Inhaltsverzeichnis

Strukturierte Programmierung

Installation und Vorbereitung

Für die Erstellung von Programmen benötigt man in erster Linie zwei Komponenten:

einen Editor zum Erstellen und Speichern des Quellcodes und einen Compiler.

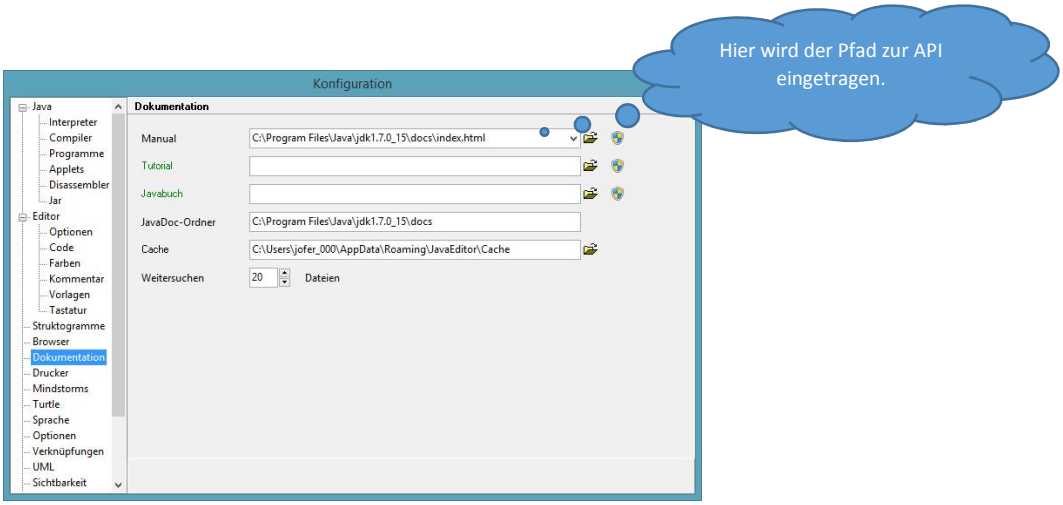
Statt des Editors kann man auch eine komplette Entwicklungsumgebung verwenden. Diese stellt alle möglichen Tools zur komfortablen Programmierung zur Verfügung. Die Gefahr hierbei ist, dass man sich in der Entwicklungsumgebung "verirrt". Eclipse und Netbeans sind solche kostenfreien Umgebungen und können gerne verwendet werden, finden aber im Kurs keinen Support!

Der JAVA-Editor vom Kollegen Gerhard Röhner bietet ausreichend viele Hilfen zum Programmieren. Codevervollständigung, Debugging, Klasseneditor, grafischer GUI-Designer sowie Stuktogrammer sind nur einige dieser Tools. Der Editor ist klein und sowohl unter 64-Bitsystemen als auch unter 32-Bit-Systemen zu verwenden. Andere Betriebssysteme außer Windows werden nicht unterstützt.

Zur ersten Installation empfiehlt sich folgende Vorgehensweise:

1. Installation des JDK[1]
2. Installation der API
3. Installation des JAVA-Editors
4. Konfiguration des JAVA-Editors

Der JAVA-Editor konfiguriert sich fast komplett von selbst. Lediglich die API (hierzu später mehr) muss manuell eingetragen werden. Ohne sie funktioniert die automatische Codevervollständigung nicht! Die Konfigurationseinträge findet man unter dem Menü "Fenster" → Konfiguration!

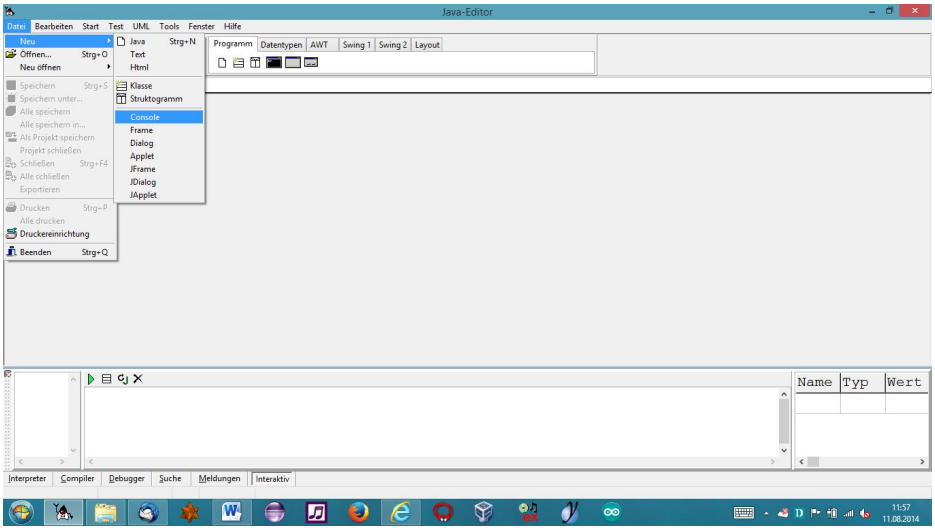


[1] Links zu den Installationsdateien werden hier nicht angegeben, da die Versionen schnell wechseln. Im Allgemeinen ergibt die Internetrecherche nach den Programmen schnelle Ergebnisse.

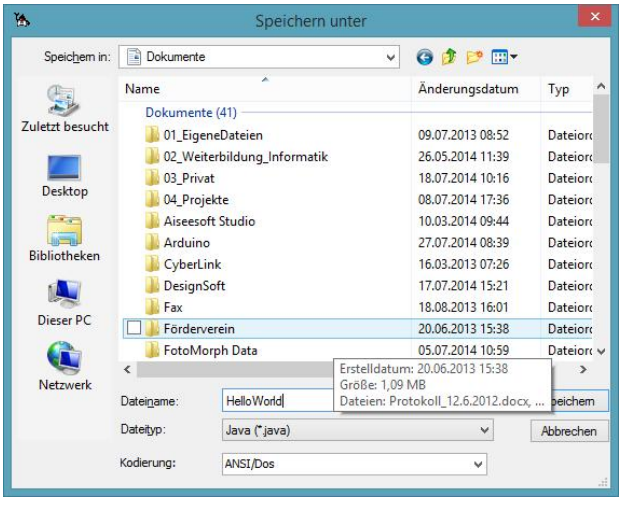
Anmerkung: Viele Konfigurationen und Installationen kann man auch direkt aus dem Editor starten. Da jedoch Firewalls und andere Sicherheitsfeatures in Netzwerken dies verhindern können sein hier diese Installationsreihenfolge beschrieben.

Ein erstes Programm:

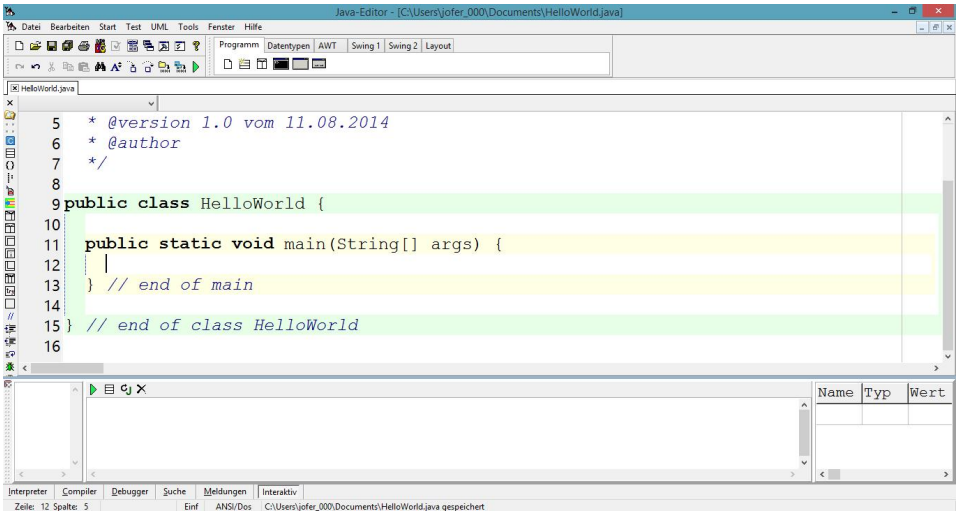
Nachdem alles installiert wurde kann es jetzt losgehen. Man wählt im Editor den Menüpunkt "Datei" → "Neu" → "Console":



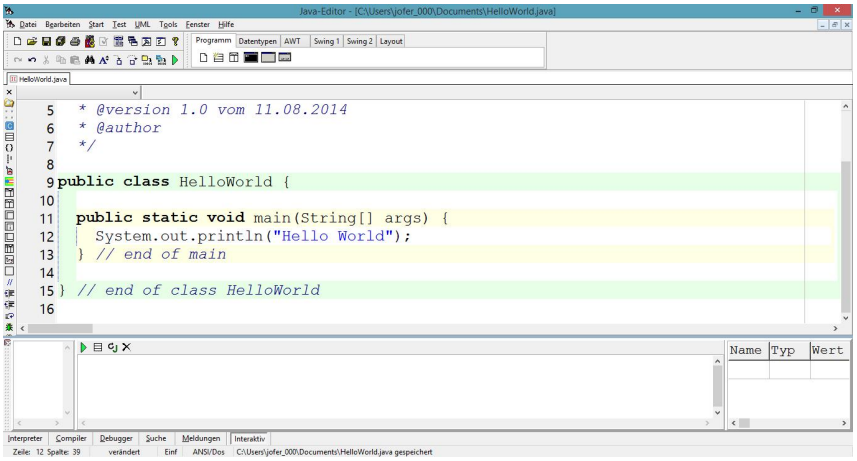
Der JAVA-Editor fordert nun zum Speichern der Datei auf: Hier wurde der Name HelloWorld gewählt!



Der JAVA-Editor erstellt nun eine Quelltextdatei:

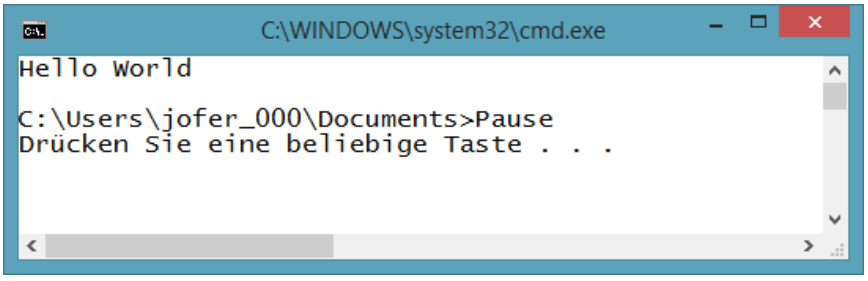


Geben Sie nun an der Cursormarke den Quelltext: System.out.println("Hello World"); ein:



Nun wird das Programm per Klick auf den grünen Pfeil kompiliert (übersetzt) und gestartet:

Das folgende Fenster sollte nun aufgehen:



Das erste Programm wurde erfolgreich geschrieben.

Variablen und Grundstrukturen

In einem Programm müssen Werte berechnet, zwischengespeichert, ausgegeben und wieder neu berechnet werden. Dies kann man nur, wenn man Variablen verwendet. Eine

Variable können Sie sich wie eine Schublade vorstellen, in die eine bestimmte Zahl oder aber auch ein Textzeichen abgelegt werden kann. Die Schublade benötigt noch einen Namen, damit man sie wiederfindet und die Größe der Schublade muss so gewählt werden, dass der zu speichernde Wert auch reinpasst. Möchten Sie beispielsweise die ganze Zahl 5 in der Schublade "zahl" abspeichern so funktioniert dies wie folgt:

```
int zahl = 5; //Variable zahl mit Wert 5 wird angelegt
```

Zur Erklärung:

Das Schlüsselwort int steht für eine Variable vom Typ integer. In diesem Variablentyp können negative und positive ganze Zahlen abgespeichert werden. Danach kommt der Name der Variablen, in diesem Falle "zahl" und die Zuweisung des Wertes 5. Hier sieht man auch wie ein Kommentar in den Quelltext eingefügt werden kann. Ein einzeiliger Kommentar wird mit den Zeichen "//" eingeleitet. Eine mehrzeiliger Kommentar wird über die Zeichen `/*` eingeleitet und über `*/` beendet. Kommentieren Sie Ihre Programme ausreichend um später einfacher Ihren eigenen Quelltext lesen zu können.

Die Zeile könnte man auch anders schreiben:

```
int zahl; //Deklaration

zahl = 5; //Initialisierung
```

In der ersten Zeile wird die Variable angelegt, man spricht von der Deklaration. In der zweiten Zeile wird der Variablen dann ein Wert zugewiesen, die Initialisierung. Übrigens werden Variablen vom Datentyp integer und auch einige andere Datentypen automatisch mit dem Wert 0 initialisiert.

Variablentypen

Alle Variablentypen aufzuzählen wäre wohl übertrieben. Hier eine Auflistung der gängigsten Typen, die Sie zur Programmierung benötigen:

Variablentyp	**Bedeutung**	**Beispiel:**

`boolean`	Logischer Wert, der entweder den Zustand true oder false enthält	`boolean b = true;` `b = false;`
`char`	Unicode Zeichen	`char c = 's';`
`byte`	Alle ganzzahligen Werte von –128 bis 127	`byte b = 10;`
`short`	Alle ganzzahligen Werte von –32768 bis 32767	`short s = -20;`
`int`	Alle ganzzahligen Werte von –2147483648 bis 2147483647	`int i = 1;`
`long`	Alle ganzzahligen Werte im 64bit – Bereich	`long l = -11000029;`
`float`	Alle Fließkommazahlen im 32bit – Bereich	`float f = 453627.768;`
`double`	Alle Fließkommazahlen im 64bit - Bereich	`double d = 34875637.3643645;`

Berechnungen

Mathematische Standardoperationen

Um Datenverarbeitung überhaupt erst durchführen zu können, bedarf es vieler mathematischer Berechnungen, die unter anderem mit den folgenden Operatoren durchgeführt werden können:

Operator	**Bedeutung**	**Beispiel**
+	Addition	i = x+2;
-	Subtraktion	i = z –3;
*	Multiplikation	x=5*2;
/	Division	e=9/4;
%	Ganzzahliger Rest einer Division	i=9%7;

++	Inkrementieren	i=0;
		i++;
		i hat hiernach den Wert 1.

--	Dekrementieren	i=0;

		i--; i hat hiernach den Wert –1.

Logische Standardoperationen

Um Vergleiche oder logische Verknüpfungen durchführen zu können, bedarf es einiger Operatoren, die in der folgenden Tabelle aufgeführt sind (i ist hierbei eine **boolsche** Variable, die den Wert true oder false annehmen kann).

Operator	Bedeutung	Beispiel
<	ist kleiner als	i=5<6; i hat hiernach den Wert true.
>	ist größer als	i = 5>6; i hat hiernach den Wert false.
<=	ist kleiner gleich als	i = 5<=5; i hat hiernach den Wert true.
>=	ist größer gleich als	i = 5 >=5; i hat hiernach den Wert true.
==	ist gleich	i = 5==6; i hat hiernach den Wert false.
!=	ist ungleich	i = 5 !=6; i hat hiernach den Wert true.
&&	logisches UND	i = ((5>6)&&(6>5)); i hat hiernach den Wert false.
||	logisches ODER	i = ((5>6)||(6>5)); i hat hiernach den Wert true.
&	bitweise UND-Verknüpfung	i = (127 & 4); i hat hiernach den Wert 4
|	bitweise ODER-Verknüpfung	i = (127 | 128); i hat hiernach den Wert 128

Klammersetzung:

Es ist zu bemerken, dass verschiedene Operatoren eine unterschiedliche Wertigkeit haben. Das heißt, manche Operatoren werden in einer Codezeile vor anderen ausgeführt. Um dies zu vermeiden, sind Klammern so zu setzen, dass einerseits die Befehlszeile ihren Zweck erfüllt und andererseits der Programmcode für den Programmierer lesbar bleibt.

Eingaben von Daten über die Tastatur bei Konsolenprogrammen

Das Einlesen von Daten von der Tastatur ist in Java nicht ganz trivial. Um die Plattformunabhängigkeit zu erhalten, wird normalerweise über einen Scanner gearbeitet. Der Scanner wird später noch näher behandelt. Um Ihnen die Arbeit zu erleichtern, wurde eine eigene Konsolenklasse entwickelt. Diese Klasse gibt es für 32-Bit-Betriebssysteme sowie für 64-Bit-Betriebssysteme.

