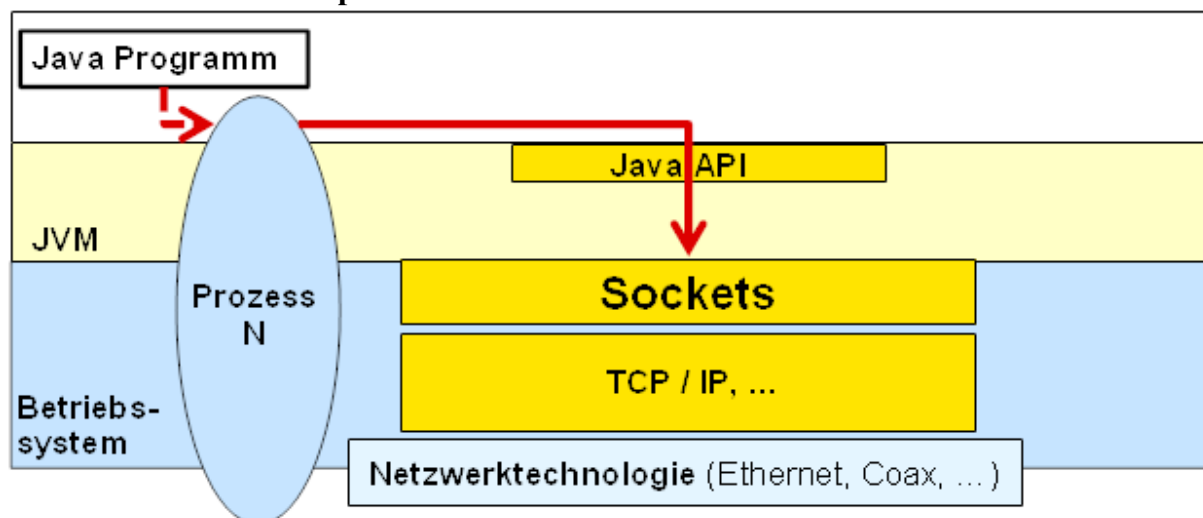
Der Zugriff und die Datenübertragung ähnelt (absichtlich) dem Zugriff auf eine Datei:

- Verbindung aufbauen,
- Daten lesen und/oder schreiben,
- Verbindungsabbau.

Sockets bieten verschiedene Kommunikationsformen:

- zuverlässiger verbindungsorientierter Byte-Stream,
- zuverlässiger, verbindungsorientierter Paketstrom (Paket-Stream),
- unsichere Paketübermittlung (Paket Transmission).

Sockets in der Java Perspektive:



Ein Prozess besteht aus Programm-Code und Code aus eingebundenen Bibliotheken des Betriebssystems und der Laufzeitumgebung. Obwohl der Prozess im Betriebssystem verwaltet wird, umfasst er damit aus Code-Sicht Programmzeilen aus den verschiedenen Schichten. In der Abbildung oben: Der Prozess „umspannt“ mehrere Abstraktionsschichten. Die Zuordnung eines Prozess zur jeweiligen Schicht ergibt sich danach, welcher Code-Anteil sich gerade in der Ausführung befindet.

Kommunikation und ihre Bestandteile:

