



Socket-Programmierung in Java

Kaffee im Netz



Die Programmierung von Netzwerken über Sockets gilt als **kompliziert**. Unter Java können Sie diese Vorurteile **getrost über Bord werfen**. Es ist **fast ein Kinderspiel**, eine Netzwerkverbindung **zwischen zwei Klassen** aufzubauen.

OLIVER MÜLLER

Die Programmierung von Netzwerk-Sockets wird in der Fachwelt als schwierig angesehen. Wenn Sie in Java programmieren, können Sie diese Auffassung ohne Weiteres als Vorurteil abtun.

In Java existieren standardmäßig vorgefertigte Klassen, über die Sie auf einfachste Weise zwischen Java-Programmen Netzwerkverbindungen herstellen können.

QUICK INDEX

- ▶ **Was ist ein Socket?**
Ein Socket stellt Verbindungen zwischen Programmen im Netzwerk her.
- ▶ **Klassisch**
Zum Arbeiten mit Sockets existieren in Java standardmäßig spezielle Klassen.
- ▶ **ABC-Schützen**
Sockets enthalten jeweils einen Stream zum Lesen und Schreiben.
- ▶ **Server - die Erste**
Ein einfacher Server besteht aus nicht mehr als einer Klasse.
- ▶ **Klientel**
Der grundlegende Aufbau eines Clients ist recht einfach.
- ▶ **Server mit Mehrwert**
Mit Multithreading können Server mehrere Clients parallel abfertigen.

■ Was ist ein Socket?

Egal, ob Sie nun unter Unix oder mit Windows arbeiten, sobald es an die Programmierung von Netzwerken geht, taucht früher oder später der Begriff "Socket" auf. Die Sockets wurden an der University of California at Berkley entwickelt. Sie sollten über das TCP/IP-Netzwerk eine ähnliche Ein- und Ausgabefunktionalität ermöglichen wie es die Ein- und Ausgabe im lokalen Dateisystem bereitstellt.

Bevor Sie einen Socket verwenden, müssen Sie ihn öffnen. Danach können Sie ähnlich wie bei einer Datei auf diesen Socket lesend und/oder schreibend zugreifen, d.h. Sie können Daten aus dem Socket lesen oder dahin ausgeben.

Wurden alle Daten übertragen, schließen Sie den Socket wieder.

Ein Socket ist kein einfacher Datenspeicher wie eine Datei auf der Festplatte, sondern verbindet zwei Programme über das Netzwerk. Die Daten, die Ihre Client-Anwendung liest, muss eine andere Applikation liefern, nämlich das Server-Programm. Damit rücken Sockets schon in eine etwas höhere Ebene als die einfache Dateiein- und -ausgabe. Jede Ihrer Anwendungen – Server und Client – besitzt einen eigenen Socket. Der Server-Socket wartet darauf, dass ein Client über einen TCP/IP-Port eine Verbindung aufbaut.

Von Steckdosen und anderen Phänomenen

Diesen Mechanismus kann man sich wie folgt verdeutlichen: Der Server ist dabei eine Steckdose, der Client der Stecker. Der Name bzw. die IP-Adresse des Server-Hosts im Netzwerk ist vergleichbar mit dem Zimmer, in dem sich die Steckdose befindet. Wie bei den Steckdosen in Ihrer Wohnung können Sie auch bei Netzwerk-Sockets mehrere Dosen (= Server) anlegen. So wie Sie Ihren Staubsauger im Wohnzimmer, im Schlafzim-

DATEN AUS INPUTSTREAM LESEN

Methode	Beschreibung
<code>int read()</code>	Liest ein Byte aus dem Stream und gibt es zurück. Wird das Ende des Streams erreicht, wird -1 zurückgegeben.
<code>int read(byte b[])</code>	Liest Daten aus dem Stream und speichert sie im Byte-Array <code>b[]</code> . Die Methode gibt die Anzahl der gelesenen Bytes zurück oder -1, wenn das Stream-Ende erreicht wurde.
<code>int read(byte b[], int of, int len)</code>	Liest Daten aus dem Stream und speichert sie im Byte-Array <code>b[]</code> ab Offset <code>of</code> . <code>len</code> gibt dabei die maximale Anzahl der zu lesenden Bytes an. Zurückgegeben wird die Anzahl der tatsächlich gelesenen Bytes. Wird das Ende des Streams erreicht, wird -1 zurückgeliefert.



mer oder im Esszimmer anschließen, legen Sie beim Socket-Client über den Hostnamen bzw. dessen IP-Adresse fest, zu welchem Socket-Server Sie eine Verbindung aufbauen möchten. Allerdings müssen Sie darauf achten, dass der Server auf dem Host läuft. Es bringt ja auch nichts, wenn Sie Ihren Staubsauger in der Gartenlaube anschließen wollen, wenn dort gar keine Steckdose vorhanden ist.