Achtung:

Einige der Funktionen der Konsolenklasse funktionieren nicht für 64-Bit-Betriebssysteme. Dies sind: `setColors()`, `gotoXY()` und `cls()`!

Vorbereitungen:

Kopieren Sie die Dateien Konsole.java, JKonsole.java sowie JKonsole.dll in das Verzeichnis, in dem auch Ihre Programmdateien liegen. Bitte verwenden Sie nicht die .class-Dateien.

Mittels der Klasse Konsole.java kann der Wert von Variablen leicht eingelesen werden.

Beispiel 1:

```
int i;
i = Konsole.readInt();
```

Erklärung:

i wird mit dem Wert belegt, den der Anwender eingegeben hat.

Beispiel 2:

```
double u;
u = Konsole.reaDouble();
```

Erklärung:

u wird mit dem Wert belegt, den der Anwender eingegeben hat.

Beispiel 3:

```
String text;
text = Konsole.readString();
```

Erklärung:

text wird mit dem Text belegt, den der Anwender eingegeben hat.

Beispiel 4 (nur 32-Bit):

Konsole.cls();

Erklärung:

Der Konsolenbildinhalt wird gelöscht.

Beispiel 5 (nur 32-Bit):

Konsole.gotoXY(4,10);

Erklärung:

Die Cursorposition wird an die Stelle 4, 10 gesetzt.

Die bedingte Entscheidung (if else)

Sollen in einem Programm bedingte Entscheidungen getroffen werden, so ist der if - else - Befehl hilfreich.

Der grundsätzliche Aufbau des if - else - Befehls:

```
if (Bedingung)
 {
     Befehlsblock1;
 }
else
 {
     Befehlsblock2;
 }
```

Erklärung:

Wenn die Bedingung, die immer eine logische Operation darstellt, wahr ist, dann wird Befehlsblock1 ausgeführt, ansonsten wird Befehlsblock2 ausgeführt.

Anmerkung:

Der else - Befehl muss nicht implementiert werden. Beispiel:

```
if (Bedingung)
```

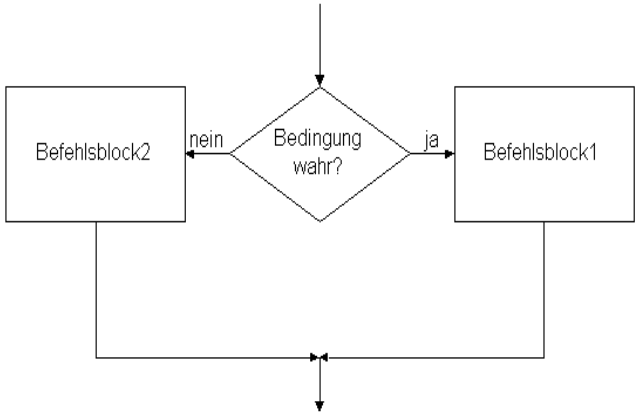
```
{
      Befehlsblock;
}
```

Erklärung:

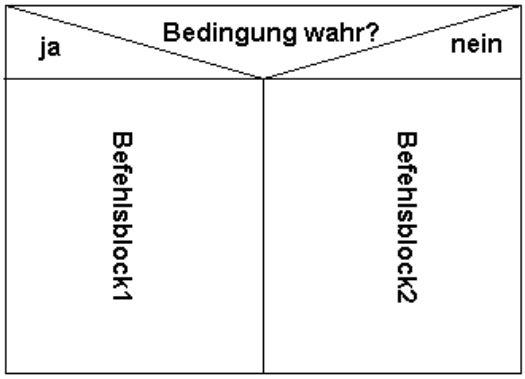
Wenn die Bedingung wahr ist, wird der Befehlsblock ausgeführt. Ansonsten wird direkt das Programm fortgesetzt.

Grafische Darstellung:

Programmablaufplan



Struktogramm



Die Mehrfachentscheidung (switch case)

Die Mehrfachentscheidung (`switch-case`) hilf dabei Übersicht zu behalten, wenn nach einer Abfrage mehrere Möglichkeiten gegeben sind. Beispielsweise hat ein Menü drei verschiedene Auswahlmöglichkeiten. Außerdem ist auch noch eine Falscheingabe möglich. Will man dies mit bedingten Entscheidungen (`if-else`) realisieren, so wird der Quelltext doch unübersichtlich:

```
int eingabe = Konsole.readInt();
if (eingabe == 1){
      mache etwas;
}else if (eingabe == 2){
            mache etwas anderes;
}else if (eingabe == 3){
                  mache etwas ganz anderes;
}else{
                        gebe aus „Falsche Eingabe“;
}
```

Hier hilf die `switch-case`-Struktur:

```
int eingabe = Konsole.readInt();
switch(eingabe){
  case 1: mache etwas; break;
  case 2: mache etwas anderes; break;
  case 3: mache etwas ganz anderes; break;
  default: gebe aus: „Falsche Eingabe"; break;
}
```

Zur Erklärung:

Mittels `switch` wird die Mehrfachentscheidung eingeleitet. In den runden Klammern steht die Variable, nach der später Entscheidungen getroffen werden. Diese Variable ist vom Typ `int` oder seit Java 7 auch vom Typ `String`.

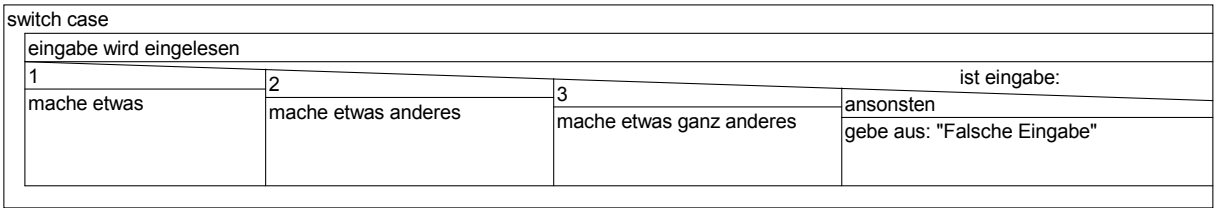
Anschließend werden die gewünschten Fälle durch `case` aufgelistet (beispielsweise wird `case` 1 dann ausgeführt, wenn in der Variablen `eingabe` eine 1 steht).

Trifft keiner der aufgeführten Fälle zu, so wird der Teil ausgeführt, der hinter `default` steht. `default` ist allerdings nicht verpflichtend.

Wichtig:

Wird ein Fall ausgeführt, so werden alle darauf folgenden Fälle auch ausgeführt. Daher steht hinter jedem Fall das Schlüsselwort `break`, welches den `switch`-Block sofort beendet.

Switch-Case-Struktur im Struktogramm:

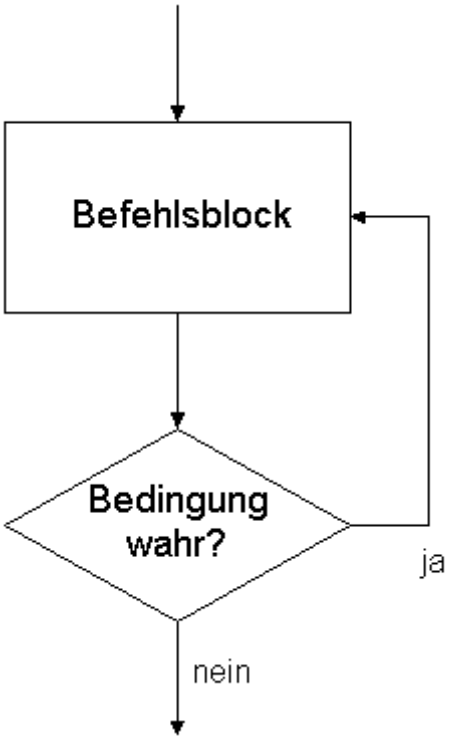


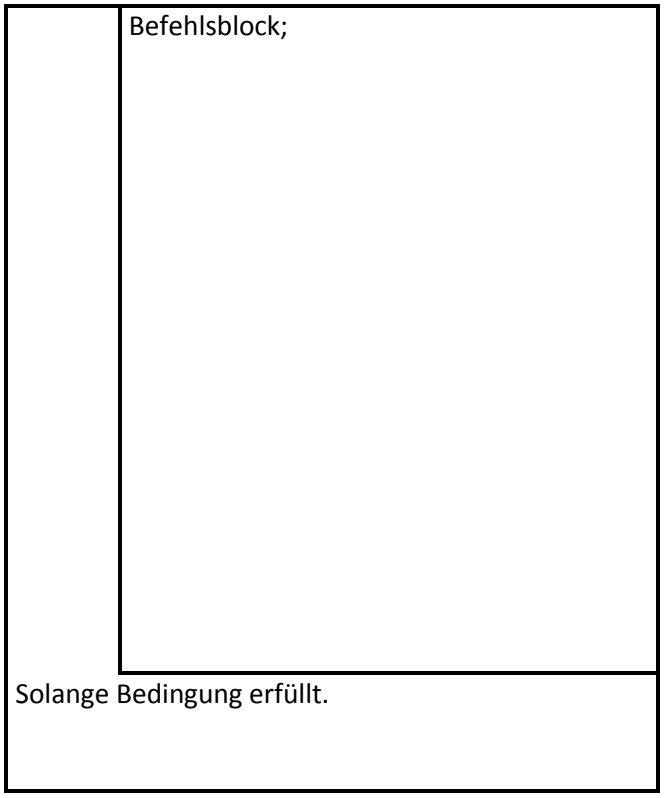
}

Die fußgesteuerte Schleife (do – while – Schleife)

Die Schleife hat in den grafischen Darstellungen folgendes Aussehen:

Programmablaufplan **Struktogramm**





Die fußgesteuerte Schleife wird in jedem Fall einmal durchlaufen und erst am Ende (am Fuß) der Schleife auf ihre Bedingung überprüft:

Die grundlegende Form einer do - while - Schleife:

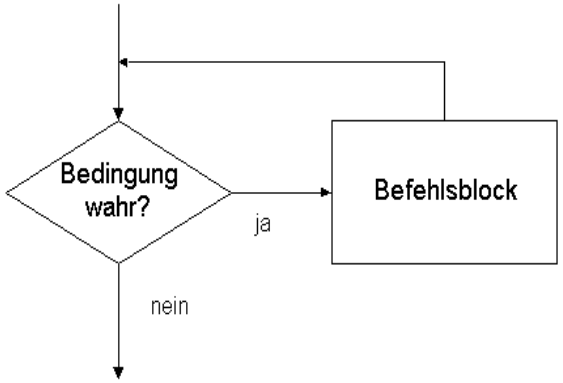
```
do
 {
  Befehlsblock;
 }
while (Bedingung);
```

Da die Bedingung am Ende der Schleife das Verlassen der Schleife bewirkt, heißt sie auch Austrittsbedingung.

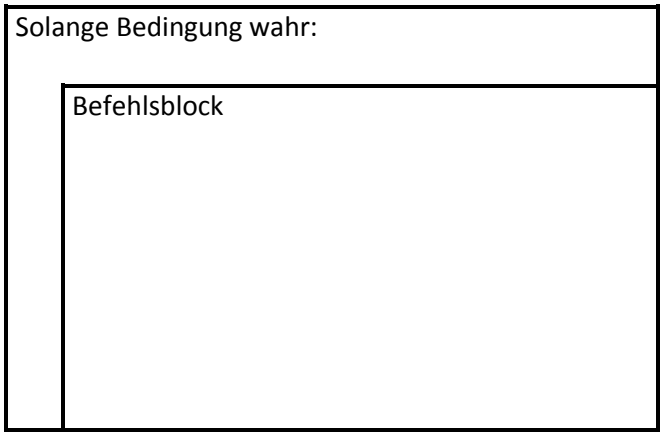
Die kopfgesteuerte Schleife (while – Schleife)

Die Schleife hat in den grafischen Darstellungen folgendes Aussehen:

Programmablaufplan



Struktogramm



Die grundlegende Form einer while - Schleife hat in JAVA folgende Darstellung

```
while (Bedingung)
{
      Befehlsblock;
}
```

Bei dieser Schleife wird zuerst die Bedingung überprüft. Falls die Bedingung wahr ist, wird der Befehlsblock ausgeführt. Die Schleife heißt auch kopfgesteuerte Schleife, da sie ihre Bedingung am Kopf der Schleife abfragt. Es ist also durchaus möglich, dass der Befehlsblock der Schleife (der Schleifeninhalt) in einem Programm nicht ausgeführt wird. Man nennt die Bedingung der Schleife auch Eintrittsbedingung.

Die Zählschleife (for – Schleife)

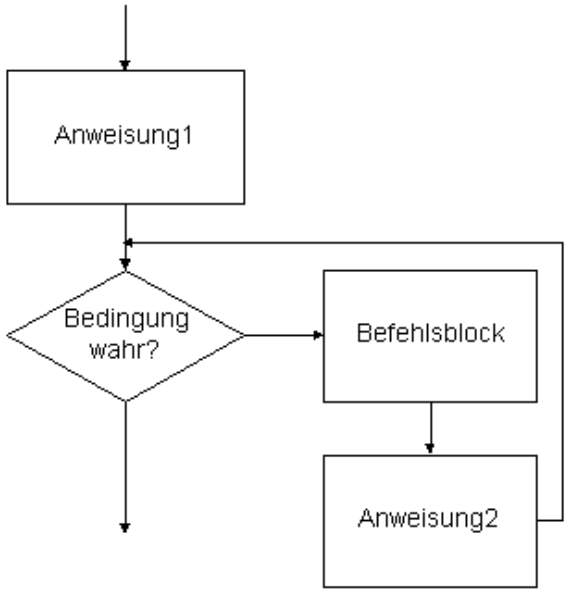
Um einen Befehlsblock n-mal durchlaufen zu lassen, ist es nicht notwendig, den Befehlsblock n - mal in den Programmcode zu schreiben. Man kann hierfür Schleifen benutzen. Ist n ganzzahlig, so eignet sich besonders die for - Schleife.

Die grundlegende Form einer for - Schleife hat in Java folgendes Aussehen:

```
for (Anweisung1; Bedingung;Anweisung2)
```

```
{

Befehlsblock;

}
```

Im Programmablaufplan stellt sich die for - Schleife wie folgt dar:



Im Struktogramm wird die Schleife wie folgt dargestellt:

for Anweisung1 to Bedingung

Befehlsblock;

Anweisung2;

Die Schleife arbeitet wie folgt:

- Anweisung1 wird nur einmal ausgeführt.
- Ist die Bedingung wahr, wird der Befehlsblock ausgeführt. Anschließend wird Anweisung2 ausgeführt. Sprung zu 2.11
- Ist die Bedingung nicht wahr, wird das Programm einfach fortgesetzt.

Anmerkung: Eine weitere Form der Zählschleife wird später behandelt.

Break / Continue

Um in Schleifen flexibler navigieren zu können existieren die beiden Schlüsselwörter `break` und `continue`.

`break`:

Das Schlüsselwort `break` bricht eine Struktur sofort ab. Beispiel:

```
..
for (int i = 0; i < 10; i++)
{
      System.out.println (i);
      if ( i== 5)
      break;
}
```

Das Programm gibt die Zahlen 0 bis 5 aus und verlässt dann die Zählschleife.

`continue`:

Das Schlüsselwort `continue` sorgt dafür, dass eine Schleife sofort wieder an den Anfang der Schleife springt. Beispiel:

```
for (int i = 0; i < 10; i++)
{
      if ( i> 5)
      continue;
      System.out.println (i);
}
```

Das Programm gibt nur die Zahlen von 0 bis 5 aus.

Arrays / Felder

In Feldern können mehrere Werte vom gleichen Datentyp abgespeichert werden.

Ein Feld, hier beispielsweise von 1000 Integerzahlen, wird wie folgt angelegt:

```
int feld [ ] = new int [ 1000 ];
```

Anschließend kann man über den Index in den eckigen Klammern auf die einzelnen Feldelemente zugreifen.

```
feld [ 9 ] = 300;
```

Die Länge eines Feldes bekommt man über folgenden Ausdruck:

```
System.out.println (feld.length); //hat hier 1000 als Ausgabe zur Folge
```

Der Index beginnt immer mit der Zahl 0. Somit hat das letzte Element des Feldes den Index `feld.length – 1 !`

Das Feld kann auch direkt beim Anlegen mit Werten gefüllt werden:

```
int [] feld = {1,2,3,4,5,6,7,8,9,10 };
```

Die for-Schleife eignet sich ideal zum Füllen eines Arrays beispielsweise mit Messwerten.

