

Musterlösungen zu den Hausaufgaben auf  
Blatt 10  
der Übungen zur Vorlesung  
“Grundlagen Betriebssysteme und Systemsoftware

---

G.Groh, 21.01.2008

## Aufgabe 1.1 Lösungsvorschlag

Ein Kommunikations-Partner spricht wahllos alle erreichbaren anderen Partner unerwartet an

Ein Kommunikations-Partner spricht einen speziellen, anderen Partner unerwartet an

Ein Kommunikations-Partner wartet darauf, von einem beliebigen anderen Partner angesprochen zu werden

Ein Kommunikations-Partner wartet darauf, von einem speziellen, anderen Partner angesprochen zu werden

## Aufgabe 1.1 Lösungsvorschlag

Ein Kommunikations-Partner spricht wahllos alle erreichbaren anderen Partner unerwartet an

Ein Kommunikations-Partner spricht einen speziellen, anderen Partner unerwartet an

Ein Kommunikations-Partner wartet darauf, von einem beliebigen anderen Partner angesprochen zu werden

Ein Kommunikations-Partner wartet darauf, von einem speziellen, anderen Partner angesprochen zu werden

## Aufgabe 1.1 Lösungsvorschlag

Ein Kommunikations-Partner spricht wahllos alle erreichbaren anderen Partner unerwartet an

Ein Kommunikations-Partner spricht einen speziellen, anderen Partner unerwartet an

Ein Kommunikations-Partner wartet darauf, von einem beliebigen anderen Partner angesprochen zu werden

Ein Kommunikations-Partner wartet darauf, von einem speziellen, anderen Partner angesprochen zu werden

Eine Seite wartet explizit darauf angesprochen zu werden. Dabei kommt für den Beginn der Kommunikation auf Ebene der Sockets jede Gegenseite in Frage, die den Ort für die Kommunikation (Host, Port) kennt. Erst im Laufe der Kommunikation kann es sich herausstellen, dass es sich nicht um einen korrekten Gesprächspartner handelt. Durch Einsatz von Nebenläufigkeit lassen sich auch die ersten beiden Arten der Kommunikation erreichen.

## Aufgabe 1.1 Lösungsvorschlag

Eine Seite (der Server) erstellt ein Objekt der Klasse **ServerSocket**. Diese definiert den Ort für den Beginn einer Kommunikation. Es ist also notwendig einen Port anzugeben. Dieser Port gilt wie gesagt für den Beginn der Kommunikation. Ist der **ServerSocket** eingerichtet, wird er in den Zustand „warten auf Verbindungswunsch“ versetzt. Dies geschieht mit dem Aufruf der Methode **accept()** des **ServerSocket**-Objekts. Die Methode wird erst beendet, wenn eine Verbindung zustande kommt (passives Warten).

## Aufgabe 1.1 Lösungsvorschlag

Anschließend versucht die andere Seite (der Client) zu einem beliebigen Zeitpunkt eine Verbindung mit dem Server aufzubauen. Dazu wird ein Objekt der Klasse **Socket** instanziiert. Dabei muss der Rechner (Host) und der Port angegeben werden, auf dem die Gegenseite auf Verbindungswünsche wartet. Ist das Socket-Objekt erfolgreich instanziiert, können mit Hilfe der Methoden **getOutputStream()** und **getInputStream()** Streams für die weitere bidirektionale Kommunikation erstellt werden.

## Aufgabe 1.1 Lösungsvorschlag

Auf der Seite des Servers geschieht in der Zwischenzeit folgendes. Die **accept()**-Methode registriert den Verbindungswunsch des Clients, kreiert auf der Seite des Servers ebenfalls ein neues Socket-Objekt und gibt dieses bei der Beendigung von **accept()** zurück. Auf Serverseite existieren zu diesem Zeitpunkt also zwei verschiedene Sockets: ein Serversocket, der bis zum nächsten Aufruf von **accept()** nicht mehr auf ankommende Verbindungswünsche wartet und ein Socket S der mit dem Socket auf der Clientseite verbunden ist. Auch der Socket S bietet über **getOutputStream()** und **getInputStream()** Streams zur direkten, bidirektionalen Kommunikation mit dem Client an. Dabei kann was auf der Server-Seite mit **write()** des OutputStreams geschrieben wird, auf der Client-Seite mit **read()** des InputStreams gelesen werden und umgekehrt.