Bei Auslandsreisen wird Ihnen sicherlich aufgefallen sein, dass nicht alle Steckdosen gleich sind. Der Versuch, Ihr Radio mit dem Eurostecker in eine US-amerikanische Steckdose hineinzustöpseln, scheitert schon rein physikalisch.

Einen ähnlichen Effekt können Sie auch bei Sockets erleben: Hier müssen die Ports zusammenpassen. Client- und Server-Socket müssen den gleichen TCP/IP-Port verwenden, damit die Verbindung überhaupt zustande kommt – es nützt nichts, wenn der Client versucht, einen Server auf Port 5000 anzusprechen, wenn der Server auf Port 5555 horcht.

Zusammengefasst heißt das, dass die beiden Anwendungen jeweils mindestens einen Socket verwenden. Der Server reagiert auf Schreib- und Leseanfragen des/der Client(s). Sein Socket wartet dabei auf Verbindungen von Clients über einen bestimmten TCP/IP-Port. Der Client seinerseits baut über seinen Socket eine Verbindung zum Server-Socket auf. Zu diesem Zweck muss er den Port und den Host des Servers seinem Socket übergeben. Nur mit diesen Angaben kann eine Verbindung erfolgreich aufgebaut werden. Danach kann der Client mit dem Socket ähnlich wie bei einem lokalen Dateizugriff arbeiten.

Klassisch

Java liefert Ihnen von Haus aus Klassen mit, über die Sie sehr einfach mit Sockets arbeiten können. Hier sind in erster Linie die Klassen *Socket*, *ServerSocket* und *InetAddress* des Pakets *java.net* wichtig.

Instanzen der Klasse *InetAddress* dienen dazu, einen Rechner (= Host) im Netz zu adressieren. Die Objekte enthalten dabei sowohl die IP-Adresse, als auch den Namen des Hosts. Diese Informationen können über die Methoden *getHostName* bzw. *getHostAddress* abgefragt werden. Um ein *InetAddress*-Objekt zu erzeugen, existieren zwei statische Methoden:

getByName erzeugt ein Objekt für den als String-Argument übergebenen Host. Hierbei kann ein Hostname oder eine IP-Adresse übergeben werden. Ein Beispiel für *getByName* wäre

```
InetAddress pcm;
pcm = InetAddress.getByName
("www.pcmagazin.de");
```

getLocalHost hingegen erzeugt eine *InetAddress*-Instanz für den eigenen Host „localhost“. Es werden keine Argumente erwartet, weil die eigene ja bekannt ist.

```
InetAddress lhost;
lhost = InetAddress.getLocalHost();
```

DATEN IN OUTPUTSTREAM SCHREIBEN

Methode	Beschreibung
<code>void write(int b)</code>	Schreibt den Wert <code>b</code> in den <i>OutputStream</i> .
<code>void write(byte b[])</code>	Gibt die Bytes aus dem Array in den <i>OutputStream</i> aus.
<code>void write(byte b[], int off, int len)</code>	Die Bytes aus dem Array <code>b[]</code> ab der Position <code>off</code> werden in den Stream ausgegeben. Wie viele Bytes aus dem Array geschrieben werden, gibt der Wert <code>len</code> an.