Im nachfolgenden Beispielprogramm wird ein Feld aus 100 Integerzahlen mit Zufallswerten im Bereich 0 bis 100 gefüllt. Die Zahlen werden auf dem Bildschirm ausgegeben. Anschließend werden das größte, das kleinsten Element und das arithmetische Mittel aller Zahlen berechnet und ausgegeben.

Kommentierter Quellcode der Klasse Felder.java

```
public class Felder
{
 public static void main (String args[])
 {
       //Ein Feld für 100 Integerzahlen wird erzeugt
       int feld[] = new int [100];

       //Deklaration der notwendigen Variablen
       int summe = 0, kleinstes = 1000, groesstes = 0;
       double schnitt = 0;

       //Das Feld wird mit Zahlen gefüllt
       for (int i = 0; i < 100; i ++)
       {
              //Die Funktion Math.random() liefert eine Zufalls-
```

```
            //zahl im Bereich 0 bis 1. Diese Zahl wird mit 100
            //multipliziert. Das ergebnis ist eine Kommazahl
            //im Bereich 0 bis 100. Diese Zahl wird explizit
            //in eine Integerzahl umgewandelt. Alle Nachkomma-
            //stellen werden abgeschnitten.
            feld[i] = (int)(100*Math.random ());
            //Ausgabe des Feldes
            System.out.print (feld[i] + "\t");
            //Bei jeder Zahl, die durch 10 teilbar ist wird
            //ein Zeilenumbruch vorgenommen.
            if (((i+1) % 10)== 0)
            {
                  System.out.println ();
            }
      }
      for (int i = 0; i < 100; i++)
      {
            //Elemente werden aufsummiert
            summe = summe + feld[i];
            //Das größte Element wird ermittelt
            if (feld [i] > groesstes)
                  groesstes = feld[i];
            //Das kleinste Element wird ermittelt
            if (feld [i] < kleinstes)
                  kleinstes = feld[i];
      }
      //Ausgabe der errechneten Ergebnisse
      System.out.println ("Groesstes Element: " + groesstes);
      System.out.println ("Kleinstes Element: " + kleinstes);
      //Der Schnitt wird errechnet. Da die Division durch 100
      //eine Kommazahl ergibt, summe aber vom Typ Integer ist
      //muss das Ergebnis der Rechnung explizit in eine Zahl
      //vom Typ double umgewandelt werden.
      schnitt = (double)summe/100;
      System.out.println ("Schnitt: " + schnitt);
 }//end of main
}//end of class
```

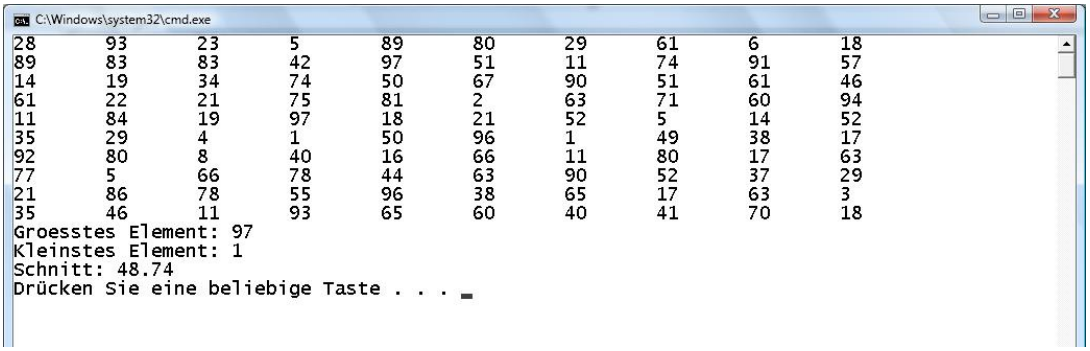


Abb.: Ausgabe des Programms mit Feld

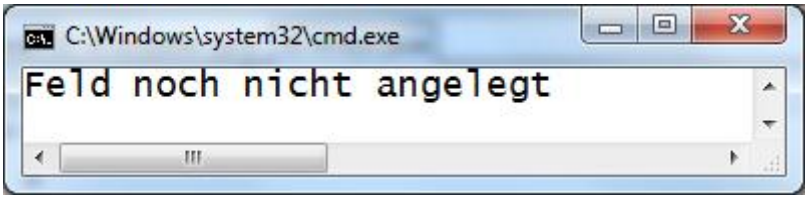
Existiert ein Feld?

Es kann überprüft werden, ob ein Feld existiert oder nicht:

```
int [] feld = null;
if (feld == null){
  System.out.println("Feld noch nicht angelegt");
}
```

Das Feld wurde deklariert, existiert aber im Speicher noch nicht.

Bildschirmausgabe:



Mehrdimensionale Felder

Felder können auch mehrdimensional angelegt werden. Beispielsweise macht es Sinn, bei einem Schachprogramm das Spielbrett als zweidimensionales Feld anzulegen:

```
int spielbrett [][] = new int [8][8];
```

Die Anzahl der Elemente bekommt man hier wie folgt heraus:

```
int erstedimension = spielbrett.length;
int zweiteDimension = spielbrett[0].length;
```

Übergabe von Arrays an Funktionen

Felder können auch an Funktionen / Methoden übergeben werden. Hierzu muss im Kopf der entsprechenden Funktion / Methode der Übergabeparameter als Array gekennzeichnet werden. Im

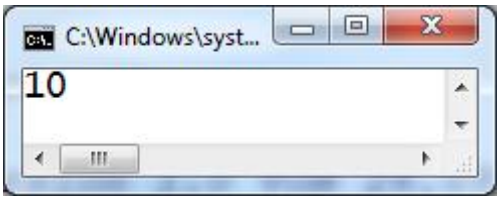
nachfolgenden Programmbeispiel wird einer Funktion ein Array übergeben und das größte Element in diesem Feld zurückgegeben.

```
public static int groesstesElement(int [] feld){
  int groesstes=feld[0];
  for (int i = 1; i < feld.length;i++){
    if(feld[i]>groesstes){
      groesstes = feld[i];
    }
  }
  return groesstes;
}
```

Die Funktion wird nun aufgerufen, indem man den Namen des Arrays ohne! die eckigen Klammern der Funktion übergibt:

```
public static void main(String[] args) {
  int [] feld = {1,2,3,4,5,6,7,8,9,10 };
  int wert = groesstesElement(feld);
  System.out.println (wert);
}
```

Das Programm hat die folgende Bildschirmausgabe zur Folge:



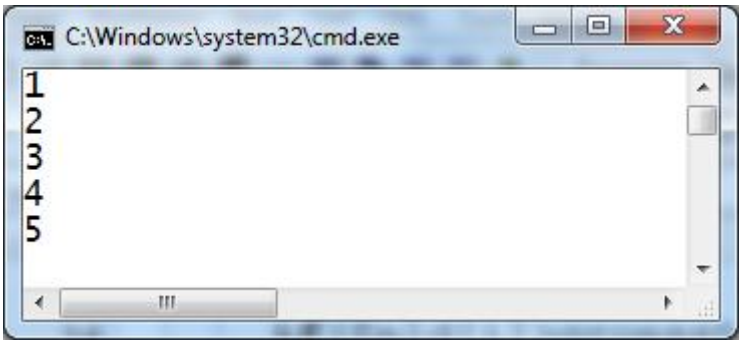
Rückgabe von einem Array von Funktionen

Im Prinzip funktioniert die Rückgabe von Arrays adäquat der Übergabe.

```
public static int [] subArray(int feld[], int letzterIndexSubArray){
    int[] rueckgabe = null;
    if (!(letzterIndexSubArray > feld.length-1)){
```

```
      rueckgabe = new int[letzterIndexSubArray+1];
      for (int i = 0; i <= letzterIndexSubArray;i++){
        rueckgabe[i] = feld[i];
      }
    }
    return rueckgabe;
  }

  public static void main(String[] args) {
    int [] feld = {1,2,3,4,5,6,7,8,9,10 };
    int [] subFeld = subArray(feld,4);
    for(int i = 0; i < subFeld.length;i++){
      System.out.println(subFeld[i]);
    }
  }
```

Das Programm extrahiert aus einem übergebenen Array ein Teilarray. Die folgende Bildschirmausgabe wird angezeigt:



Funktionen

Beispiel:

In einem Programm werden vier Zahlen vom Zufallsgenerator erzeugt, dann bestimmt das Programm die größte der ersten beiden Zahlen und die größte der letzten beiden Zahlen:

```
public class Zahlen1
{
      public static void main (String args[])
      {
            int zahl1 = (int)(100*Math.random());
```

```
            int zahl2 = (int)(100*Math.random());
            int zahl3 = (int)(100*Math.random());
            int zahl4 = (int)(100*Math.random());
            if (zahl1 > zahl2)
            {
                  System.out.println (zahl1 + " ist größer");
            }
            else
            {
                  System.out.println (zahl2 + " ist größer");
            }
            if (zahl3 > zahl4)
            {
                  System.out.println (zahl3 + " ist größer");
            }
            else
            {
                  System.out.println (zahl4 + "ist größer");
            }
      }
}
```

Man sieht, dass sich hier Teile des Programms wiederholen. Hier bietet es sich an mit Funktionen zu arbeiten. Eine Funktion kann man sich wie eine Werkstatt vorstellen. Man bringt etwas oder auch mehrere Teile in die Werkstatt (die natürlich einen Namen besitzt), diese bearbeitet das und man holt gegebenenfalls auch wieder etwas ab.

Genauso sind Funktionen aufgebaut:

Funktionen haben einen Namen, man kann (muss aber nicht unbedingt) ihnen etwas übergeben (Übergabeparameter) und bekommt etwas zurück (Rückgabeparameter).

Das obige Programm soll mit einer Funktionen umgesetzt werden:

```
public class Zahlen2
{
```

Rückgabeparameter

```
      public static int maximum (int zahla, int zahlb)
      {
```

Übergabeparameter (können mehrere sein)

```
        if (zahla > zahlb)
        return zahla;
        else
        return zahlb;
    }
```

Funktionsname

```
    public static void main (String args[])
    {
        int zahl1 = (int)(100*Math.random());
        int zahl2 = (int)(100*Math.random());
        int zahl3 = (int)(100*Math.random());
        int zahl4 = (int)(100*Math.random());
        //Die Funktion maximum wird aufgerufen, ihr werden zwei Zahlen
        //übergeben
        int temp = maximum (zahl1,zahl2);
        System.out.println (temp + " ist größer");
        //Die Funktion maximum wird aufgerufen, ihr werden zwei Zahlen
        //übergeben
        temp = maximum (zahl3,zahl4);
        System.out.println (temp + " ist größer");
    }
}
```

Erklärung:

Eine Funktion beginnt immer mit den Schlüsselwörtern `public static`(die Erklärung hierzu erfolgt erst im Kurs "Objektorientierte Programmierung"). Anschließend kommt der Datentyp, den die Funktion zurückgibt, im obigen Beispiel `int` (Anmerkung: Gibt eine Funktion nichts zurück, so steht hier das Schlüsselwort `void`! Es kann nur ein Typ zurückgegeben werden). Als nächstes kommt der Funktionsname, im Beispiel `maximum`. Es folgen zwei runde Klammern. In diesen Klammern stehen die Übergabeparameter. Dies können beliebig viele sein, soll nichts übergeben werden, so bleiben die runden Klammern leer. Der Programmcode der Funktion wird in geschweiften Klammern eingefasst. Innerhalb der Funktion können Werte mittels des Schlüsselwortes `return` zurückgegeben werden.

Übergabe und Rückgabe von Feldern an Funktionen

Auch Arrays können an Funktionen übergeben und auch zurückgegeben werden. Im Funktionskopf stehen dann beim Datentyp leere eckige Klammern. Im nachfolgenden Beispiel wird in einer Funktion der Inhalt eines Arrays auf den Bildschirm ausgegeben und in einer anderen Funktion ein mit Zufallszahlen von 0 bis `max` gefülltes Array mit `groesse` Elementen erzeugt:

```
public class Zahlen3
{
	public static void feldAusgeben (int array[])
	{
		for (int i = 0; i < array.length;i++)
		System.out.println (array[i]);
	}

	public static int[] feldErzeugen(int groesse, int max)
	{
		int feld[] = new int [groesse];
		for (int i = 0; i < array.length;i++)
		{
			feld[i] = (int)(max*Math.random());
		}
		return feld;
	}

	public static void main (String args[])
	{
		int feld[] = feldErzeugen (20;100);
		feldAusgeben(feld);
	}
}
```

Rekursion

Unter einer Rekursion versteht man den Aufruf einer Funktion aus sich selbst heraus. Dies bietet sich dann an, wenn es sich um Berechnungen oder Algorithmen handelt, die wiederkehrende Vorschriften beinhalten.

Beispiele:

Fakultätsberechnung:

Die Fakultät einer Zahl n berechnet sich aus n mal der Fakultät der Zahl n-1: $n! = n \cdot (n-1)!$, wobei die Fakultät $1! = 1$ definiert ist.

Zur Programmierung:

Die Programmierung von Rekursionen ist immer sehr ähnlich: Zuerst prüft man, ob die gegebene Bedingung eingetreten ist (hier: ist n = 1). Ist dies der Fall so gibt man den definierten Wert zurück. Ist dies nicht der Fall, so gibt man die gegebene Zusammensetzungsvorschrift zurück:

```
public static int fakultaet(int n){
```

```
    if(n == 1)
    return 1;
    else
    return n*fakultaet(n-1);
  }
```

Potenzierung einer Zahl

Die Zusammensetzungsvorschrift lautet: $x^n = x \cdot x^{n-1}; ist\ n = 1, so\ ergibt\ x^1 = x$

Quelltext:

```
public static int xHochN(int x, int n){
    if(n == 1)
      return x;
    else
      return x*xHochN(x,n-1);
  }
```

Auch hier wieder: zuerst den definierten Wert abfragen und eventuell zurückgeben, ist dies nicht der Fall, so wird die Zusammensetzungsvorschrift zurückgegeben.

Fibunaccizahlen

Die n-te Fibunaccizahl ergibt sich wie folgt: $n = 1\ oder\ n = 2 \rightarrow fibu\ nacci(n) = 1; ansonsten: fibu\ nacci(n) = fibu\ nacci(n-1) + fibu\ nacci(n-2)$

Quelltext:

```
  public static int fibunacci(int n){
    if (n==1||n==2)
      return 1;
    else
      return fibunacci(n-1)+fibunacci(n-2);
  }
```

Größter gemeinsamer Teiler

Der größte gemeinsame Teiler der Zahlen n und m ist wie folgt definiert:

$$ist\ n = m \rightarrow ggT(n,m) = n$$

$$ist\ n > m \rightarrow ggT(n,m) = ggT(n-m,m)$$

$$ist\ n < m \rightarrow ggT(n,m) = ggT(n,m-n)$$

Quelltext:

```
public static int ggT(int n,int m){
    if (n==m) {
      return n;
    }
    if (n>m) {
      return ggT(n-m,m);
    }
    if (n<m) {
      return ggT(n,m-n);
    } // end of if
    return 0;
  }
```

Sortieralgorithmen

Um Objekte oder aber Variablen in Listen bzw. Arrays leichter und damit schneller finden zu können, müssen diese sortiert werden. Es ist leicht einzusehen, dass beispielsweise in einer Bank das Konto mit der Kontonummer 1000 wesentlich schneller zu finden ist, wenn alle Konten in einer nach Kontonummern sortierten Liste stehen.

Zum Sortieren von Listen oder Arrays stehen verschiedene Sortieralgorithmen zur Verfügung. Hier sollen die folgenden behandelt werden:

1) Bubble-Sort
2) Insertion-Sort (Sortieren durch Einfügen)
3) Sekection-Sort (Sortieren durch Auswahl)
4) Quick-Sort (Teile und Herrsche)

1. Bubble-Sort

Bei diesem Algorithmus wird eine Liste von Werten von links nach rechts (natürlich funktioniert der Algorithmus auch umgekehrt) durchlaufen. Jeweils zwei Werte werden verglichen. Stehen Diese nicht in der gewünschten Reihenfolge, so werden sie vertauscht, beispielsweise durch einen Ringtausch. Wurde die Liste einmal komplett durchlaufen, beginnt der Vorgang von neuem, so lange, bis keine Werte mehr getauscht wurden.

Beispiel:

Gegeben ist eine Liste von 5 Zahlen, die der Größe nach beginnend mit dem kleinsten Element sortiert werden sollen:

5	1	3	2	4

Im ersten Durchlauf passiert folgendes:

5 und 1 werden verglichen, da die Reihenfolge nicht stimmt werden die beiden Elemente vertausch:

1	5	3	2	4

5 und 3 werden verglichen, da die Reihenfolge nicht stimmt werden die beiden Element vertausch:

1	3	5	2	4

5 und 2 werden verglichen und getauscht:

1	3	2	5	4

5 und 4 werden verglichen und getauscht:

1	3	2	4	5

Man sieht, dass das größte Element ganz nach rechts gewandert ist (ähnlich der größten Luftblase im Wasser, daher der Name: Bubble-Sort).