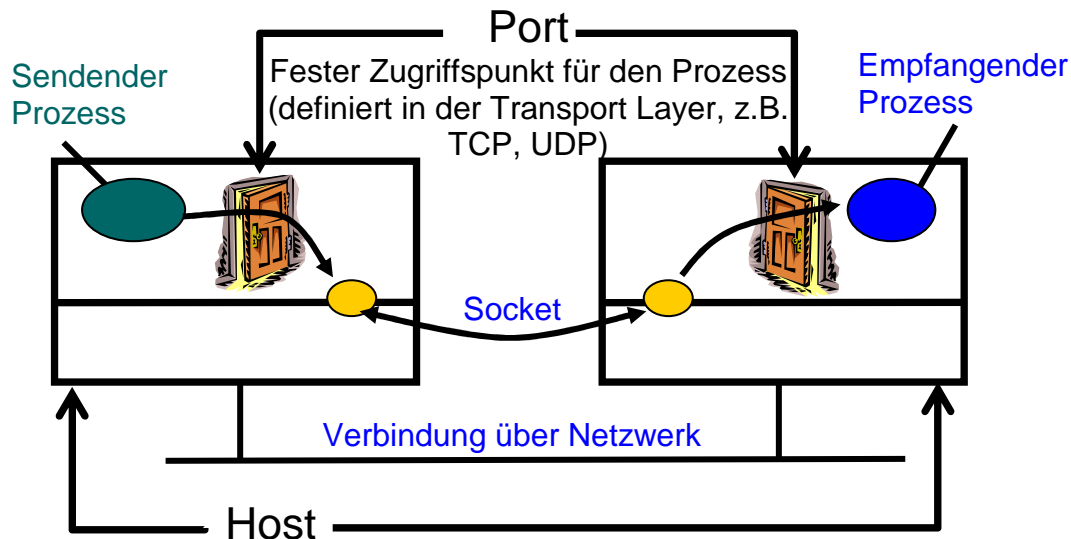


Abbildung nach [Ta92] A.S. Tanenbaum. Modern Operating Systems, Prentice-Hall, 1992

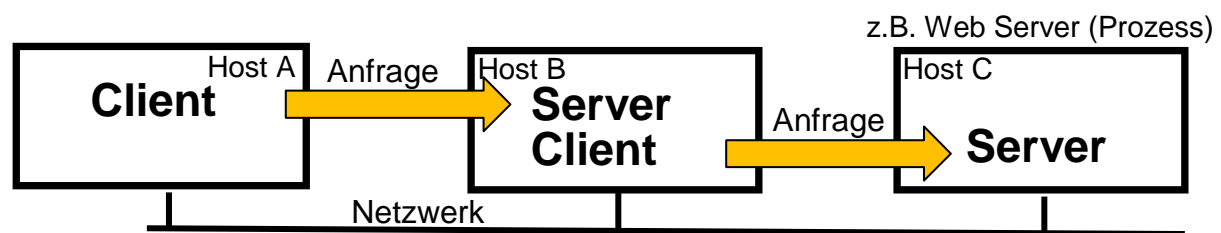
- Socket: Schnittstelle zur Bestimmung des Hosts und des Prozesses.
Ein Socket umfasst die für die Kommunikation benötigten Ports und Host-Adressen.
- Port: Bestimmung der Anwendung (bzw. des Prozesses) auf dem Host.
- Beispiel: FTP-Prozesse (zur Datenübertragung) haben den Port 21, der Http-Server-Prozess wartet auf Port 80 auf Anfragen (z.B. www.uni-osnabrueck.de:80)
- Vergabe von Port-Nummern (Nummernkreise):

Die Konvention zur Verwendung der Nummern als Port-Nummern zeigt die folgende Tabelle (Nummernkreise):

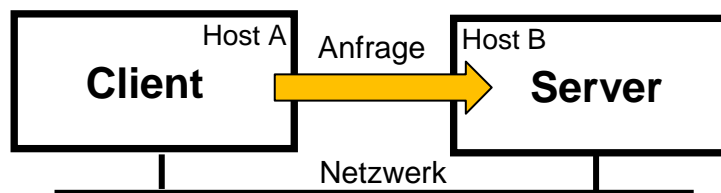
1 bis 1023	Systemprozesse oder systemnahe Prozesse (Superuser-Rechte)
1024 bis 5000	Werden von der Portverwaltung vergeben
5001 bis 65535	Vom Anwender frei verwendbar

11.5 Client/Server

- Bekanntes Kommunikationsprinzip: Client/Server
- Definiert eine Rollenverteilung.
- Client: Eine Anwendung, die über das Netzwerk kommunizieren möchte.
- Server: Eine Anwendung, die über das Netzwerk einen Dienst bereitstellt und auf Anfragen wartet.
- Ein Server kann in einer anderen Beziehung selbst Client sein.
- In der Realität kann nicht immer strikt unterschieden werden, wer Client und wer Server ist.



- Faustregel: Der Client initiiert die Verbindung und stellt die Anfrage, der Server wartet auf Anfragen.



Client:	Server:
<ul style="list-style-type: none"> ▪ nutzt die Dienste eines (oder mehrerer) Server ▪ kennt die Server Adresse und den Port des Dienstprozesses ▪ baut eine Verbindung zum Server auf ▪ kommuniziert mit dem Server (und hält sich dabei an das vom Server vorgegebene Protokoll) 	<ul style="list-style-type: none"> ▪ stellt einen Dienst zur Nutzung durch andere zur Verfügung (öffnet eine Verbindung und horcht auf Anfragen) ▪ erhält vom Betriebssystem eine oder mehrere Port-Nummern ▪ Nutzer müssen sich an das Server-Protokoll halten ▪ oft: Der Server-Prozess läuft im Hintergrund und wartet, dass ein Nutzer eine Verbindung zu ihm aufbaut.