LISTING 1

```
/* einfacher Server */

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

public class SingleSampleServer {
    public static void main(String args[]) {
        System.out.println("SingleSampleServer\n");
        if(args.length != 1) {
            System.err.println("Syntax: java SingleSampleServer <Port>");
            System.exit(1);
        }

        try {
            // ServerSocket-Instanz erzeugen
            ServerSocket myserver = new ServerSocket(new Integer(args[0]).
            intValue());

            System.out.print("Warte auf Verbindung (Port " + args[0] + ")...");

            // Auf Verbindung warten und
            // Socket für den Server erzeugen
            Socket socket = myserver.accept();

            System.out.println("OK");

            // Streams holen
            InputStream input = socket.getInputStream();
            OutputStream output = socket.getOutputStream();

            String s = "";
            int ch;

            // Zeichen für Zeichen vom Client
            // empfangen
            while((ch = input.read()) != -1) {
                if(ch == '\n') {
                    s = s + "\r\n";

                    // String in Großbuchstaben wandeln
                    // und zum Client zurückgeben
                    output.write(s.toUpperCase().getBytes());

                    System.out.println("String verarbeitet: " + s);
                    s = "";
                } else if(ch != '\r') {
                    s = s + (char)ch;
                }
            }

            System.out.println("Verbindung beendet");

            // Streams und Sockets schließen
            input.close();
            output.close();
            socket.close();
            myserver.close();

        } catch(IOException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```



LISTING 2

```
/* Ein Socket-Beispiel */

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

public class SampleClient {
    public static void main(String args[]) {
        System.out.println("SampleClient\n");

        if(args.length != 2) {
            System.err.println("Syntax: java SampleClient <Host> <Port>");
            System.exit(1);
        }

        try {
            // Adresse erzeugen
            InetAddress myhost = InetAddress.getByName(args[0]);

            System.out.print("Verbinde zu Host \"" + myhost.getHostAddress() + "\" (" + myhost.getHostAddress() + ")...");

            // Socket für Verbindung erzeugen
            Socket socket = new Socket(myhost, new Integer(args[1]).intValue());

            // Streams anlegen
            InputStream input = socket.getInputStream();
            OutputStream output = socket.getOutputStream();

            // Timeout für das Socket setzen
            socket.setSoTimeout(600);

            System.out.println("Ein Leerstring beendet das Programm.");

            String str = "";
            int chi;
            char ch;
            while(true) {
                while((chi = System.in.read()) != -1) {
                    ch = (char)chi;
                    if(ch == '\n')
                        break;
                    else if(ch != '\r')
                        str = str + ch;
                }
                if(str == "")
                    break;
                str = str + "\r\n";

                // String über Socket-OutputStream
                // ausgeben
                output.write(str.getBytes());

                str = "";
                while((chi = input.read()) != -1) {
                    ch = (char)chi;
                    if(ch == '\n') {
                        System.out.println(str);
                        break;
                    } else if(ch != '\r')
                        str = str + ch;
                }

                str = "";
            }

            // Streams und Socket schließen
            input.close();
            output.close();
            socket.close();

        } catch(UnknownHostException e) {
            System.err.println(e.toString());
            System.exit(1);
        } catch(IOException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```



Beide Methoden können die Exception *UnknownHostException* auslösen.

Im Client kommt als Socket eine Instanz der Klasse *Socket* zum Einsatz. Die Konstruktoren der Klasse *Socket* können ein *InetAddress*-Objekt als Argument verarbeiten und so den Host identifizieren. Alternativ kann der Host-Name oder seine IP-Adresse als String angegeben werden. Last but not least muss der Socket auch wissen, auf welchem Port der Server läuft, zu dem eine Verbindung aufgebaut werden soll. Hierzu existiert jeweils der zweite Parameter der Konstruktoren *Socket* (*String host, int port*) und *Socket(InetAddress address, int port)*. Beide Konstruktoren können im Übrigen die Exception *IOException* auslösen. Wurde das Socket-Objekt ohne Fehler erzeugt, steht auch schon die Verbindung.

Server-seitig kommt die Klasse *ServerSocket* zum Einsatz. Der Konstruktor unterscheidet sich in einem wesentlichen Detail von der Client-Variante: Er erwartet nur einen einzigen Parameter, nämlich den Port. Die Angabe eines Hosts wäre bei einem Server unsinnig, dieser läuft ohnehin auf dem Rechner, auf dem er gestartet wurde. Eine Verbindung wird nicht aufgebaut, da ein Server-Socket grundsätzlich darauf wartet, dass Clients Verbindungen aufbauen. Nach dem Erzeugen des *ServerSocket*-Objekts ist der Socket lediglich "verbindungsbereit". Geht beim Erzeugen des Sockets irgendetwas schief, meldet sich die Exception *IOException*.