Im zweiten Durchlauf passiert folgendes:

1	3	2	4	5

1 und 3 werden verglichen, die Reihenfolge stimmt, also kein Tausch:

1	3	2	4	5

3 und 2 werden verglichen, es wird getauscht:

1	2	3	4	5

3 und 4 werden verglichen, es wird nicht getauscht.

4 und 5 werden verglichen, es wird nicht getauscht.

Im dritten Durchlauf werden noch einmal alle benachbarten Elemente verglichen. Da aber kein Tausch mehr vorkommt, bricht der Algorithmus ab, das Feld ist sortiert.

2. Insertion-Sort

Dieser Algorithmus funktioniert nach dem Prinzip so, wie Spielkarten auf der Hand einsortiert werden. Zuerst nimmt man die erste Karte auf. die zweite Karte wird dann direkt passend auf der Hand einsortiert usw.

Mittels einer ArrayList kann man diesen Algorithmus exakt so nachbilden, bei der Verwendung eines Arrays ist ein Umkopieren der Elemente notwendig:

Beispiel:

Gegeben ist wieder das oben verwendete Array von Zahlen.

5	1	3	2	4

Die erste Zahl ist die 5 und wird in ein neues Array eingefügt:

5				

Nun kommt die 1, sie ist kleiner als 5 und wird links von der 5 einsortiert, die 5 muss dann um einen Platz nach rechts verschoben werden:

1	5			

Als nächstes ist die 3 an der Reihe, sie muss links von der 5 und rechts von der 1 einsortiert werden, die 5 muss also wieder umkopiert werden:

1	3	5		

Die 2 wird links von der 3 einsortiert, damit müssen 3 und 5 umkopiert werden:

1	2	3	5	

Die 4 wird als letztes links von der 5 einsortiert, 5 muss also umkopiert werden:

1	2	3	4	5

Damit ist das Feld sortiert.

Der Algorithmus funktioniert mit einem neuen Array, aber genauso mit dem schon vorhandenen Array, was Speicherplatz schont, jedoch mehr Aufwand beim Kopieren erfordert.

3. Selection-Sort

Beim Selection-Sort wird das vorhanden Feld durchlaufen. Das kleinste Element (bei anderer Sortierreihenfolge das größte Element) wird identifiziert und sein Index gespeichert. Ist das Feld

durchlaufen, wird das kleinste Element mit dem ersten Element getauscht. Nun wird das Feld wieder durchlaufen, jedoch erst ab dem zweiten Element (im ersten Element steht ja schon das Minimum). Es wird wieder das kleinste Element gesucht und anschließend mit dem zweiten Element getauscht, usw.

Beispiel:

Gegeben ist wieder das oben verwendete Array von Zahlen.

5	1	3	2	4

Nach dem ersten Durchlauf wird das kleinste Element, die 1 mit der 5 getauscht:

1	5	3	2	4

Nach dem zweiten Durchlauf wird das übriggebliebene Feld durchsucht. Das nun kleinste Element, die 2, wird mit der 5, dem zweiten Element getauscht:

1	2	3	5	4

Im dritten Durchlauf wird die 3 als kleinstes Element identifiziert, bleibt aber an Ihrem Platz stehen:

1	2	3	5	4

Im letzten Durchlauf nimmt die 4 den Platz der 5 ein:

1	2	3	4	5

4. Quicksort

Beim Quicksort wird ein Feld in eine linke und eine rechte hälft unterteilt. Ein Element (das Pivotelement) stellt die Grenze zwischen linker und rechter Hälfte dar. Alle Elemente, die kleiner als das Pivotelement sind, werden in die linke Hälfte kopiert, alle anderen in die rechte Hälfte. Nun wird mit der linken Hälfte der gleiche Vorgang gestartet, ein Pivotelement wird bestimmt und die linke Hälfte wird in eine neue linke und neue rechte Hälfte geteilt usw. Ist die erste linke Hälfte sortiert, wird mit der ersten rechten Seite identisch vorgegangen.

Da die Reihenfolge: Pivotelement bestimmt→Feld teilen→umkopieren identisch für alle Teilfelder ist, kann man den Algorithmus rekursiv programmieren.

Beispiel:

Gegeben ist ein Array von Zahlen:

5	7	1	6	3	9	2	10	4	8

Als erstes Pivotelement wird dir 3 gewählt, alle Elemente kleiner der 3 werden nach links kopiert, alle anderen nach rechts:

2	1	3	7	6	9	5	10	4	8

Im linken übriggebliebenen Feld wird die 2 als Pivotelement gewählt, und wieder kopiert:

1	2	3	7	6	9	5	10	4	8

Das linke Feld ist sortiert, jetzt kommt das rechte Feld an die Reihe, Pivotelement ist die 5:

1	2	3	4	5	9	6	10	7	8

Im neuen linken Feld gibt es nur noch die 4, das neue Linke Feld ist sortiert. Im neuen rechten Feld wird die 10 als Pivotelement gewählt:

1	2	3	4	5	9	6	7	8	10

Im neuen linken Feld wird die 7 als Pivotelement gewählt:

1	2	3	4	5	6	7	9	8	10

Es gibt kein weiteres linkes Feld mehr. Nun wird die 8 als Pivotelement gewählt:

1	2	3	4	5	6	7	8	9	10

Das Feld ist sortiert.

Strings

Die Klasse String wird verwendet, um Zeichenketten zu verwalten bzw. zu speichern.

Strings können wie folgt erzeugt werden:

String text = "Dieser Text wird in einem String gespeichert";

Da String eine Klasse ist, sind die angelegten Strings Objekte. Also werden ihre Methoden mit dem Punktoperator aufgerufen. In der JAVA-Dokumentation kann man die Methoden einsehen.

Einige Beispiele die häufig Anwendung finden sollen hier dargestellt werden:

Strings vergleichen

Strings werden mit der Methode equals verglichen. Die Methode bekommt den zu vergleichenden String übergeben und gibt true zurück, wenn beide String gleich sind:

```
String text1 = "hallo";
String text2 = "hallo";
String text3 = "Hallo";
System.out.println (text1.equals(text2)); // Ausgabe: true
System.out.println (text1.equals(text3)); // Ausgabe: false
```

Über die Methode equalsIgnoreCase() können Strings verglichen werden, ohne dass der Vergleich Groß- und Kleinschreibung berücksichtigt:

```
System.out.println (text1.equalsIgnoreCase(text2)); // Ausgabe: true
System.out.println (text1.equalsIgnoreCase(text3)); // Ausgabe: true
```

Einlesen eines Strings:

Beim Einlesen eines Strings in der Console wird dieselbe Funktion, wie auch bei anderen Datentypen verwendet.

```
(name)=Konsole.readString();
```

Dies funktioniert nur mit Hilfe der Konsolendatei.

Umwandlung von Großbuchstaben in Kleinbuchstaben:

Für die Umwandlung von Großbuchstaben in Kleinbuchstaben gibt es die Funktion:

```
(name).toLowerCase();
```

Beispiel:

```
String text = "GROßBUCHSTABEN";
 text.toLowerCase();
 System.out.println(text);
    // Ausgabe: "großbuchstaben"
```

Umwandeln eines Strings in ein Char-Array:

Hierbei erhält jedes Zeichen eine eigene Stelle im Char-Array.

```
char [] (nameDesArrays)=(nameDesStrings).toCharArray();
```

Größe eines Strings ermitteln:

```
(name).length();
```

Beispiel:

```
String text = "Dies ist ein Text";
System.out.println(text.length());
// Ausgabe:  "17"
```

Alle ausgewählten Zeichen durch einen andere Zeichen ersetzen:

```
(name)=(name).replaceAll("ein String","ein anderer String");
```

Beispiel:

```
String text = "Entferne alle Leerzeichen";
    text = text.replaceAll(" ","");
    System.out.println(text);
    // Ausgabe: "EntfernealleLeerzeichen"
```

Alle ausgewählten Chars durch einen anderen gewählten Char ersetzen:

```
(name)=(name).replace('ein Char','ein anderer Char');
```

Beispiel:

```
String text="aaaa";
text = text.replace('a','b');
System.out.println(text);
//Ausgabe:"bbbb"
```

Zwei Strings vergleichen:

Diese Funktion vergleicht zwei Strings. (Gibt 0 zurück, wenn alle Zeichen gleich sind.)

```
(name).compareTo(String andererString);
```

Beispiel:

```
 String text = "Hallo"
 String text2 = "Hallo"
 System.out.println(text.compareTo(text2));
//Ausgabe:"0"
//Wenn die beiden Strings nicht denselben Inhalt haben wird eine Zahl, die nicht gleich 0 ist, ausgegeben, die vom Unterschied abhängt.
```

An welcher Stelle kommt ein gewähltes Zeichen in einem String zum ersten Mal vor :

Die Funktion überprüft einen String und gibt einen Integer aus, der angibt, an welcher Stelle das gewählte Zeichen zum ersten Mal vorkommt.

```
(name).indexOf(char);
```

Beispiel:

```
String text="Das E ist an 4. Stelle."
System.out.println(text.indexOf(E));
// Ausgabe."4", da bei 0 angefangen wird zu Zählen.
```

Ist ein String leer?:

Mithilfe dieser Funktion kann überprüft werden, ob ein String leer ist:

```
(name).isEmpty();
```

Beispiel:

```
String text="";
System.out.println(text.isEmpty());
//Ausgabe :"true"
```

Einen String aufsplitten und in ein Array von Strings speichern:

```
(name).split(String text)
```

Der String wird um den übergebenen String herum abgeschnitten und die einzelnen Teile in einem Array aus Strings gespeichert.

Beispiel:

```
String text="Dieser Text hat Leerzeichen";
String[] test = text.split(" ");
System.out.println(test[0]);
System.out.println(test[1]);
System.out.println(test[2]);
System.out.println(test[3]);


// Ausgabe: Dieser
//              Text
//              hat
//              Leerzeichen
```

Structs

Benötigt man zur Erstellung eines Programms Datensätze oder Datentypen die sich aus mehreren Variablentypen zusammensetzen, so reichen Arrays oft nicht aus, da sie nur mehrere Daten des gleichen Typs verwalten können. Eine Adresse z.B. beinhaltet jedoch sowohl Strings als auch Integervariablen. Eine Lösung für dieses Problem sind die structs[2].
Ein struct ist ein selbstdefiniertes Variablenkonstrukt und kommt eigentlich aus der Programmiersprache C. In Java kann man aber mit Hilfe eines Kniffes einen Struct realisieren. Soll Beispielsweise ein Struct Adresse erstellt werden so steht als erstes die Überlegung an, aus welchen Datentypen sich eine Adresse zusammensetzt. Besteht diese beispielsweise aus einen Vornamen (String), einem Namen (String), einer Strasse (String), einer Postletzahl (int) und einem Ort (String), so setzen sich hier unterschiedliche Datentypen zusammen.
Um einen Struct Adresse anzulegen wird eine neue Datei Adresse.java erzeugt und mit dem folgenden Inhalt gefüllt:

```
public class Adresse
{
      public String vorname, name, strasse, ort;
      public int plz;
}
```

Will man nun den Struct Adresse in einer Anwendung benutzen so muss die Datei Adresse.java im gleichen Verzeichnis liegen, wie die eigentliche Anwendung. Auf den Struct wird nun wie im folgenden Beispiel zugegriffen:

```
public class Anwendung
{
      public static void main (String args[])
      {
            Adresse adr = new Adresse();  //ein neuer struct wird erzeugt
            adr.vorname = "Fritz";
            adr.name = "Pils";
            adr.strasse = "Hauptstrasse";
            adr.plz = 12345;
            adr.ort = "Hausen";
      }
}
```

Auch können für den struct eigene Funktionen geschrieben werden. Beispielsweise könnte in der Klasse Anwendung die folgende Funktion stehen, um die Inhalte einer Adresse auf den Bildschirm auszugeben:

```
public static void adresseAusgeben (Adresse a)
{
      System.out.println (a.vorname);
      System.out.println (a.name);
      System.out.println (a.strasse);
      System.out.println (a.plz + " " + a.ort);
}
```

Wichtig: Will man Funktionen in der eigentlichen Anwendungsdatei schreiben, so sind diese als static zu deklarieren!

[2] Natürlich würde mich jetzt jeder Programmierer im Sinne einer objektorientierten Programmierung lynchen, jedoch ist diese didaktische Reduzierung im Kurs „Strukturierte Programmierung“ von mir vertretbar.

Objektorientierte Programmierung

Klasse (Programmierung)[3]:

Unter einer Klasse (auch *Objekttyp* genannt) versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen *Bauplan* für eine Reihe von ähnlichen Objekten. (Z. B. ist der Bauplan eines Autos etwas ganz anderes als ein Auto selbst!)

Die Klasse dient als Bauplan für die Abbildung von realen Objekten in Softwareobjekte und beschreibt Attribute (Eigenschaften) und Methoden (Verhaltensweisen) der Objekte. Verallgemeinernd könnte man auch sagen, dass eine Klasse dem Datentyp eines Objekts entspricht. Formal gesehen belegt eine Klasse somit zur Programm-Ausführungszeit *keinen Arbeitsspeicher*, sondern immer nur die Objekte, die von ihr instanziiert wurden.

[3] aus: http://de.wikipedia.org/wiki/Klasse_(Programmierung)

In vielen Programmiersprachen ist es üblich, dass der Name einer Klasse mit einem Großbuchstaben beginnt, Variablennamen dagegen mit einem Kleinbuchstaben.

Objekte[4]:

Ein Objekt bezeichnet in der objektorientierten Programmierung (OOP) ein Exemplar eines bestimmten Datentyps oder einer bestimmten Klasse (auch „Objekttyp“ genannt). In diesem Zusammenhang werden Objekte auch als „Instanzen einer Klasse“ bezeichnet. Objekte sind also konkrete Ausprägungen („Instanzen“) eines Objekttyps.

Attribute[5]:

Ein Attribut (von lateinisch *attribuere* = zuteilen, zuordnen), auch „Eigenschaft“ genannt, gilt im Allgemeinen als Merkmal, Kennzeichen, Informationsdetail etc., das einem konkreten Objekt zugeordnet ist. Dabei wird unterschieden zwischen der Bedeutung (z. B. 'Augenfarbe') und der konkreten Ausprägung (z.B. 'blau') des Attributs.

In der Informatik wird unter Attribut die *Definitionsebene* für diese Merkmale etc. verstanden. Als solche werden sie (i. d. R. im Rahmen von Projekten) analytisch ermittelt, definiert und beschrieben sowie für einen bestimmten *Objekttyp* (z. B. 'Person') als Elemente seiner *Struktur* festgelegt ('modelliert').
Daten über die Objekte werden in dieser Struktur und (i. d. R.) nur mit ihrem Inhalt, den *Attributwerten* gespeichert. Jedes Objekt repräsentiert sich somit durch die Gesamtheit seiner Attributwerte.

Methoden:

Eine Methode beschreibt die Verhaltensweise eines Objektes.

Beispiel:

Einem Objekt aus der Klasse Konto kann die Nachricht `boolean abheben (1000)` geschickt werden. Das Objekt führt diese Methode aus und verringert den Kontostand um den Betrag 1000. Gelingt dies, so wird ein `true` zurückgegeben. Würde der Kontostand überzogen, so wird der Kontostand nicht verringert und ein `false` zurückgegeben. Eine Methode hat, wie auch die Funktionen in der strukturierten Programmierung einen Namen sowie mögliche Übergabeparameter und gegebenenfalls einen Rückgabeparameter.

Datenkapselung (Programmierung)[6]:

Datenkapselung im objektorientierten Paradigma

[4] aus: http://de.wikipedia.org/wiki/Objekt_(Programmierung)
[5] aus: http://de.wikipedia.org/wiki/Attribut_(Objekt)
[6] aus: http://de.wikipedia.org/wiki/Datenkapselung_(Programmierung)

Kapselung ist auch ein wichtiges Konzept der objektorientierten Programmierung. Als Kapselung bezeichnet man den kontrollierten Zugriff auf Methoden bzw. Attribute von Klassen. Klassen können den internen Zustand anderer Klassen nicht in unerwarteter Weise lesen oder ändern. Eine Klasse hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit der Klasse interagiert werden kann. Dies verhindert das Umgehen von Invarianten des Programms. Vom Innenleben einer Klasse soll der Verwender – gemeint sind sowohl die Algorithmen, die mit der Klasse arbeiten, als auch der Programmierer, der diese entwickelt – möglichst wenig wissen müssen (*Geheimnisprinzip*). Durch die Kapselung werden nur Informationen über das „Was“ (Funktionsweise) einer Klasse nach außen sichtbar, nicht aber das „Wie“ (die interne Repräsentation). Dadurch wird eine Schnittstelle nach außen definiert und zugleich dokumentiert.