Typische Beispiele für Server: Name Server (-Prozess), Web Server (-Prozess).

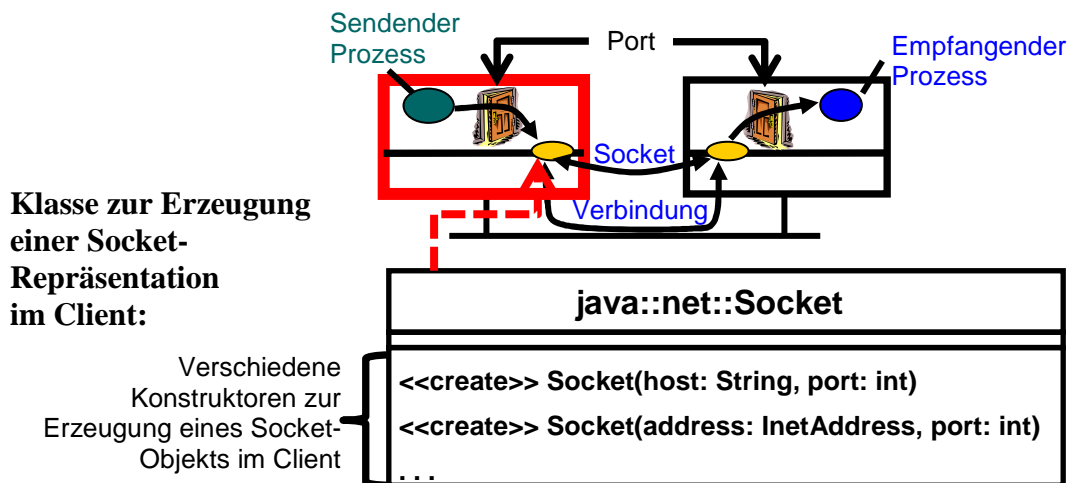
11.6 Implementierung in Java

- Eine Verbindung zwischen zwei Prozessen (und deren Hosts) ist definiert durch einen Socket der Form:

`<Protokoll, lokale IP, lokale Portnummer, ferne IP, ferne Portnummer>`
- Eine Netzwerkverbindung wird in Java aufgebaut und verwaltet mit Hilfe von Klasse **Socket** und Klasse **ServerSocket** (Socket-Objekte dienen als Verbindungsendpunkte)
- Zusätzliche Hilfsklassen, z.B.:
InetAddress: Darstellung der IP-Adresse (wie oben schon gesehen),
Streams: Für den Datentransfer zwischen den Sockets (wie bekannt).

11.6.1 Client Seite

Prinzip und Klasse für einen Client im Netzwerkprogramm:

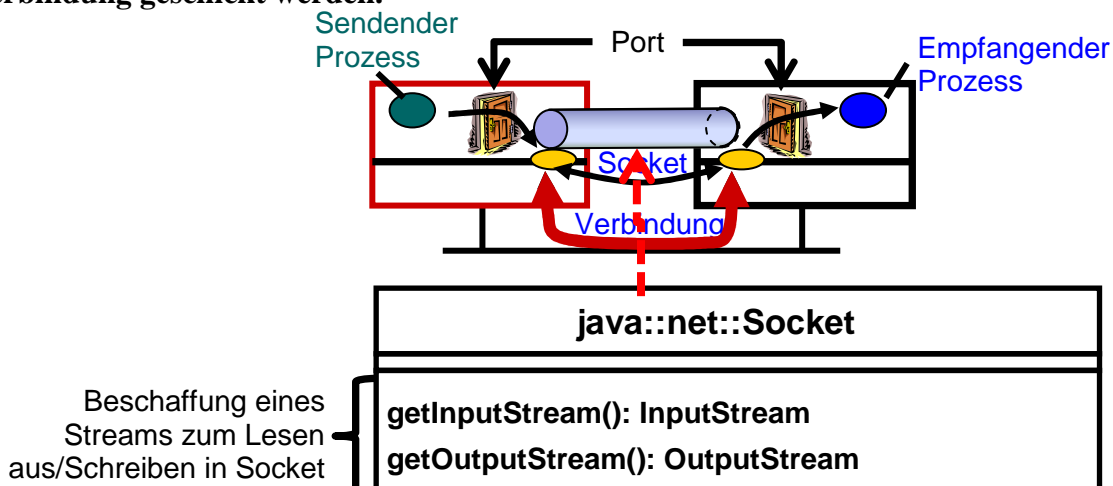


Schritt 1: Socket-Erzeugung und Verbindungsaufbau in Java

```
Socket clientSocket = new Socket(address, 6000);
```

Der erzeugende Thread (der diesen Code ausführt) ist blockiert, bis die TCP/IP-Verbindung zum Host mit Adresse **address** und Port 6000 steht oder ein Timeout eintritt.