■ ABC-Schützen

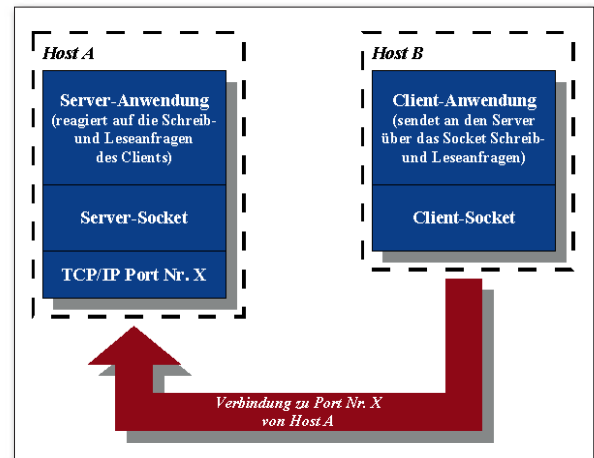
Sockets können über einen Stream lesen und schreiben. Die Methoden *getInputStream()* und *getOutputStream()* der Klasse *Socket* geben jeweils ein Objekt der Klasse *InputStream* bzw. *OutputStream* zurück. Über diese Objekte können Daten aus Sockets gelesen und in diese geschrieben werden.

Damit der Server mit seinen Streams arbeiten kann, wird über die Methode *accept()* des *ServerSocket*-Objekts eine Instanz von *Socket* erzeugt. Dieses Socket-Objekt stellt dann die Ein- und Ausgabe-Streams für den Server zur Verfügung. *accept()* wartet, bis eine Verbindung vom Client aufgebaut wird. Kommt es schließlich und endlich zu diesem Verbindungsaufbau, erzeugt *accept()* ein Socket-Objekt und gibt es zurück.

Sobald jetzt der Client Daten in seinen *OutputStream* schreibt, landen diese Daten im *InputStream* des *ServerSocket*-Objekts. Der Server liest diese jetzt ein und schickt seine adäquate Antwort in seinen *OutputStream*. Diese Antwort fließt nun über das Netzwerk und die Socket-Verbindung zurück und landet im *InputStream* des Client-Sockets. Die beiden Streams stellen die Methoden *read()* bzw. *write()* zum Lesen und Schreiben von Daten bereit.

Näheres zu diesen Methoden halten die Textkästen "Daten aus *InputStream* lesen" und "Daten in *OutputStream* schreiben".

Werden Sockets und Streams nicht mehr verwendet, müssen sie über die Methode *close()* geschlossen werden. Dabei schließen Sie zuerst die Streams über diese *close*-Methode, dann den zugehörigen Socket. Danach herrscht wieder die beste Ordnung im System.



NETZWERKVERBINDUNG über Sockets.

■ Server – die Erste

Für einen einfachen Server benötigen Sie nicht mehr als eine Klasse und eine Methode *main()*. Das Listing 1 "SingleSampleServer.java" zeigt Ihnen einen solchen einfachen Server. Zunächst wird ein *ServerSocket*-Objekt erzeugt. Der Port wird über die Kommandozeile, d. h. über das Array *args[]*, übergeben.