Für die Kapselung verwendete Zugriffsarten

Die UML als De-facto-Standardnotation erlaubt die Modellierung folgender Zugriffsarten (in Klammern die Kurznotation der UML):

`public (+)`

zugreifbar für alle Ausprägungen (auch die anderer Klassen),

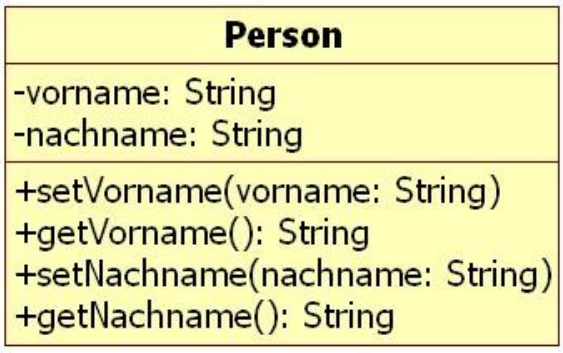
`private (-)`

Nur für Ausprägungen der eigenen Klasse zugreifbar,

Beispiel:

Jede Person hat einen Vornamen und einen Nachnamen.

Vorname und Nachname sind also Attribute der Klasse Person. Im Klassendiagramm würde die Klasse Person wie folgt aussehen:



Umgesetzt im JAVA-Quelltext:

```
pubic class Person
{
 private String vorname;
 private String nachname;

 public void setVorname(String vorname)
 {
       this.vorname = vorname;
 }

 public String getVorname()
 {
       return this.vorname;
 }

 public void setNachname(String nachname)
 {
       this.nachname = nachname;
 }

 public String getNachname()
 {
       return this.nachname;
 }
}
```

Schlüsselwort static

Eine Person hat einen Vornamen und einen Nachnamen. Demnach sieht der Quellcode der Klasse Person wie folgt aus:

```
public class Person{
  //Attribute
  private String vorname, nachname;

  public Person(String vorname, String nachname){
    this.vorname = vorname;
    this.nachname = nachname;
  }
  //Methoden
  public void setVorname(String vorname){
    this.vorname = vorname;
  }

  public String getVorname(){
    return this.vorname;
  }

  public void setNachname(String nachname){
    this.nachname = nachname;
  }

  public String getNachname(){
    return this.nachname;
  }

  public String toString(){
    return this.vorname + " " + this.nachname;
  }
```

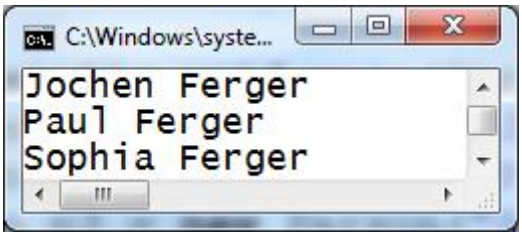
```
}
```

Man sieht, dass die Attribute vorname und nachname direkt an das jeweilige Objekt gebunden sind. Das macht ja auch Sinn, wenn Sie jemanden fragen, wie er heißt, wird er nur seine Daten bekannt geben.

Eine mögliche Anwendung mit drei Personen könnte wie folgt aussehen:

```
public class Anwendung {

  public static void main(String[] args) {
    Person p1 = new Person("Jochen","Ferger");
    Person p2 = new Person("Paul","Ferger");
    Person p3 = new Person("Sophia","Ferger");
    System.out.println(p1.toString());
    System.out.println(p2.toString());
    System.out.println(p3.toString());
  }
```

Die folgende Bildschirmausgabe wäre die Folge:



Will man nun wissen, wie viele Objekte es aus der Klasse Person gibt, muss man dies im Moment in der Anwendung speichern:

```
public class Anwendung {
  public static void main(String[] args) {
    int anzahlPersonen = 0;
    Person p1 = new Person("Jochen","Ferger");
    anzahlPersonen = 1;
    Person p2 = new Person("Paul","Ferger");
    anzahlPersonen = 2;
    Person p3 = new Person("Sophia","Ferger");
    anzahlPersonen = 3;
    System.out.println(p1.toString());
    System.out.println(p2.toString());
    System.out.println(p3.toString());
  }
}
```

Schöner wäre es, man könnte direkt mit der Klasse Person auf die entsprechende Anzahl der erzeugten Objekte zugreifen. Hier hilft das Schlüsselwort static. Die Klasse Person ändert sich wie folgt:

```
public class Person{
```

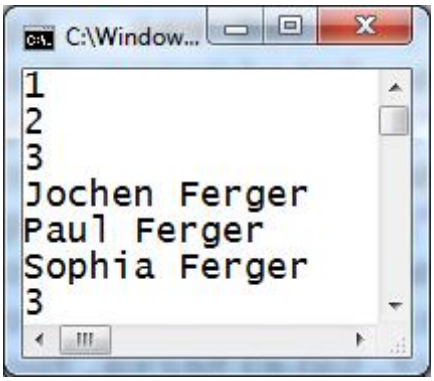
```
  //statisches Atribut
  static int anzahlPersonen = 0;
  //Attribute
  private String vorname, nachname;

  public Person(String vorname, String nachname){
    this.vorname = vorname;
    this.nachname = nachname;
    anzahlPersonen = anzahlPersonen + 1;
  }
..
..
```

Zur Erklärung:

Objekte aus der Person wissen immer noch um ihren Vornamen und ihren Nachnamen. Allerdings haben alle Objekte aus der Klasse Person nun die Information anzahlPersonen (und alle die gleiche Information!) zur Verfügung. und zwar jeweils mit dem gleichen Wert. Auf diese Information kann über das Objekt, aber auch direkt über den Klassennamen zugegriffen werden:

```
public class Anwendung {
  public static void main(String[] args) {
    Person p1 = new Person("Jochen","Ferger");
    System.out.println(p1.anzahlPersonen);
    Person p2 = new Person("Paul","Ferger");
    System.out.println(p1.anzahlPersonen);
    Person p3 = new Person("Sophia","Ferger");
    System.out.println(p1.anzahlPersonen);
    System.out.println(p1.toString());
    System.out.println(p2.toString());
    System.out.println(p3.toString());
    System.out.println(Person.anzahlPersonen);
  }
}
```

Das Programm hat folgende Bildschirmausgabe zur Folge:



Man sieht: Die Information anzahlPersonen ist nicht von dem jeweiligen Objekt abhängig, sondern kann von jedem beliebigen Objekt oder aber auch von der Klasse abgerufen werden.

Umfangreicheres Beispiel zu der objektorientierten Analyse:

Situationsbeschreibung:

Schüler haben einen Vornamen und einen Namen. Sie besitzen ein Geburtsdatum und ein Geschlecht. Weiter haben Schüler eine Adresse, bestehend aus Straße, Hausnummer, Postleitzahl und Ort.

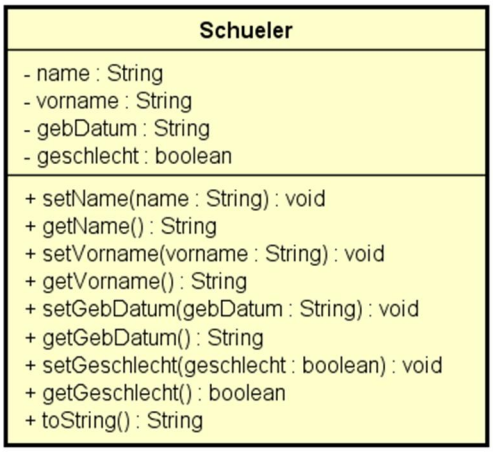
Aufgabenstellung:

Erstellen Sie das Klassendiagramm dieser Situation und implementieren Sie den Quelltext!

Lösung:

Der Schüler besitzt insgesamt vier Attribute: Name, Vorname, Geburtsdatum und Geschlecht. Diese Attribute sind alle recht simpel gehalten (nur ein Datentyp). Die Adresse, die jeder Schüler besitzt ist zwar eine Eigenschaft des Schülers, besteht jedoch aus weiteren vier Informationen. Sie ist also komplex. Somit wird sie zur eigenen Klasse.

Das Klassendiagramm des Schülers sieht also zuerst einmal wie folgt aus:



Erklärung:

Die vier Attribute werden nach dem Kapselungsprinzip auf die Sichtbarkeit: privat gesetzt. Im Klassendiagramm wird dies durch das Zeichen „-“ dargestellt. Der Entwickler kann nun selbst entscheiden, ob er der Außenwelt Zugriff zu den Attributen gewähren will. oder nicht.

Oft wird der Zugriff durch die set- und get-Methoden realisiert (die Methoden werden auch als setter und getter bezeichnet). Da hier ein Zugriff erfolgen soll, werden diese Methoden durch die Sichtbarkeit: öffentlich (Zeichen: „+“) gekennzeichnet.

Die toString()-Methode gehört zu einer Klasse einfach dazu. Sie gibt einen repräsentativen String zurück, der die Daten beispielsweise durch „;“ getrennt zurückgibt.

Quelltext:

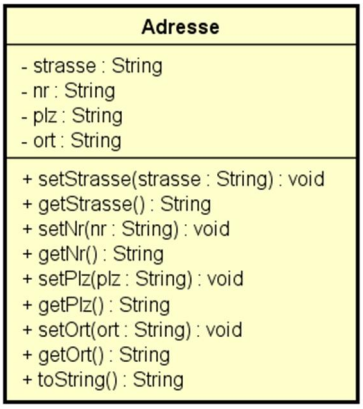
Im Quelltext wird die Klasse wir folgt umgesetzt:

```
public class Schueler{
  //Anfang Attribute
  private String name;
  private String vorname;
```

```
  private String gebDatum;
  private boolean geschlecht;

  //Anfang Methoden
    public void setName(String name){
    this.name = name;
  }

  public String getName(){
    return name;
  }

  public void setVorname(String vorname){
    this.vorname = vorname;
  }

  public String getVorname(){
    return vorname;
  }

  public void setGebDatum(String gebDatum){
    this.gebDatum = gebDatum;
  }

  public String getGebDatum(){
    return gebDatum;
  }

  public void setGeschlecht(boolean geschlecht){
    this.geschlecht = geschlecht;
  }

  public boolean getGeschlecht(){
    return geschlecht;
  }

  public String toString(){
    String rueck="";
    if (geschlecht) {
      rueck += "Frau" ;
    }else{
      rueck += "Herr";
    }
    rueck = rueck + ";"+vorname+";"+name+";"+gebDatum;
    return rueck;
  }
}//end of class
```

Modellierung der Klasse Adresse

Die Adresse besitzt vier einfache Attribute, welche alle durch den Datentyp String realisiert werden können. Nach den Vorgaben des Kapselungprinzips sieht die Klasse wie folgt aus:



Die Umsetzung im Quelltext:

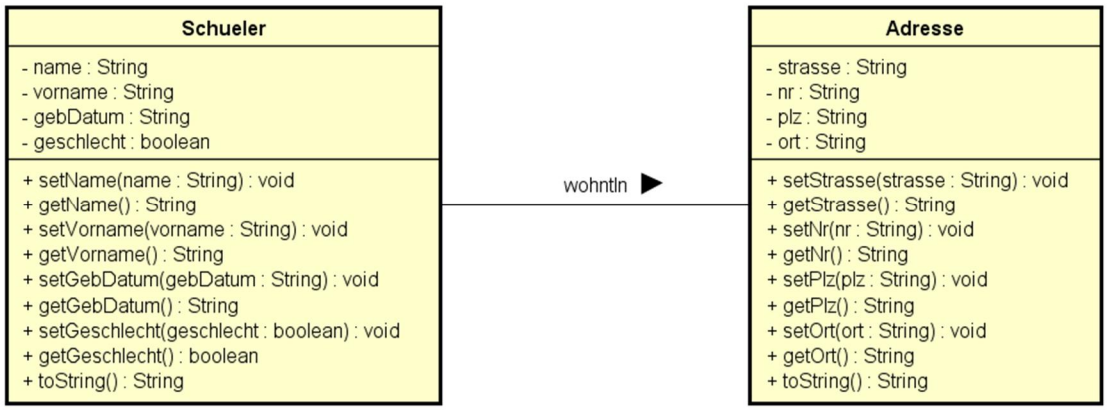
```
public class Adresse {

  // Anfang Attribute
  private String strasse;
  private String nr;
  private String plz;
  private String ort;
  // Ende Attribute

  //Anfang Methoden
  public String getStrasse() {
    return strasse;
  }

  public void setStrasse(String strasse) {
    this.strasse = strasse;
  }

  public String getNr() {
    return nr;
  }

  public void setNr(String nr) {
    this.nr = nr;
  }

  public String getPlz() {
    return plz;
  }

  public void setPlz(String plz) {
    this.plz = plz;
  }

  public String getOrt() {
    return ort;
  }

  public void setOrt(String ort) {
    this.ort = ort;
  }
```

```
  public String toString() {
    return strasse+";"+nr+";"+plz+";"+ort;
  }
  // Ende Methoden
} // end of Adresse
```

Assoziationen

Wie bekommt man nun eine Verbindung zwischen Schüler und Adresse?

Hier helfen die Assoziationen. Eine Assoziation ist eine Verbindung zwischen zwei Klassen. Eine Assoziation wir im Klassendiaramm durch einen einfachen Strich dargestellt, der die beiden Klassen verbindet. An den Strich schreibt man einen repäsentativen Namen für die Assoziation. Ein Pfeil an diesem Namen stellt die Leserichtung dar:



Navigierbarkeit, Multiplizität, Rolle

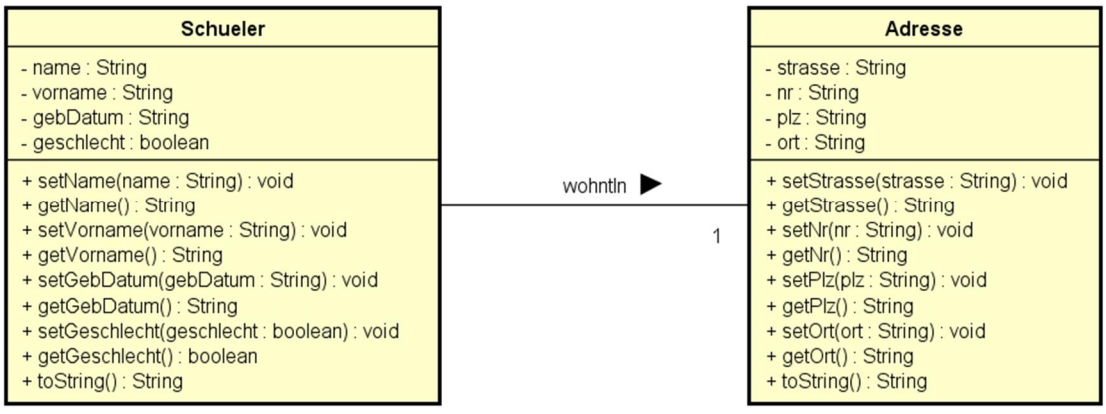
Die Navigierbarkeit zeigt an, von welcher Klasse man auf die andere zugreifen kann. Dies wird dann durch einen Pfeil am Ende der Assoziation dargestellt.

Die Multiplizität stellt die Anzahl der Objekte dar, mit denen die beiden Klassen in Verbindung stehen. Sie wird in einer Min-Max-Notation aufgeschrieben. Um die Multiplizität herauszufinden, helfen zwei ragen, die jeweils aus Sicht der beiden Klassen gestellt werden:

Wie viele Adressen muss ein Schüler mindestens haben? Antwort: Eine.

Wie viele Adressen kann ein Schüler maximal haben? Antwort: Eine.