## Aufgabe 1.1 Lösungsvorschlag

Zum Beenden der Kommunikation werden die Sockets mit der Methode **close()** geschlossen. Dabei ist die Reihenfolge egal. Unabhängig von den beiden Socket-Objekten kann das ServerSocket-Objekt ebenfalls mit der **close()**-Methode geschlossen werden.

## Aufgabe 1.2 Lösungsvorschlag

```
class Server {  
    public static void main( String[] args ) {  
        Server _this = new Server();  
        _this.listen();  
    }  
}
```

## Aufgabe 1.2 Lösungsvorschlag

```
public void listen() {
    ServerSocket listenSocket;
    Socket workSocket;
    BufferedWriter out;
    BufferedReader in;
    String msg;

    try { // etwas großzügig geklammert
        listenSocket = new ServerSocket( 9000 );
        workSocket = listenSocket.accept();

        out = new BufferedWriter(
            new OutputStreamWriter( workSocket.getOutputStream() ) );
        in = new BufferedReader(
            new InputStreamReader( workSocket.getInputStream() ) );

        msg = in.readLine();
        out.write( "Hallo " + msg + "\n" );
        out.flush();

        in.close();
        out.close();

        workSocket.close();
        listenSocket.close();
    } catch (IOException ex) {}
}
}
```

## Aufgabe 1.3 Lösungsvorschlag

```
class Client {  
    public static void main( String[] args ) {  
        Client _this = new Client();  
        _this.connect( args[0] );  
    }  
}
```

## Aufgabe 1.3 Lösungsvorschlag

```
public void connect( String sende ) {
    Socket workSocket;
    BufferedWriter out;
    BufferedReader in;
    String empfang;

    try {        // etwas großzügig geklammert
        workSocket = new Socket( "localhost", 9000 );

        out = new BufferedWriter(
            new OutputStreamWriter( workSocket.getOutputStream() ) );
        in = new BufferedReader(
            new InputStreamReader( workSocket.getInputStream() ) );

        out.write( sende + "\n" );
        out.flush();

        empfang = in.readLine();

        System.out.println( empfang );

        in.close();
        out.close();

        workSocket.close();
    } catch (IOException ex) {}
}
```

## Aufgabe 1.4 Lösungsvorschlag

Zum einen kann ohne Nebenläufigkeit immer nur ein Client auf einmal bedient werden. Zum anderen stellt sich auch folgende Problem. Will man den Server explizit beenden können, muss dieser Mechanismus in die Endlos-Schleife des Servers eingebaut werden. Eine elegante Lösung ist es, wenn der Server beendet wird, falls ein Client eine spezielle Nachricht schickt. Damit kann der Server aber von jedem der die Nachricht kennt gestoppt werden. Eine andere Möglichkeit ist das Testen auf das vorhanden sein/nicht vorhanden sein einer Datei.

## Aufgabe 1.4 Lösungsvorschlag

```
...
    listenSocket = new ServerSocket( 9000 );

+ boolean weiter = true;
+ while ( weiter ) {
    workSocket = listenSocket.accept();

    out = new BufferedWriter(
        new OutputStreamWriter( workSocket.getOutputStream() ) );
    in = new BufferedReader(
        new InputStreamReader( workSocket.getInputStream() ) );

    msg = in.readLine();
    out.write( "Hallo " + msg + "\n" );
    out.flush();

    in.close();
    out.close();

    workSocket.close();

+   weiter = !msg.equals("stopserver");
+ }

    listenSocket.close();
...

```

Die listen-Methode des Servers wird wie folgt modifiziert. Neue bzw. modifizierte Zeilen sind mit einem "+" markiert.

