Allgemein

Um von einem Client zu einem Server eine Verbindung aufzubauen, werden in Java die TCP/IP-Protokolle verwendet. Dabei werden die Rechner durch ihre **IP-Adresse** identifiziert. Zusätzlich muss noch die **Portnummer** (16 Bit ➔ max. 65536) mit angegeben werden.
Die Port Nummern < 1024 werden dabei als **known**-Ports bezeichnet und sind für diverse Systemdienste (Telnet, Ftp, http usw.) reserviert.
Deswegen müssen eigene (Server-)Programme auf einem Port >1024 laufen, wobei auch hier inzwischen einige Portnummern durch diverse Applikationen belegt sein können (z.B. mysql = 3306 oder Oracle-Listener=1524).

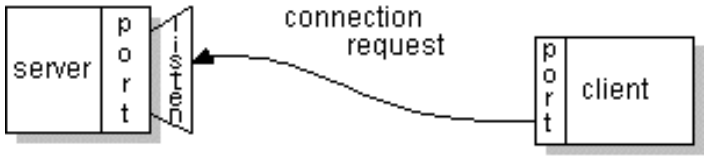
Für den Datenaustausch werden Kommunikationskanäle gebraucht, die Sockets genannt werden.

Abb. 1.1: Datenkommunikation über Sockets

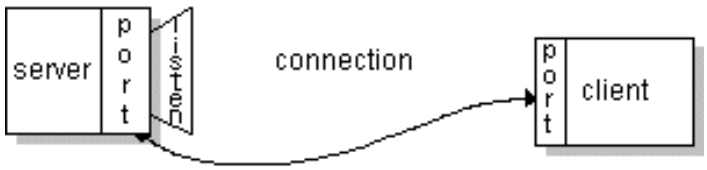
Sockets (`package java.net`)

Server-Seite: Ein Server läuft auf einem Rechner und besitzt einen Socket, der an eine bestimmte **Portnummer** gebunden ist. Der Server wartet, indem er an seinem **ServerSocket** auf eingehende Verbindungsanfragen „hört".

Client-Seite: Ein Client kennt den **Hostnamen**, auf dem die Server-Anwendung läuft sowie die **Portnummer** an dem der Server wartet. Beim Verbindungsaufbau sendet der Client eine Anfrage und versucht sich mit dem Server zu verbinden. Der Client muss sich dabei selbst identifizieren. Dazu belegt er eine **lokale Portnummer** auf seinem Rechner, die für die Dauer der Verbindung verwendet wird. Diese lokale Portnummer wir normalerweise vom System zugeteilt.



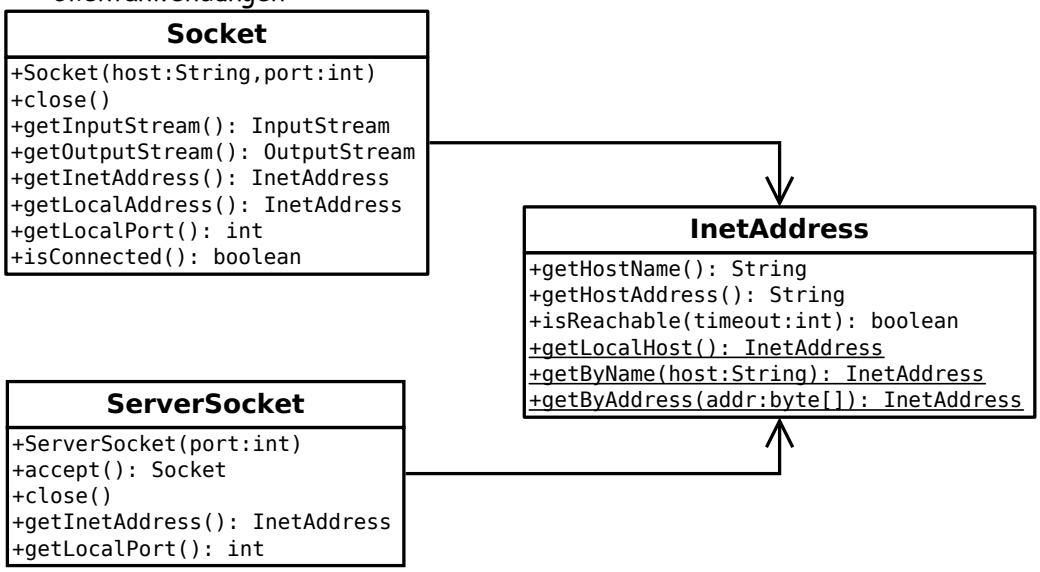
Geht alles gut, wird der Server die Verbindung akzeptieren. Der Server wird einen neuen Socket mit dem gleichen lokalen Port verbinden und als „Remote-Endpunkt" die Adresse/Port des Clients setzen. Der Server benötigt diesen neuen Socket, damit er wieder am originalen Socket auf weitere Verbindungsanfragen regieren kann.



➔ Ein Socket repräsentiert einen Endpunkt einer Kommunikationsverbindung und arbeitet auf einem Port.