Somit ergibt sich eine 1..1 – Multiplizität:



Anmerkung:

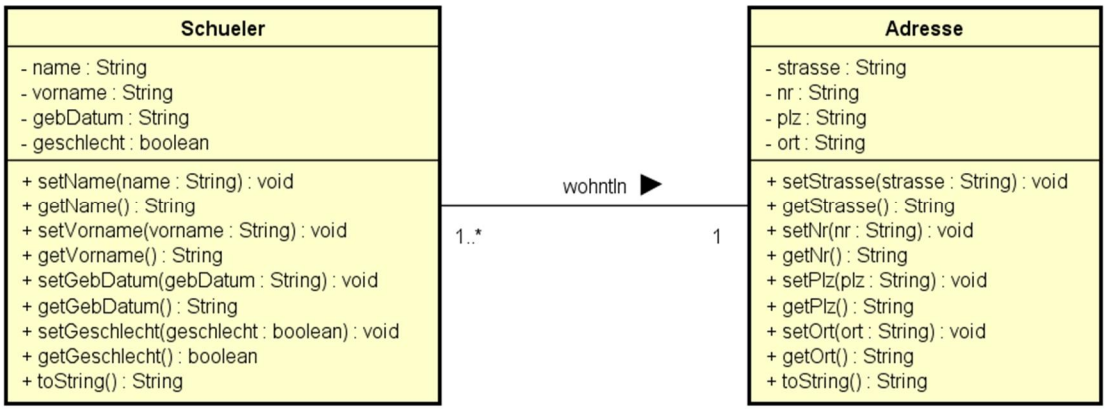
Im UML-Tool Astah wird eine 1..1 – Multiplizität nur mit einer 1 dargestellt.

Weiter geht es mit den beiden Fragen aus Sicht der Adresse:

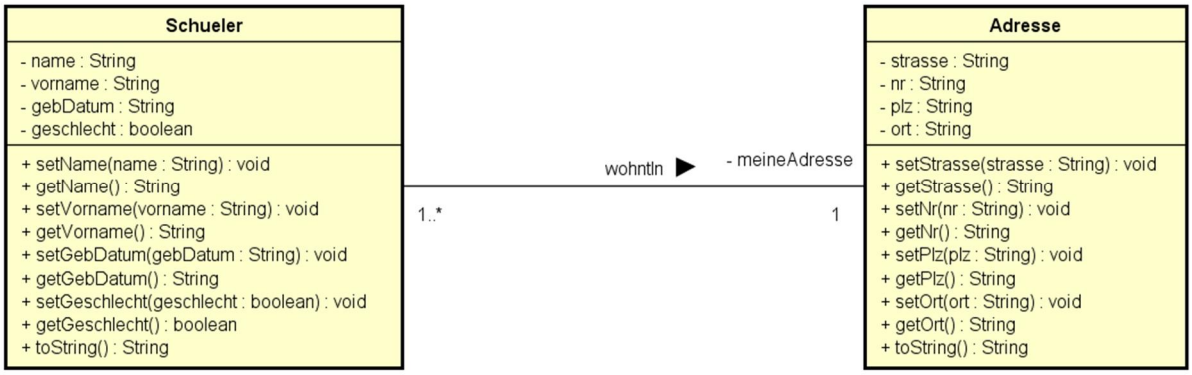
Wie viele Schüler wohnen mindestens an einer Adresse? Antwort: Einer.

Wie viele Schüler wohnen maximal an einer Adresse? Antwort: Umbestimmt, es können mehrere Schüler an der gleichen Adresse wohnen.

Dies wird im Klassendiagramm mit einer 1..* - Multiplizität dargestellt:



Die Rolle nützt im Diagramm, dass man erkennen kann, welche Eigenschaft die Objekte für die Objekte der jeweiligen anderen Klasse besitzt. Die Rolle taucht später dann auch im Quelltext auf. Sie bekommt eine Sichtbarkeit (im Allgemeinen privat):



Im Quelltext wird die Assoziation durch ein Assoziationsattribut dargestellt. Durch entsprechende set- und get- Methoden kann dann die Navigierbarkeit realisiert werden. Da hier nur eine Navigierbarkeit vom Schüler zur Adresse dargestellt wird, ändert sich der Quelltext der Klasse Schueler wie folgt:

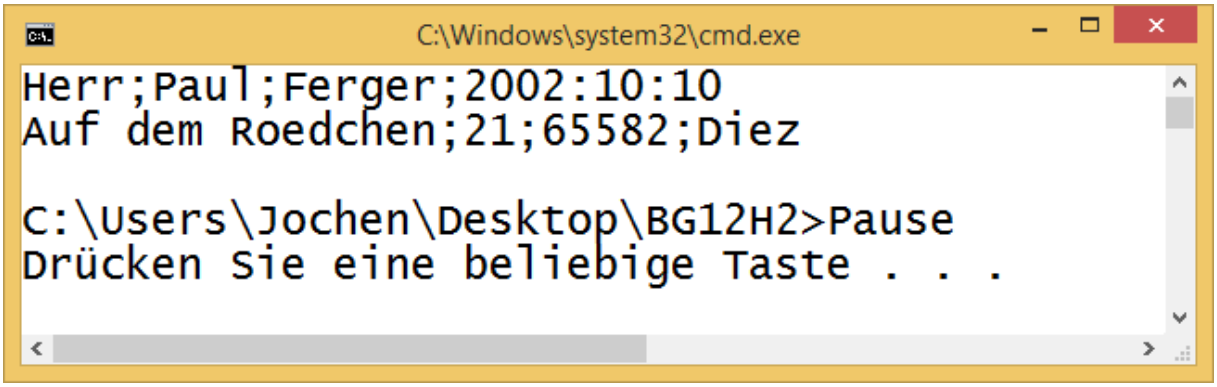
```
public class Schueler{
  //Anfang Attribute
  private String name;
  private String vorname;
  private String gebDatum;
  private boolean geschlecht;
  //Ende Attribute
```

```
  //Anfang Assoziationsattribut
  private Adresse meineAdresse;

  //Anfang Methoden
  public void setName(String name){
    this.name = name;
  }

  public String getName(){
    return name;
  }

  public void setVorname(String vorname){
    this.vorname=vorname;
  }

  public String getVorname(){
    return vorname;
  }

  public void setGebDatum(String gebDatum){
    this.gebDatum=gebDatum;
  }

  public String getGebDatum(){
    return gebDatum;
  }

  public void setGeschlecht(boolean geschlecht){
    this.geschlecht=geschlecht;
  }

  public boolean getGeschlecht(){
    return geschlecht;
  }

  public void setMeineAdresse(Adresse meineAdresse){
    this.meineAdresse=meineAdresse;
  }

  public Adresse getMeineAdresse(){
    return meineAdresse;
  }

  public String toString(){
    String zurueck = "";
    if (geschlecht) {
      zurueck += "Frau";     //zurueck = zurueck + "Frau";
    }else{
      zurueck +="Herr";
    }
    zurueck += ";"+vorname+";"+name+";"+gebDatum;
    return zurueck;
  }
}//end of class
```

In einer Main-Klasse kann dies nun getestet werden:

```
public class Main {
  public static void main(String[] args) {
    Schueler s1 = new Schueler();
    s1.setName("Ferger");
    s1.setVorname("Paul");
    s1.setGebDatum("2002:10:10");
    s1.setGeschlecht(false);
    Adresse a1 = new Adresse();
    a1.setStrasse("Auf dem Roedchen");
    a1.setNr("21");
    a1.setPlz("65582");
    a1.setOrt("Diez");
    s1.setMeineAdresse(a1);
    System.out.println(s1.toString());
    System.out.println(s1.getMeineAdresse().toString());
  } // end of main
} // end of class Main
```

Der Ablauf ergibt die folgende Bildschirmausgabe:



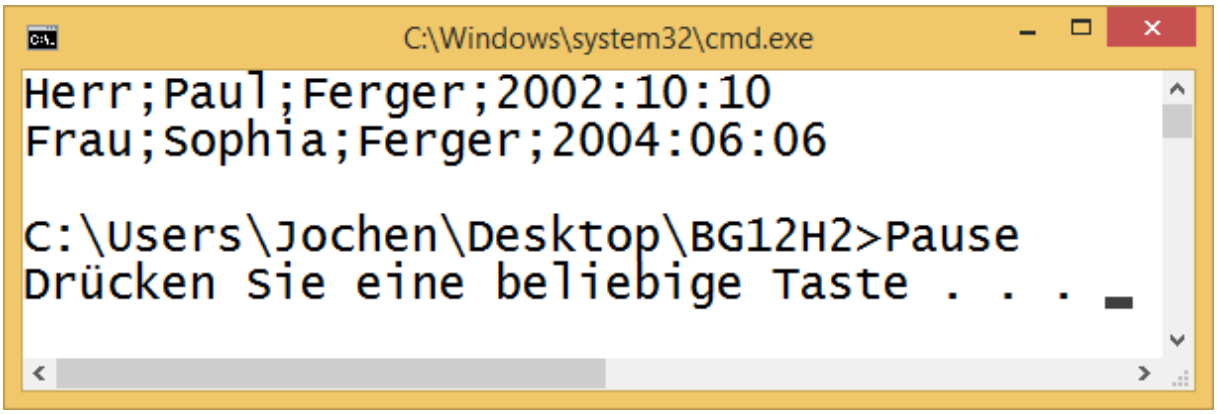
Man sieht, dass nun nicht mehr direkt über die Referenz a1 auf das Adressenobjekt zugreift, sondern über die Assoziation.

Referenzen

```
Schueler s1 = new Schueler();
```

Beim Erstellen eines Objektes wird dieses über das Schlüsselwort new erzeugt. Das Objekt liegt nun im Speicher des Systems. Auf diesen Speicherplatz kann man mittels der Referenz s1 zugreifen. Dies bedeutet also, dass s1 nicht das Objekt selbst darstellt, sondern nur eine Zugriffsmöglichkeit auf den entsprechenden Speicherplatz. Dies wird im folgenden Beispiel deutlich:

```
public class Main {
  public static void main(String[] args) {
    Schueler s1 = new Schueler();
    s1.setName("Ferger");
    s1.setVorname("Paul");
    s1.setGebDatum("2002:10:10");
    s1.setGeschlecht(false);
    System.out.println(s1.toString());
    Schueler s2 = s1;
    s2.setVorname("Sophia");
    s2.setGeschlecht(true);
    s2.setGebDatum("2002:06:06");
    System.out.println(s1);
  } // end of main
} // end of class Main
```



Erklärung:

In der Zeile:

```
Schueler s2 = s1;
```

wird kein neues Objekt erzeugt. Es entsteht eine zweite Referenz auf das schon erstellte Objekt.

Konstruktor / Überladen von Methoden

Schaut man sich das obige Beispiel an, so sind die vielen set-Methoden hilfreich aber umständlich. Einfacher wäre es, man gibt dem zu erzeugenden Objekt die Eigenschaften bei der Erstellung mit. Hierzu dient der Konstruktor. Er ist eine Methode, die exakt so wie die Klasse heißt. Sie hat keinen Rückgabewert (auch nicht void).

Im Quelltext könnte ein solcher Konstruktor der Klasse Schueler wie folgt aussehen:

```
  public Schueler(String name, String vorname, String gebDatum, boolean geschlecht){
    this.name = name;
    this.vorname = vorname;
    this.gebDatum = gebDatum;
    this.geschlecht = geschlecht;
  }
```

Das Objekt kann nun wie folgt erstellt werden:

```
Schueler s1 = new Schueler("Ferger","Paul","2010:10:10",false);
```

Will man ein zweites Objekt nach der alten Methode erstellt:

```
Schueler s2 = new Schueler();
```

kommt es nun zu einer Fehlermeldung:

```
Compiliere C:\Users\Jochen\Desktop\BG12H2\Main.java mit Java-Compiler
Main.java:4:19: error: constructor Schueler in class Schueler cannot be applied to given
types;
    Schueler s2 = new Schueler();
                  ^
  required: String,String,String,boolean
  found: no arguments
```

Erklärung:

Da nun ein Konstruktor mit Übergabeparameter implementiert wurde gibt es keinen anderen ohne Parameter mehr. Dieser existiert also immer automatisch (er wird als Standardkonstruktor bezeichnet), es sei denn, man schreibt einen eigenen.

Es ist aber möglich, den alten Konstruktor wieder herzustellen:

```
public Schueler(){
}
```

wird dem Quelltext hinzugefügt. Jetzt lässt sich der Quelltext problemlos kompilieren.

Anmerkung:

Man kann also im Prinzip so viele Konstruktoren schreiben, wie man will. Sie müssen sich in den Übergabeparametern unterscheiden. Der Konstruktor ist eine Methode. Das mehrfache Implementieren von Methoden mit gleichem Namen und unterschiedlichen Übergabeparametern nennt man Überladen von Methoden.

Vererbung, Überschreiben von Methoden, Polymorphismus, Late Binding

Vererbung

Situationsbeschreibung:

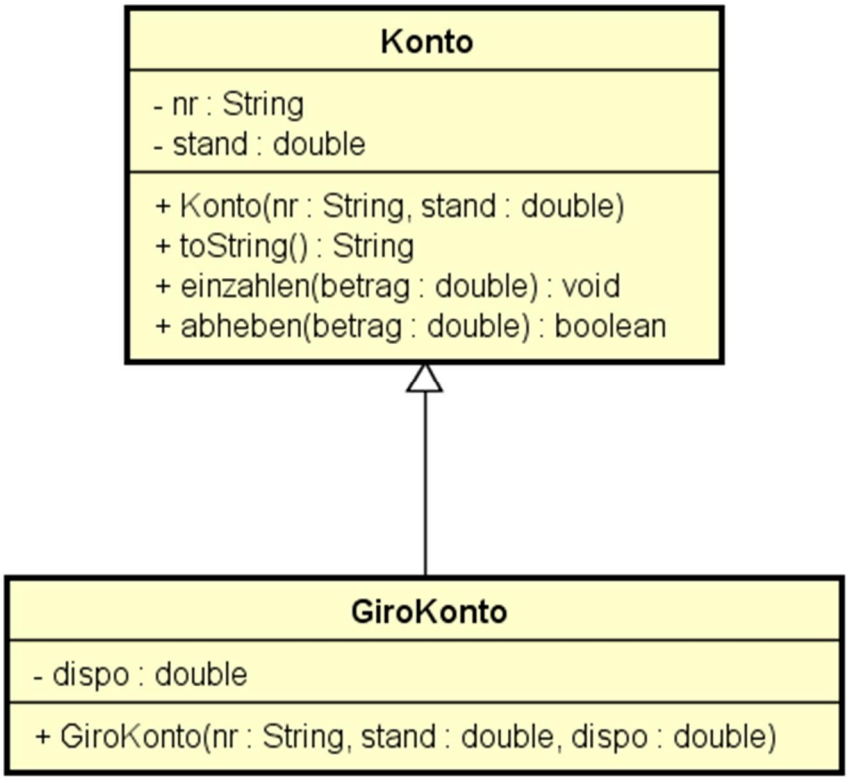
Die Klasse Konto kapselt eine Kontonummer und einen Kontostand. Zusätzlich zu den set- und get-Methoden gibt es einen geeigneten Konstruktor und eine toString-Methode. Ein Girokonto hat zusätzlich einen Dispositionsrahmen. Da dies ein weiteres Attribut darstellt, ist ein geeigneter Konstruktor hinzuzufügen.

Aufgabenstellung:

Stellen Sie beide Klassen im Diagramm dar und implementieren Sie die Klassen.