Schritt 2: Ist die Verbindung erfolgreich aufgebaut, können Nachrichten über die Verbindung geschickt werden.



Für den Datenaustausch stellt ein **Socket**-Objekt Streams zur Verfügung.
Diese Stream-Objekte können direkt verwendet werden oder es können - wie schon bekannt - Streams mit Hilfe von Filterstreams je nach Bedarf geschachtelt werden.



Zu beachten:

- Es kann nur eine begrenzte Anzahl von Sockets aufgebaut werden.
Bei Kommunikationsende sollten daher Streams und Socket geschlossen werden.
Dazu wird am Socket die Methode **close()** aufgerufen.
- Sind noch zu dem Socket gehörige Streams offen, werden diese ebenfalls geschlossen.

Beispiel: Übertragung des String „Hello world“ vom Client-Prozess zum Server-Prozess (Client-Seite)

Schritt 1: Festlegung von Client und Server auf einen gemeinsamen Port
im Beispiel: localhost:6000

Schritt 2: Client Implementierung ←

Schritt 3: Server Implementierung

Client Implementierung:

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

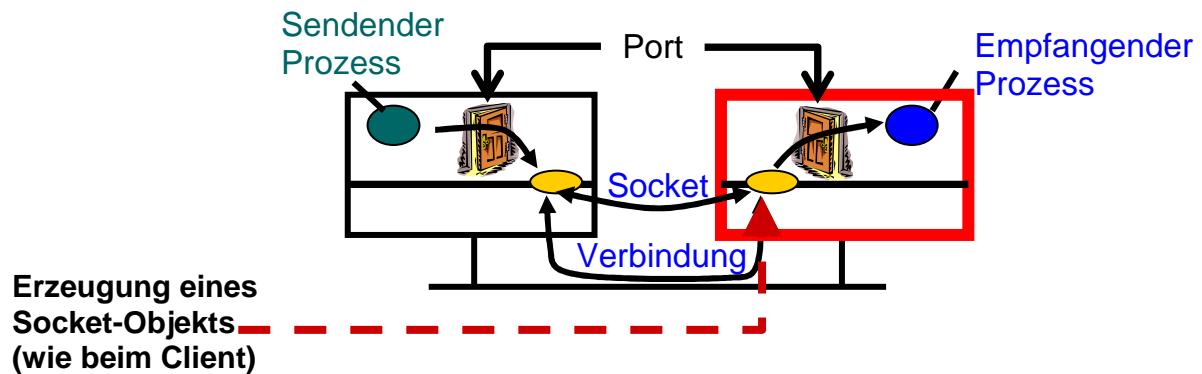
public class Client {

    public static void main(String argv[]) {
        try {
            Socket s = new Socket("localhost",6000);
            OutputStream o = s.getOutputStream();
            OutputStreamWriter out = new OutputStreamWriter(o);
            out.write("Hello world"); out.flush();
            out.close();
            o.close();
            s.close();
        }
        catch (UnknownHostException e) {
            System.err.println("Client: Unknown Host" + e.getMessage());
            System.exit(1);
        }
        catch (IOException e) {
            System.err.println("Client: IO-Error" + e.getMessage());
            System.exit(1);
        }
    }
}
```

Bei gepufferten Streams nicht vergessen: **flush()**.

11.6.2 Server Seite

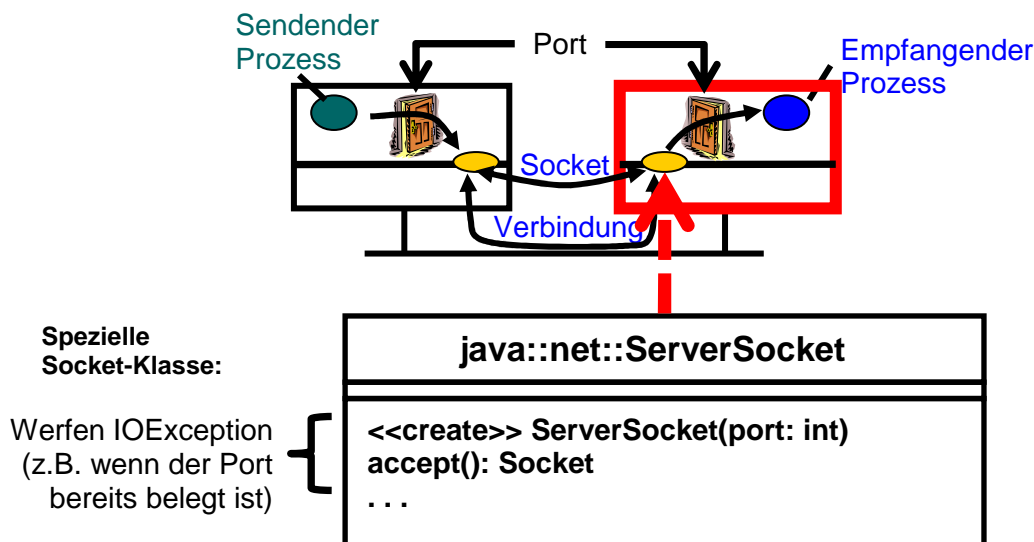
Prinzip und Klasse für einen Server im Netzwerkprogramm:



Der Socket-Aufbau und die Kommunikation erfolgt im Server analog zum Client. Aber: Server bauen keine eigene Verbindung auf, sondern horchen an ihrem zugewiesenen Port auf Eingaben und Anfragen.

Spezielle Methoden vereinfachen diese serverseitige Verbindungsvorbereitung und den serverseitigen Verbindungsaufbau:

- Methoden, um auf einen eingehenden Verbindungswunsch zu warten.
- Methoden, um nach erfolgreichem Verbindungsaufbau einen Socket zur Kommunikation mit dem Client zurückzugeben.



Schritt 1: Socket-Erzeugung für eine bestimmte Serveranwendung (bereitgestellter Dienst)

Der verwendete Parameter: Port-Nummer (im Beispiel: 6000), zu der sich Clients verbinden können.

Am Port 6000 soll also der Server-Prozess (in unserem Beispiel) warten.

ServerSocket servSocket = new ServerSocket(6000);

Schritt 2: Am Socket auf Anfragen von Clients warten

servSocket.accept();

Nach Aufruf der Methode **accept()** wartet bzw. blockiert der Server-Prozess solange, bis ein Client eine Verbindung (zum Host und Port) herstellt.

Nach dem erfolgreichen Aufbau einer Verbindung mit einem Client liefert `accept()` ein `Socket`-Objekt zurück.

Dieser Socket kann dann (wie bei Client-Anwendungen) zur Kommunikation mit der Gegenseite verwendet werden.


Anschließend steht der `ServerSocket` für einen weiteren Verbindungsaufbau zur Verfügung (oder kann mit `close()` geschlossen werden).

Beispiel: Übertragung des String „Hello world“ vom Client-Prozess zum Server-Prozess (Server-Seite)

Schritt 1: Festlegung von Client und Server auf einen gemeinsamen Port

im Beispiel: localhost:6000

Schritt 2: Client Implementierung

Schritt 3: Server Implementierung 

Server Implementierung:

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

public class Server {

    public static void main(String argv[]) {
        int input;
        try {
            ServerSocket servSock = new ServerSocket(6000);
            Socket sock = servSock.accept();
            InputStream i = sock.getInputStream();
            InputStreamReader in = new InputStreamReader(i);
            while ((input = in.read()) != -1)
            {
                System.out.print((char)input);
            }
            in.close();
            i.close();
            sock.close();
            servSock.close();
        }
        catch (IOException e) {
            System.err.println("Client: IO-Error" + e.getMessage());
            System.exit(1);
        }
    }
}
```