Wichtige Klasse in Java für die Client-/Server-Programmierung sind:

- Klasse **Socket**: Wird von Clients verwendet um Verbindung mit Server herzustellen, sowie auf Serverseite, um mit verbundenen Clients Daten auszutauschen
- Klasse **ServerSocket**: für die Serverseite zum Warten auf Verbindungsanfragen von Clientanwendungen



Sind auf beiden Seiten Sockets verfügbar, so erfolgt die Kommunikation aus Sicht der Anwendung über je zwei Streams, einen **`InputStream`**, der Daten von einem Socket liest, und einen **`OutputStream`**, der Daten in einen Socket schreibt. Von jedem Socket können entsprechende Stream-Referenzen abgefragt werden.

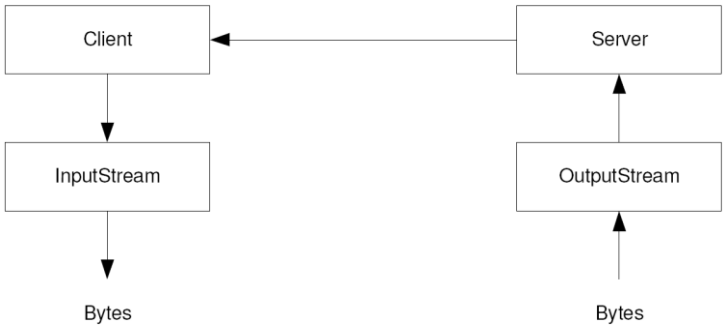
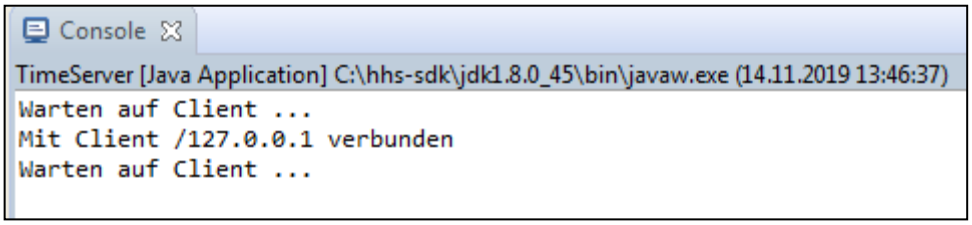


Abb. 1.2: Streams und Sockets (hier Server an Client)

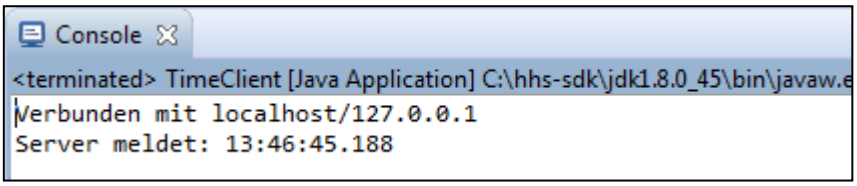
Beispielanwendungen – Uhrzeit-Dienst

Die Aufgabe des Servers soll es sein, auf die Anfrage eines Clients zu warten und diesem bei einer Anfrage die aktuelle Uhrzeit mitzuteilen.

Meldungsfenster des TimeServers



Meldungsfenster des TimeClients



Quellcode „TimeClient“:

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

public class TimeClient {
  public static void main(String[] args) {

    try (Socket toServer = new Socket("localhost", 4711)){

      System.out.println("Verbunden mit " +
                         toServer.getInetAddress());


      InputStream fromServer = toServer.getInputStream();

      byte[] daten = new byte[100];
      int num = fromServer.read(daten);


      String datumZeit = new String(daten, 0, num);
      System.out.println("Server meldet: "+datumZeit);

      fromServer.close();
    } catch (IOException e) { ... }
  }
}
```

Client baut zum lokalen Rechner eine Verbindung auf. Anstelle von `localhost` kann auch eine andere IP-Adresse verwendet werden. Ist der Rechner nicht erreichbar, wird eine Exception geworfen, die abgefangen werden muss.

Bei erfolgreichem Verbindungsaufbau gibt der Client die entsprechenden IP Adresse des Servers aus

InputStream vom Socket abfragen

Byte-Array zur Aufnahme der zu lesenden Daten anlegen und Daten von InputStream / Socket lesen.

Aus allen gelesenen Bytes (Anzahl num) wird ein String ab offset 0 initialisiert und ausgegeben.

Quellcode „TimeServer“:

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

public class TimeServer {
  public static void main(String[] args) {
    try (ServerSocket server = new ServerSocket(4711)){
      while(true){
        System.out.println("Warten auf Client…");
        Socket client = server.accept();

         System.out.println("Mit Client "+   client.getInetAddress()+" verbunden");

        OutputStream toClient = client.getOutputStream();

        LocalTime time = LocalTime.now();

        toClient.write( time.toString().getBytes() );


        toClient.close();
        client.close();
      }
    } catch (IOException e) { ... }
} }
```

Objekt der Klasse **ServerSocket** erzeugen, um auf Anfragen auf Port 4111 zu warten.

Methode **accept()** wartet solange, bis ein Client einen Verbindungsaufbau wünscht und liefert dann ein neues **Socket**-Objekt für die Kommunikation mit diesem Client zurück.

OutputStream vom neuen Socket zum Client holen.

Aktuelle Uhrzeit abfragen

Byte-Array über OutputStream-Objekt an den Client zurückschreiben.

Uhrzeit als String holen und diesen in ein Byte-Array umwandeln .

Hinweis: In der Praxis wird man nicht direkt read/write von InputStream/OutputStream verwenden (da unhandlich), sondern mit diesen einen BufferedReader bzw. PrintWriter initialisieren!