Nachdem das *ServerSocket*-Objekt existiert, erzeugt das Programm einen

LISTING 3

```
/* MultiSampleServer */

import java.net.*;
import java.io.*;
import MSSClientThread;

public class MultiSampleServer {
    protected static int instances = 0;

    public static void main(String args[]) {
        System.out.println("MultiSampleServer\n");

        if(args.length != 1) {
            System.err.println("Syntax: java MultiSampleServer <Port>");
            System.exit(1);
        }

        try {
            // ServerSocket-Instanz erzeugen
            ServerSocket myserver = new ServerSocket(new Integer(args[0]).intValue());

            System.out.println("Bereit für Verbindungen über Port " + args[0] + "...");

            while(true) {
                // Auf Verbindung warten und
                // Socket für die Verbindung erzeugen
                Socket socket = myserver.accept();

                // Thread für die neue Verbindung
                // erzeugen und starten
                new MSSClientThread(instances++, socket).start();
            }
        } catch(IOException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```



Socket. Dies geschieht mittels der Methode `accept()` von `ServerSocket`. Steht die Verbindung, werden die Streams in die Variablen `input` und `output` geholt.

Jetzt folgt auch schon die zentrale Schleife des Programms. Hier wird über `input.read()` so lange vom Socket gelesen, bis ein Zeilenwechsel (`\n`) erfolgt. Der bis dahin erzeugte String wird in Großbuchstaben umgewandelt und an den `OutputStream output` geschickt.



Wenn Sie einen String an die Methode `write()` von `OutputStream` übergeben wollen, verwenden Sie `getBytes()`. Auf diese Weise wird Ihr String zu einem Byte-Array umgewandelt.

Die Funktion, einen String vom Client zu empfangen und in Großbuchstaben umgewandelt zurückzuschicken, ist lediglich ein einfaches Beispiel. In dieser Schleife könnten Sie auch jeden anderen Netzwerkdienst implementieren.

Ist die Verbindung beendet, werden die Streams und die Socket-Objekte geschlossen.

Klientel

Der Client (Listing 2 „SampleClient.java“) erzeugt zunächst eine `InetAddress` und einen Socket mit diesem `InetAddress`-Objekt. Die richtigen Werte für den Host werden über die Kommandozeile übergeben. Jetzt werden die Streams geholt und es wird

ein Timeout von sechs Sekunden gesetzt, also bis zu sechs Sekunden gewartet, bevor eine Verbindung als gescheitert gemeldet wird. Nun folgt eine Schleife, die lediglich Strings einliest und diese über den Socket an den Server sendet. Wird eine Leerzeile eingegeben, beendet sich das Programm automatisch. Zuletzt werden die Streams und des Sockets geschlossen.

Wenn Sie jetzt den kompilierten und gelinkten Server in einem Kommandozeilenfenster durch

```
java SingleSampleServer 5000
```

starten, können Sie den Client mit

```
java SampleClient localhost 5000
```

in einem anderen Fenster verwenden ein.



Statt 5000 können Sie auch einen anderen Port eingeben, der noch nicht verwendet wird. Starten Sie den Server auch einmal auf einem anderen Host im Netzwerk. Statt „localhost“ setzen Sie dann den Namen dieses Hosts beim Aufruf des Clients ein.

Server mit Mehrwert

So schön einfach der Server auch ist, so eingeschränkt ist seine Funktionalität. Er kann nur einen einzigen Client bewältigen. Unter Java ist es jedoch kein großes Problem in den Griff zu bekommen. Am besten verpackt man jede Verbindung zwischen Client und Server in einen eigenen Thread. Auf diese Weise können die Clients ohne großen Aufwand parallel abgefertigt werden, ohne sich gegenseitig in die Quere zu kommen. Außerdem müssen Sie keinen Gedanken an die Verwaltung der einzelnen Clients verschwenden.

Der grundlegende Aufbau des Servers (Listing 3) hat sich nicht stark verändert. Der Verbindungsaufbau via `accept()` wurde durch eine Endlos-Schleife ersetzt. Darin werden die über `accept()` aufgebauten Verbindungen in Objekte der Klasse `MSSClientThread` verpackt. Diese verwalten nach dem Aufruf der von `Thread` geerbten Methode `start()` die Verbindungen selbstständig.

Der Code hierfür findet sich in der Methode `run()` (Listing 4). Diese Methode wird durch den Code der Basis-Klasse `Thread` gestartet und läuft dann wie eine unabhängige Anwendung. In `run()` werden für jeden Client getrennt die selben Operationen wie beim `SingleSampleServer` realisiert.

UR

LISTING 4

```
/* Thread für den MultiSampleServer */
import java.net.*;
import java.io.*;

class MSSClientThread extends Thread {
    protected int instance;
    protected Socket socket;

    public MSSClientThread(int inst, Socket sock) {
        instance = inst;
        socket = sock;
    }

    public void run() {
        try {
            InputStream input = socket.getInputStream();
            OutputStream output = socket.getOutputStream();

            String s = "";
            int ch;

            System.out.println("(" + instance + ") Verbindung aufgebaut");

            // Zeichen für Zeichen vom Client
            // empfangen
            while((ch = input.read()) != -1) {
                if(ch == '\n') {
                    s = s + "\r\n";

                    // String in Großbuchstaben wandeln
                    // und zum Client zurückgeben
                    output.write(s.toUpperCase().getBytes());

                    System.out.println("(" + instance + ") String verarbeitet: " + s);
                    s = "";
                } else if(ch != '\r') {
                    s = s + (char)ch;
                }
            }

            System.out.println("(" + instance + ") Verbindung beendet");

            // Streams und Sockets schließen
            input.close();
            output.close();
            socket.close();

        } catch(IOException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```