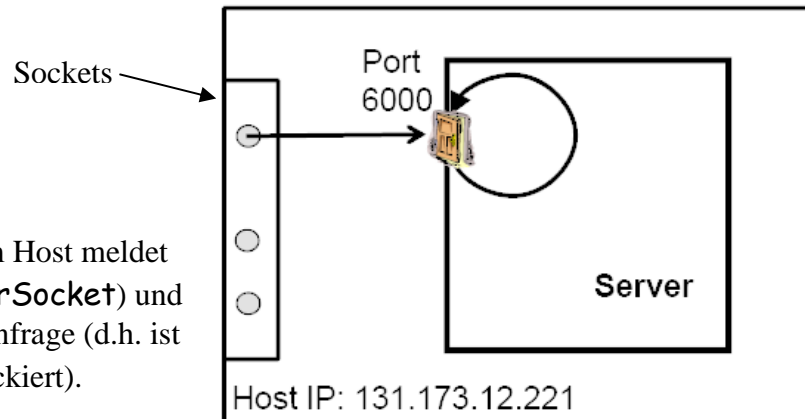
Server Seite: Auf die Verbindung horchen mit **accept()**

- Damit der Port (in unserem Beispiel: Port 6000) nicht blockiert wird, wird der von **accept()** erzeugte Socket auf einen freien Port gelegt.
- Der **ServerSocket** wäre somit prinzipiell wieder frei für eine weitere Anfrage.
- Analogie:
 - Klienten wollen in einem Bürogebäude (Server Host) etwas bearbeitet haben.
 - Sie wenden sich an die für sie vorgesehene Auskunftsstelle (Port).
 - Sie werden begrüßt und an einen freien Mitarbeiter verwiesen (anderer Port).

AblaufszENARIO:

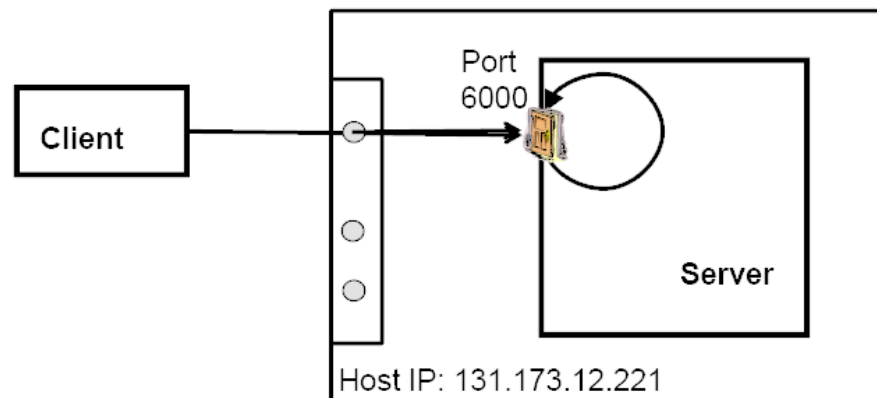
Schritt 1:

Der Server-Prozess auf einem Host meldet sich auf Port 6000 an (**ServerSocket**) und wartet auf eine Verbindungsanfrage (d.h. ist in der Methode **accept()** blockiert).



Schritt 2:

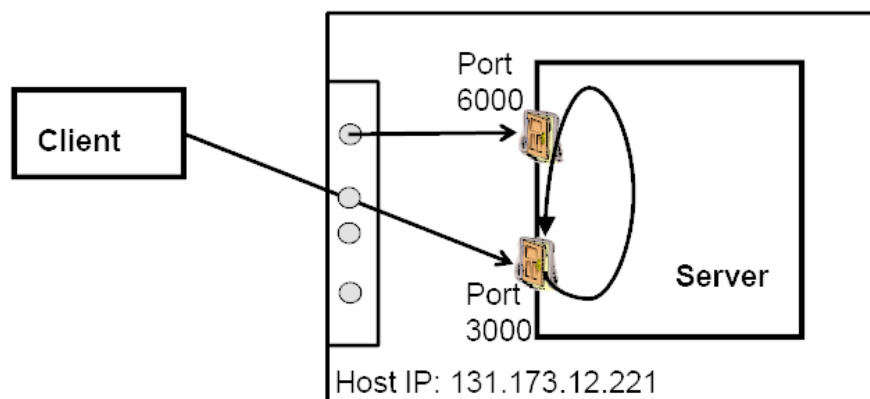
Der Client-Prozess verbindet sich über sein **Socket**-Objekt mit dem Server an dem entsprechenden Port (auf dem entsprechenden Host).



Der Client wartet, bis die Verbindung hergestellt ist (oder bis zum Timeout).

Schritt 3:

Der Server akzeptiert die Verbindung und erzeugt ein neues **Socket**-Objekt an einem anderen Port (z.B. mit Port-Nummer 3000).