## Aufgabe 2 Lösungsvorschlag

Das typische Merkmal synchroner Nachrichten-Kommunikation ist das Blockieren beider Primitive (`send()` und `receive()`) bis der Kommunikationspartner die entsprechende "Gegenoperation" vollständig ausgeführt hat. Bei der asynchronen Kommunikation blockiert nur der Aufruf der `receive()`-Operation wohingegen nach einer `send()`-Operation immer sofort weitergearbeitet werden kann.

- Nachbildung synchroner Kommunikation: Zur Verfügung stehen die beiden asynchronen Kommunikationsprimitive `send()` mit nicht blockierendem Verhalten und `receive()` mit blockierendem Verhalten. Ziel ist es auch beim Senden einer Nachricht solange zu blockieren, bis der Empfänger die `receive()`-Operation abgeschlossen hat.

## Aufgabe 2 Lösungsvorschlag



Das linke Bild zeigt die asynchronen Primitive, das rechte Bild veranschaulicht, wie daraus ein synchrones Verhalten modelliert werden kann. Da `receive()` das einzig blockierende Primitiv ist, benützen wir es dazu, den Sender solange zu blockieren, bis der Empfänger mit einem `send()` bekannt gibt, dass seine `receive()`-Operation abgeschlossen ist.

Nachbildung asynchroner Kommunikation: Obwohl uns nur blockierende Kommunikationsprimitive zur Verfügung stehen, wollen wir ein asynchrones Verhalten modellieren. Dies gelingt dadurch, dass wir eine dritte Aktivität (Hilfsprozeß h) einführen, die eine gesendete Nachricht sofort übernimmt, antwortet und anschließend an den Empfänger weiterleitet.

## Aufgabe 2 Lösungsvorschlag

### Vor- und Nachteile

- Vorteile asynchroner Kommunikation: Der Sender verliert keine Zeit, da er nicht blockiert wird und kann diese Zeit unter Umständen sinnvoll nutzen.
- Nachteile asynchroner Kommunikation: Der Sender weiß nicht, ob die Nachricht bereits empfangen wurde. Oft ist dies eine wichtige Voraussetzung, dass der Sender weiterarbeiten kann. Bei der Realisierung tritt das Problem auf, wann die Variable (der Puffer), die die Nachricht enthält, wieder verfügbar ist. Entweder blockiert das Betriebssystem den Sender solange, bis die Nachricht aus dem Adressraum des Senders herauskopiert wurde oder der Sender darf die Nachricht erst verändern, wenn ihn das Betriebssystem informiert hat, dass die Nachricht bereits abgeschickt wurde.
- Vorteile synchroner Kommunikation: Der Sender ist sicher, dass die Nachricht empfangen wurde, wenn er entblockiert wird. Diese Semantik ist intuitiv und leicht verständlich.
- Nachteile synchroner Kommunikation: Das Blockieren kann zu einem Deadlock führen. Dies wird in der Realisierung oft durch Timeouts zu verhindern versucht.

## Aufgabe 3 Lösungsvorschlag

```
import java.io.RandomAccessFile;
import java.util.Date;

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

        try {
            RandomAccessFile file = new RandomAccessFile(args[0], "rw");

            long length=file.length();
            long pos=0;
            int key;
            int invertedKey;

            while(pos<length) {
                file.seek(pos);
                key = file.read();

                invertedKey = ~key;

                file.seek(pos); // erneut positionieren, da durch Lesen verändert
                file.write(invertedKey);

                pos += 2;
            }

            file.close();

        } catch (Exception ex) {ex.printStackTrace();}
    }
}
```

## „Entschlüsseltes“ Bild:

(In memoriam Bernd Pfarr: )



Dass ich nur einen einzigen Mitbewerber für die Stelle hatte, war schon mal gut. Noch besser war aber, dass dieser Mitbewerber offensichtlich nicht einmal Zeugnisse und Bewerbungsunterlagen dabei hatte! Und diese nachlässige Kleidung! Und diese schlaife Haltung! Kein Zweifel, der Job als Tanzbär war mir sicher!