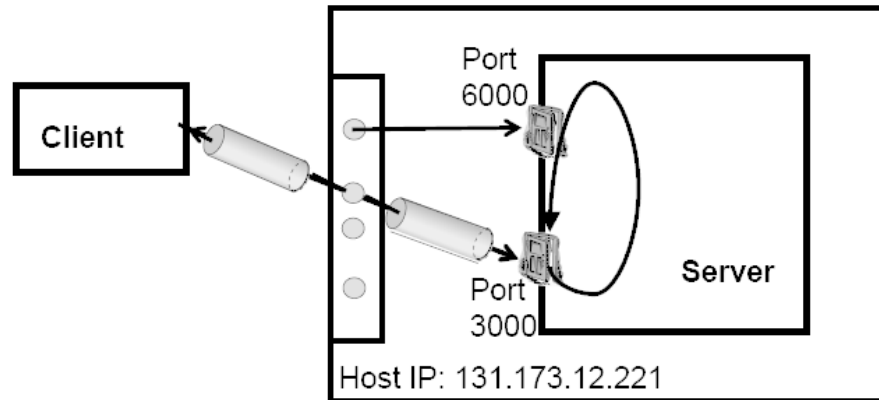


Schritt 4:

Durch die Sockets
kann auf einen
InputStream
bzw.

OutputStream
zugegriffen werden.
Über diese wird die
Kommunikation
abgewickelt. Der
verwendete **Socket**
ist solange blockiert.

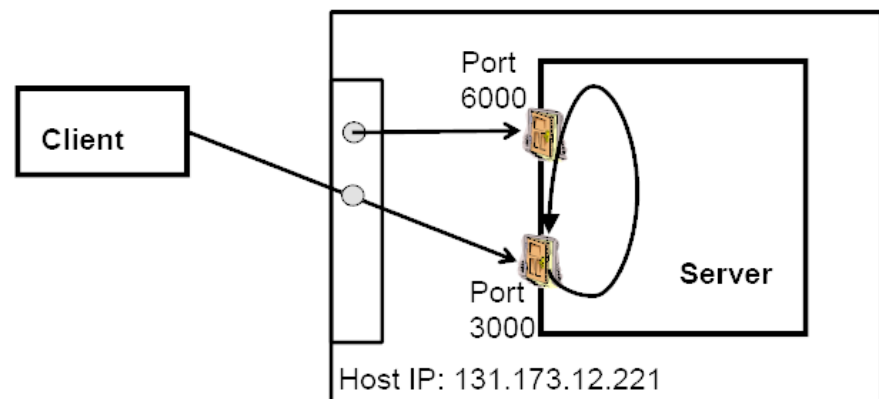
Tatsächlich ist auch der **ServerSocket** solange blockiert, da er im Standardfall im gleichen Thread verwaltet wird.



Bei Kommunikationsende: Die Ressourcen (Streams, **Socket**, ggf. auch **ServerSocket**) sollten dringend freigegeben werden.

Server Seite: Verarbeitung vieler Anfragen

Problem im
bisherigen Ablauf:
Port 6000 wird
zwar freigegeben.
Auf neue
Verbindungs-
anfragen horcht der
Server
(**ServerSocket**)
aber erst wieder,
wenn die
Kommunikation mit dem Client abgeschlossen ist.

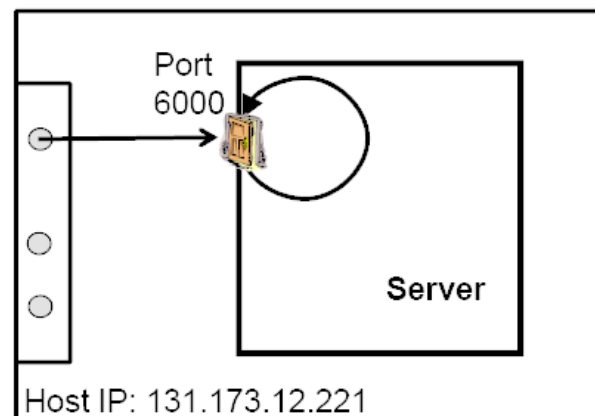


Abhilfe: Nebenläufigkeit (Implementierung mit Threads)

Nochmal von vorn, jetzt aber besser im Fall von vielen Anfragen ...

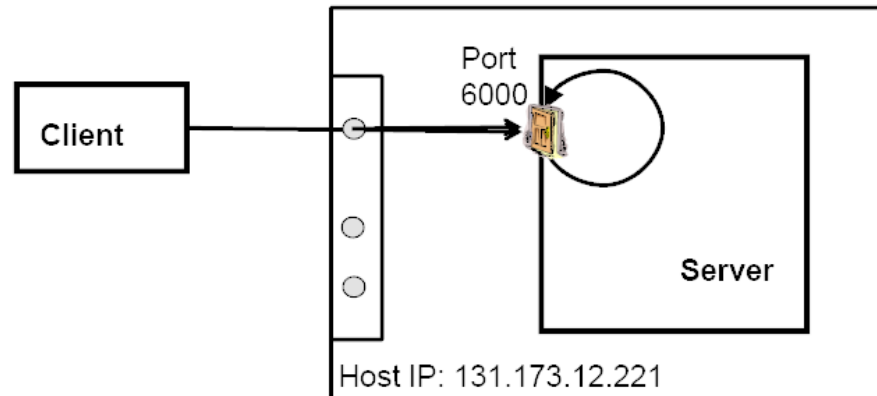
Schritt 1:

Der Server-Prozess auf einem Host meldet
sich auf Port 6000 an (**ServerSocket**)
und wartet auf eine Verbindungsanfrage
(d.h. ist in der Methode **accept()**
blockiert).



Schritt 2:

Der Client-Prozess verbindet sich über sein **Socket**-Objekt mit dem Server an dem entsprechenden Port (auf dem entsprechenden Host).



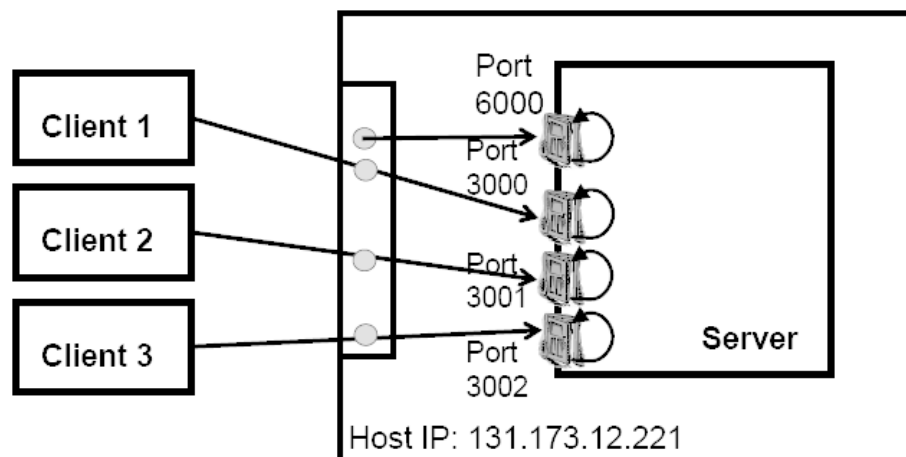
Der Client wartet, bis die Verbindung hergestellt ist (oder bis zum Timeout).

Schritt 3:

Ist die Verbindung hergestellt wird ein neuer Thread zur Kommunikation mit dem neuen **Socket** gestartet.

ServerSocket kann weitere Anfragen bearbeiten.
Mehrere Clients

können so „gleichzeitig“ versorgt werden (d.h. kein Client wird blockiert).



11.6.3 UDP Paket-Verbindung in Java

- Client und Server kommunizieren über Datagram-Pakete (keine stehende Verbindung)
<Protokoll, lokale IP, lokale Portnummer, ferne IP, ferne Portnummer>
- Netzwerkverbindung in Java mit Hilfe von
Klasse **Socket** und
Klasse **DatagramSocket**
(**Socket**-Objekte als Verbindungsendpunkte)
- Zusätzliche Hilfsklassen, z.B.:
DatagramPacket: zu versendendes Datagram-Paket
InetAddress: Darstellung der IP-Adresse
Streams: für den Datentransfer zwischen den Sockets

11.7 Höhere Kommunikation

- Es gibt viele höhere Verbindungen (höheres Abstraktionsniveau).
- Alle höheren Verbindungen bauen in der Regel auf Sockets auf.
- Beispiel: URL-Verbindungen (**URL**-Objekte und **URLConnection**-Objekt)

URI: Uniform Resource Identifier
Name für eine Ressource im Netz
Beispiel: `www.inf.uos.de/index.html`

URL: Uniform Resource Locator
Spezialisierung einer URI
Bindung an ein Netzwerkprotokoll (z.B. `http`, `ftp`, ...)
Beispiel: `http://www.inf.uos.de/index.html`