Lösung:

Betrachtet man die beiden Klassen genau, so erkennt man, dass die Klasse GiroKonto eine Erweiterung der Klasse Konto darstellt. Und genau so wird die Modellierung angesetzt:



Die Verbindung zwischen GiroKonto und Konto bedeutet, dass die Klasse GiroKonto von der Klasse Konto erbt. Dies wird durch das Schlüsselwort `extends` dargestellt. Sie übernimmt damit alle Attribute und Methoden. Im Quellcode sieht dies vermeintlich wie folgt aus:

Klasse Konto:

```
public class Konto {

  // Anfang Attribute
  private String nr;
  private double stand;
  // Ende Attribute

  // Anfang Methoden
  public Konto(String nr, double stand){
    this.nr = nr;
    this.stand = stand;
  }


  public String getNr() {
    return nr;
  }

  public void setNr(String nr) {
    this.nr = nr;
  }

  public double getStand() {
    return stand;
  }
```

```
  public void setStand(double stand) {
    this.stand = stand;
  }

  public String toString(){
    return nr+";"+stand;
  }


  // Ende Methoden
} // end of Konto
```

Klasse GiroKonto:

```
public class GiroKonto extends Konto {

  // Anfang Attribute
  private double dispo;
  // Ende Attribute

  // Anfang Methoden
  public GiroKonto(String nr, double stand,double dispo){
    this.nr=nr;
    this.stand=stand;
    this.dispo=dispo;
  }


  public double getDispo() {
    return dispo;
  }

  public void setDispo(double dispo) {
    this.dispo = dispo;
  }

  // Ende Methoden
} // end of GiroKonto
```

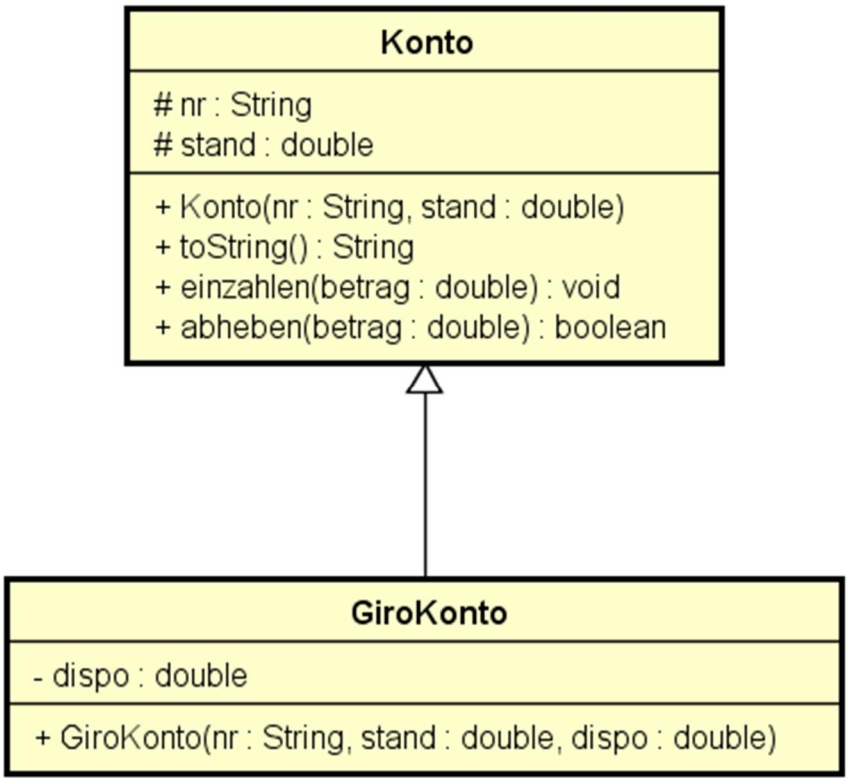
Beim Kompilieren führt dieser Quelltext zu drei Fehlern:

```
Compiliere
C:\Users\Jochen\Documents\01_Schule\01_Informatik\01_Programmierung\00_Skript\GiroKonto.java
mit Java-Compiler
GiroKonto.java:16:57: error: constructor Konto in class Konto cannot be applied to given
types;
  public GiroKonto(String nr, double stand,double dispo){
                                                        ^
  required: String,double
  found: no arguments
  reason: actual and formal argument lists differ in length
```

```
GiroKonto.java:17:9: error: nr has private access in Konto
    this.nr=nr;
        ^
GiroKonto.java:18:9: error: stand has private access in Konto
    this.stand=stand;
        ^
3 errors
```

Betrachtet man zuerst die Fehler 2 und 3, so stellt man fest, dass auch eine erbende Klasse nicht auf die privat deklarierten Attribute der Klasse Konto (man nennt diese Klasse auch Oberklasse) zugreifen kann. Dies könnte man durch ein Ersetzen der Attribute durch die entsprechenden get-Methoden lösen. Eleganter ist es, die Attribute der Oberklasse als protected zu deklarieren. protected bedeutet, dass alle erbenden Klassen direkt auf die entsprechenden Attribute zugreifen können. Alle anderen Klassen weiterhin nicht. Im Klassendiagramm wird dies durch das Zeichen „#“ dargestellt:

Klassendiagramm:



Quellcode Klasse Konto:

```
public class Konto {


 // Anfang Attribute

 protected String nr;

 protected double stand;

 // Ende Attribute
```

…

…

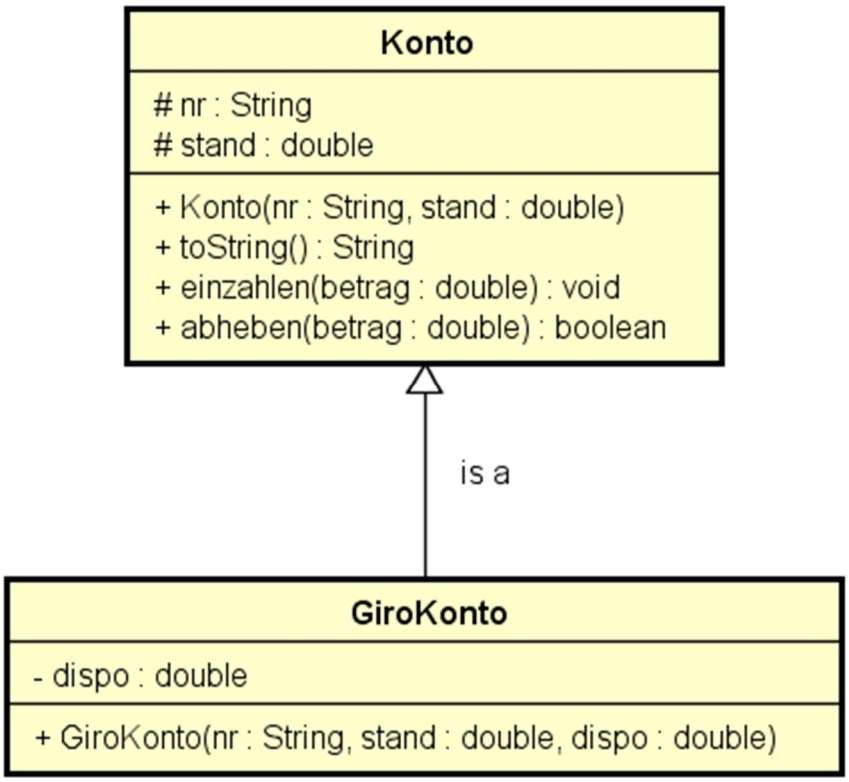
Somit wurden zwei Fehlerquellen beseitigt. Es bleibt:

```
Compiliere
C:\Users\Jochen\Documents\01_Schule\01_Informatik\01_Programmierung\00_Skript\GiroKonto.java
mit Java-Compiler
GiroKonto.java:9:57: error: constructor Konto in class Konto cannot be applied to given types;
  public GiroKonto(String nr, double stand,double dispo){
                                                        ^
  required: String,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Es wird bemängelt, dass die Klasse Konto keinen Konstruktor mit den entsprechenden drei Übergabeparametern besitzt.

Erklärung und Lösung:

In der UML1.1 stand beim Vererbungspfeil noch „is a“ anbei:



Dies bedeutet, dass ein Objekt der Klasse GiroKonto auch ein Objekt der Klasse Konto ist. Dadurch wird im Konstruktor der Klasse GiroKonto automatisch der Konstruktor der Klasse Konto aufgerufen. Und dies mit den gleichen Übergabeparameter, welche der aufgerufene Konstruktor von GiroKonto hat. Da die Klasse Konto aber einen solchen Konstruktor nicht besitzt, kommt es zu einer Fehlermeldung.

Beheben kann man dies durch Aufruf des korrekten Konstruktors von Konto. Das geht aber nur, wenn man ein Objekt aus der Klasse Konto benennen kann. Hier hilft das Schlüsselwort super. Mittels super kann man direkt auf Attribute und Methoden der jeweiligen Oberklasse zugreifen:

Abänderung im Quellcode der Klasse GiroKonto:

```
  public GiroKonto(String nr, double stand,double dispo){
    super(nr,stand);
    this.dispo=dispo;
  }
```

Nun lässt sich der Quellcode sauber kompilieren.

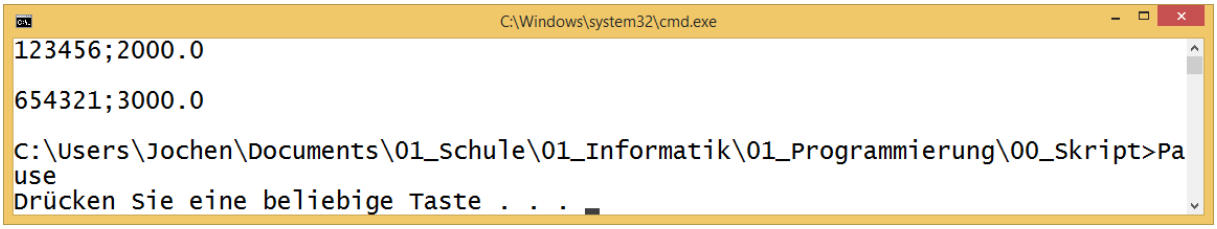
Überschreiben von Methoden

In einer main-Methode sollen nun je ein Objekt der Klassen angelegt werden und jeweils die Daten der toString-Methode ausgegeben werden:

Quellcode der Klasse:

```
public class Bank {
  public static void main(String[] args) {
    Konto k = new Konto("123456",2000);
    GiroKonto g = new GiroKonto("654321",3000,1000);
    System.out.println(k.toString());
    System.out.println();
    System.out.println(g.toString());
  } // end of main
} // end of class Bank
```

Das Programm hat die folgende Bildschirmausgabe zur Folge:



Man sieht, dass beim Objekt aus der Klasse GiroKonto der Dispositionsrahmen nicht ausgegeben wird. Dies ist klar, da in der Methode toString der Klasse Konto kein Attribut dispo bekannt ist.

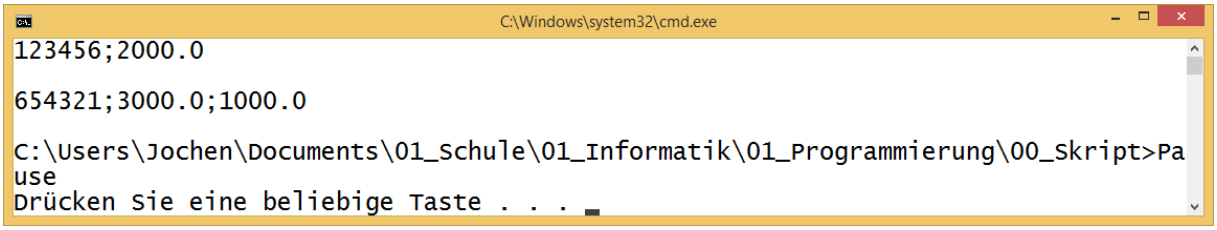
Zu lösen ist dieses Probelem mittels einer neuen Methode toString in der Klasse GiroKonto. Man nennt diesen Vorgang: „Überschreiben von Methoden“. Dies bedeutet, dass in einer erbenden Klasse eine Methode implementiert wird, deren Namen und deren Übergabeparameter zur entsprechenden Methode der Oberklasse identisch sind:

Quellcode der Methode toString in der Klasse GiroKonto:

```
  public String toString(){
    return super.toString() + ";"+dispo;
  }
```

Erklärung:

Die Methode bedient sich über das Schlüsselwort super der toString-Methode der Klasse Konto und gibt zusätzlich den Wert des Attributs dispo zurück. Die Bildschirmausgabe ändert sich nun:



Polymorphismus

Polymorphismus bedeutet, dass ein Objekt in einer Erbhierarchie mehrere Formen annehmen kann.

Beispiel:

Die Klasse GiroKonto erbt von der Klasse Konto. Beide Klassen besitzen einen Standardkonstruktor. Nach der Polymorphie ist nun folgendes möglich:

```
Konto k1 = new Konto();
Konto k2 = new GiroKonto();
```

Man sieht, dass beide Objekte in einer Referenz der Klasse Konto abgelegt werden. Die Referenz k2 zeigt aber auf ein Objekt der Klasse GiroKonto. Vorteil hier ist, dass in einer Erbhierarchie alle Objekte beispielsweise in einer Liste abgespeichert werden können.

Late Binding

Beide Klassen bekommen eine toString-Methode, die die jeweiligen Attribute zurückgeben.

Ruft man nun die folgende Quelltextzeilen auf:

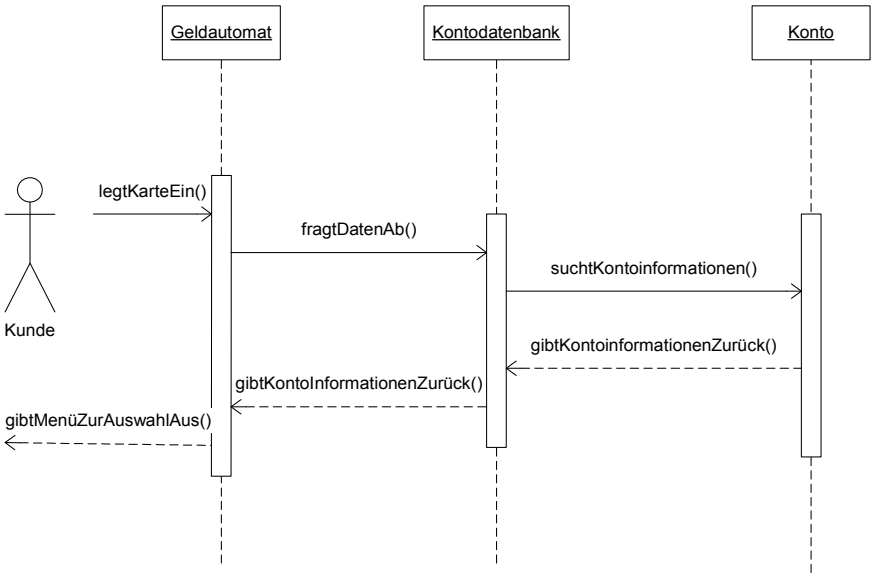
```
System.out.println(k1.toString());
System.out.println(k2.toString());
```

dann wird die jeweils passende toString-Methode verwendet.

Diesen Vorgang nennt man Late Binding (spätes Binden). Das System untersucht das jeweilige Objekt, stellt bei der Referenz k2 fest, dass dieses Objekt aus der Klasse GiroKonto ist, und ruft die passende Methode auf.

Sequenzdiagramm

Das Sequenzdiagramm zeigt den Ablauf einer Methode und berücksichtigt dabei die Interaktion mit Objekten aus anderen Klassen. Das Diagramm kann sicherlich am einfachsten an einem Beispiel dargestellt werden:

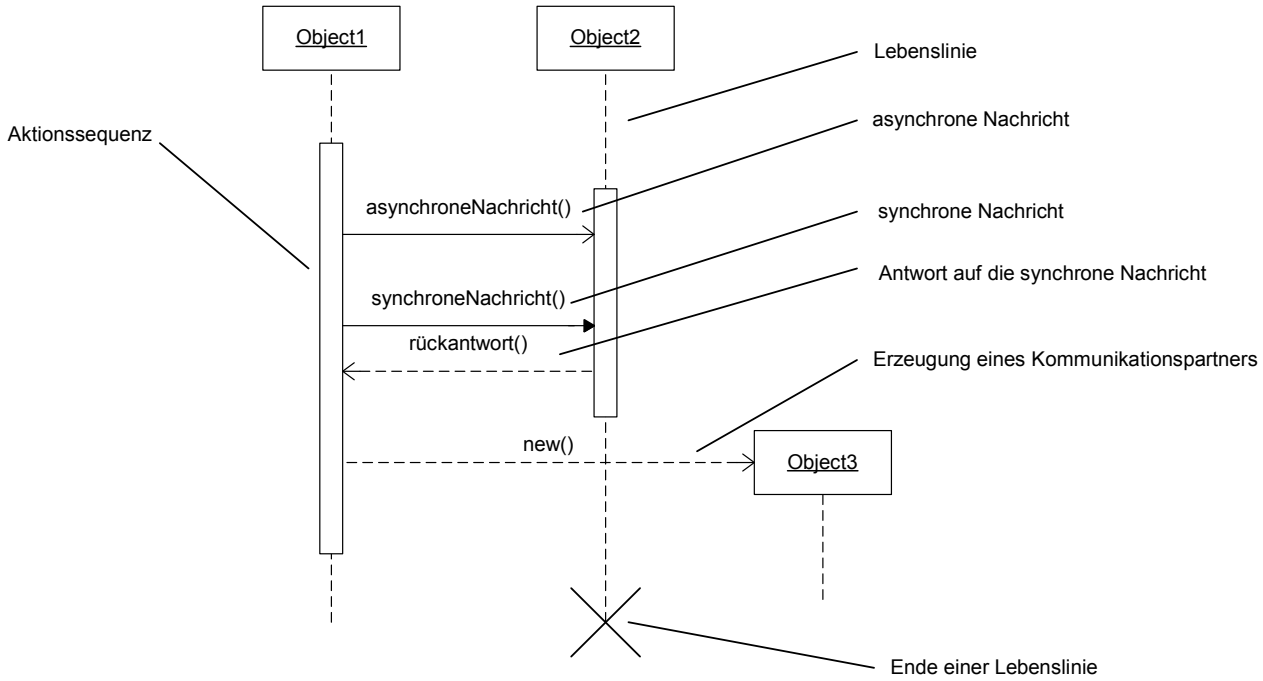


In dem obigen Beispiel legt der Kunde eine Kreditkarte in einen Kontoautomat ein. Anschließend schickt der Automat eine Nachricht an die Datenbank ab und fordert die der Karte entsprechenden Kontoinformationen an. Die Datenbank sucht das entsprechende Konto und holt sich die Informationen. Anschließend gibt die Datenbank die Informationen an den Automaten zurück, welcher dem Kunden ein Auswahlmenü anbietet.

In diesem Diagramm wurde nicht der Fall berücksichtigt, dass irgendein Vorgang nicht funktioniert.

Anmerkung: Die Rückantwort ist im Allgemeinen keine eigene Methode sondern nur der entsprechende Rückgabewert.

Begriffe Teil 1

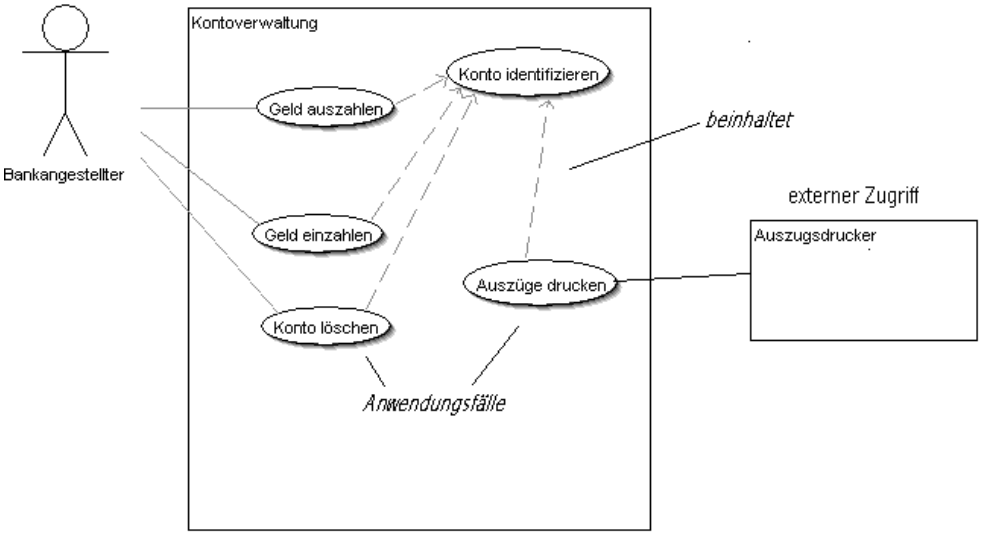


Während der Aktionssequenz kommunizieren die Objekte miteinander. Eine asynchrone Nachricht erfolgt von einem Objekt zu einem anderen und bedarf keiner Rückantwort. Die synchrone verlangt immer nach einer Rückantwort. Die Sequenz läuft immer von oben nach unten ab.

Anmerkung: Die asynchrone Nachricht wird kaum noch verwendet. Wird kein Rückgabewert geliefert, so zeichnet man die gestrichelte Linie trotzdem zurück und ergänzt sie nicht um einen Rückgabewert.

Das Anwendungsfalldiagramm (Use Case)

Das Anwendungsfalldiagramm beschreibt die Schnittstelle zum Programmbenutzer. Es wird festgelegt, welche Aktionen ein Benutzer ausführen kann. Es ist durchaus möglich, dass es Fälle gibt, die in anderen Fällen auch implementiert sind. Auch können externe Benutzer einen Anwendungsfall auslösen. Ein Anwendungsfall wird häufig mit einem eigenen Formular umgesetzt und hat immer ein Sequenzdiagramm zur Folge.



Objekte während der Laufzeit verwalten (Listen)

Listenklassen, können nichts anderes, als andere Objekte bewahren. Im Gegensatz zu den bekannten Feldern aus C++ oder Pascal können die Java – Listen - Klassen beliebige Objekte speichern. Dies funktioniert, da in Java die Oberklasse aller Klassen immer und automatisch die Klasse Objects ist. Somit können auch Objekte aus unterschiedlichen Klassen in einer Liste gespeichert werden. Im Folgenden sollen einige mögliche Listen (bei weitem nicht alle) beschrieben werden.

Die Klasse "Vector"

Die Klasse "Vector" befindet sich im Paket java.util.*, welches zur Benutzung eines Vektors importiert werden muss.

Aus der JAVA - Dokumentation:

- `void add (Object o)`
  Hängt ein Objekt an das Ende der Liste an.
- `Object elementAt (int index)`
  Liefert das Element an der Stelle index.
- `void insertElementAt (Object o, int index)`
  Fügt ein Objekt o an der Stelle index in die Liste ein.
- `void remove (int index)`
  Löscht das Element an der Stelle index.
- `boolean isEmpty()`
  Liefert true zurück, wenn die Liste leer ist.

- `int size(Vector v)`
  Liefert die Anzahl der Elemente, die in der Liste gespeichert sind zurück.

Aufgabenstellung:

Schreiben Sie die Klasse Anwendung so um, dass Sie beim Starten des Programms mehrere Adressen anlegen und anzeigen können.

Das Problem an der oben gestellten Aufgabe ist, dass alle Methodenaufrufe in der main – Methode stattfinden. Dies widerspricht jedoch dem objektorientierten Ansatz. Wir erstellen also eine weitere Klasse, die uns alle Funktionalitäten eines Vectors, aber auch alle von uns erstellten Methoden enthält. Die Klasse muss also ein Vector sein, der erweitert wird, sie erbt also vom Vector alle Attribute und Methoden. Im Folgenden sei der Quelltext dieser Klasse "Adressenliste" abgebildet und erläutert:

```
import  java.util.*;
public class Adressenliste extends Vector
{
 public void Adressenliste ()
 {
  super();
 }

 public void ausgeben()
 {
  int i;
  i = this.size();
  for (int z =0;z<i;z++)
  {
        Adresse temp = (Adresse)this.elementAt(z);
        System.out.println (temp.getVorname()+ " " + temp.getName());
  }
 }

 public void eintragen()
 {
  Adresse temp = new Adresse();
  this.add(temp);
 }
}
```

Im Klassennamen selbst steht das Schlüsselwort extends, welches darauf hinweist, dass diese Klasse von der nach dem Schlüsselwort extends folgenden Klasse erbt. Erbt eine Klasse von einer anderen Klasse, so **muss** im Konstruktor zuerst der Konstruktor der Oberklasse mit dem Schlüsselwort super aufgerufen werden (Über das Schlüsselwort "super" kann man also auf Attribute und Methoden der Oberklasse zugreifen. Über das Schlüsselwort "this" greift man auf das Objekt der aktuellen Klasse zu.).

Aufgabenstellung:

Erweitern Sie die obige Klasse um eine Methode "austragen", bei der man einzelne Adressen aus der Liste entfernen kann.

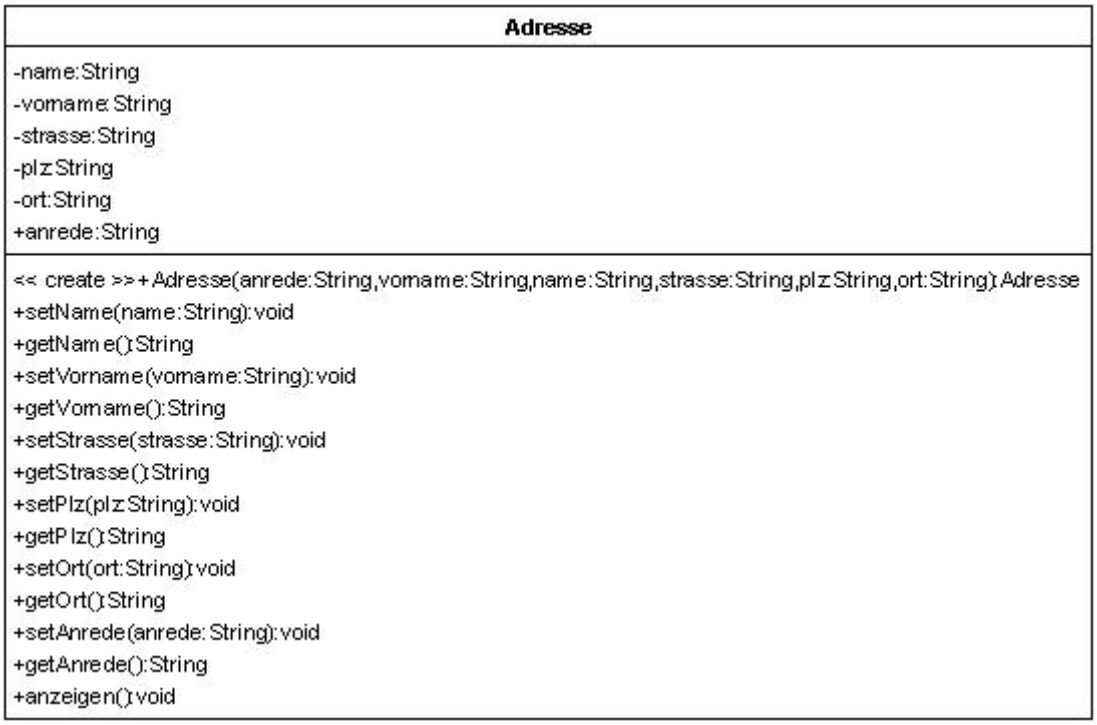
Die Klasse "Stack"

Die Klasse Stack (Stapel) verwaltet Objekte aus der Klasse Object nach dem LIFO – Prinzip (last in first out).

Über die Methode push (Object o) wird das Objekt o in den Stapel geschoben. Über die Methode Object pop () wird das zuletzt in den Stapel geschobene Objekt herausgeholt. Es existiert nun nicht mehr im Stapel! Dies vermeidet die Methoden Objekt peek (). Sie liefert das zuletzt in den Stapel geschobene Objekt, löscht dieses aber nicht im Stapel. Die Methode boolean empty () gibt für den Fall, dass der Stapel leer ist, den Wert true zurück. Ansonsten liefert diese Methode den Wert false.

Beispiel:

In diesem Beispiel werden Objekte aus der Klasse Adresse (diese Klasse wird nur als Klassendiagramm dargestellt!) in einem Stapel während der Laufzeit verwaltet:



Datei MeinStack.java

```
import java.util.*;

public class MeinStack extends Stack
{
        public MeinStack()
        {
                super();
        }

        public void adresseEintragen(Adresse adr)
        {
                this.push (adr);
```

```
        }

        public Adresse adresseHerausholen()
        {
                Object o = this.pop();
                Adresse adr = (Adresse)o;
                return adr;
        }

        public Adresse adresseLaden ()
        {
                Object o = this.peek();
                Adresse adr = (Adresse)o;
                return adr;
        }

        public int anzahlAdressen ()
        {
                return this.size();
        }
}
```

Datei StartMeinStack.java :

```
public class StartMeinStack
{
  public static void main (String args[])
  {
  MeinStack stapel = new MeinStack();

  Adresse adr1 = new Adresse ("Herr","Jochen","Ferger","Altengarten 2", "65558",
"Langenscheid");
  stapel.adresseEintragen (adr1);
  Adresse adr2 = new Adresse ("Frau","Michaela","Ferger","Altengarten 2", "65558",
"Langenscheid");
  stapel.adresseEintragen (adr2);
  Adresse adr3 = new Adresse ("Herr","Paul","Ferger","Altengarten 2", "65558",
"Langenscheid");
  stapel.adresseEintragen (adr3);
  Adresse adr4 = new Adresse ("Frau","Sophia","Ferger","Altengarten 2", "65558",
"Langenscheid");
  stapel.adresseEintragen (adr4);

  System.out.println ("Beispiel fuer pop!");
  do
        {
                Adresse adr = stapel.adresseHerausholen();
                adr.anzeigen();
        }while (!stapel.empty());

  stapel.adresseEintragen (adr4);
  stapel.adresseEintragen (adr3);
  stapel.adresseEintragen (adr2);
  stapel.adresseEintragen (adr1);

  System.out.println ("Es sind " + stapel.anzahlAdressen()+" Adressen im Stack gespeichert");

  System.out.println ("\nBeispiel fuer peek");
```

```
  stapel.adresseLaden().anzeigen();
  do
        {
              Adresse adr = stapel.adresseHerausholen();
              adr.anzeigen();
        }while (!stapel.empty());
  System.out.println ("Es sind " + stapel.anzahlAdressen()+" Adressen im Stack gespeichert");
}// end of main
}// end of class
```

Die Schlange

Das Prinzip einer Schlange (Queue) ist FIFO (first in first out). In JAVA gibt es einige Klassen mit denen man Schlangen realisieren kann. In diesem Beispiel wird die Klasse ArrayBlockingQueue verwendet. Diese stellt unter anderem die Methoden `offer (Object o)` zum Eintragen von Objekten und `Object poll ()` zum Austragen von Objekten aus einer Schlange zur Verfügung. Weitere Methoden können aus der Dokumentation entnommen werden.

Im folgenden Beispiel werden Strings in eine Schlangen geschrieben. Anschließend wird die Schlange nach dem FIFO-Prinzip geleert:

Datei: StringSchlange.java

```
import java.util.concurrent.*;

public class StringSchlange extends ArrayBlockingQueue
{
        public StringSchlange ()
        {
              super (100);
        }

        public void stringEintragen (String text)
        {
              this.offer (text);
        }

        public String stringRausholen ()
        {
              String text = (String)this.poll();
              return text;
        }

        public boolean istLeer()
        {
              if (this.peek() == null)
              return true;
              else
              return false;
        }
}
```

Datei: StartStringSchlange.java

```
public class StartStringSchlange
{
	public static void main (String args[])
	{
		String text1 = "Hans";
		String text2 = "Fritz";
		String text3 = "Paul";
		String text4 = "Axel";
		String text5 = "Matze";
		String text6 = "Jo";
		String text7 = "Uli";

		StringSchlange schlange = new StringSchlange();
		schlange.stringEintragen (text1);
		schlange.stringEintragen (text2);
		schlange.stringEintragen (text3);
		schlange.stringEintragen (text4);
		schlange.stringEintragen (text5);
		schlange.stringEintragen (text6);
		schlange.stringEintragen (text7);

		while (!schlange.istLeer())
		{
			System.out.println (schlange.stringRausholen());
		}
	}
}
```

Die Hashtable

Die Hashtable ist eine Liste in der immer ein Objektpaar aus der Klasse Object gespeichert wird. Hierbei dient das erste Objekt als Schlüssel und das zweite Objekt als Wert. Es werden unter anderem die Methoden `put (Object key, Object value)` und `Object get (Object key)` zur Verfügung gestellt (weitere Methoden entnimmt man aus der Dokumentation). Die Methode put speichert ein Objektpaar in die Tabelle. Die Methode get liefert das passende Objekt zum Objekt key, also dem Schlüssel. Man muss sich also keine Gedanken um Suchalgorhythmen machen. Im folgenden Beispiel werden Vokabeln (deutsch - englisch) in einer Tabelle gespeichert und die Übersetzung einer deutschen Vokabel anschließend gesucht. Hinweis: Die Klasse KonsolenIn ermöglicht das Einlesen eines Wertes über die Tastatur. Hierfür ist das JDK15 Vorraussetzung.

Datei: VokabelListe.java

```
import java.util.*;

public class VokabelListe extends Hashtable
{
	public void vokabelEintragen(String deutsch, String englisch)
	{
		this.put (deutsch,englisch);
	}

	public String vokabelSuchen (String deutsch)
	{
```

```
                Object o = this.get(deutsch);
                if (o == null)
                {
                        return "Vokabel nicht gefunden";
                }

                String englisch = (String)o;
                return englisch;
        }
}
```

Datei: StartVokabelListe.java

```
public class StartVokabelListe
{
        public static void main (String args[])
        {
                KonsoleIn eingabe = new KonsoleIn();
                VokabelListe liste = new VokabelListe();
                liste.vokabelEintragen ("Haus","house");
                liste.vokabelEintragen ("Schule","school");
                liste.vokabelEintragen ("Lehrer","teacher");
                liste.vokabelEintragen ("Schüler","pupil");
                liste.vokabelEintragen ("Klassenarbeit","examination");

                while (true)
                {
                        System.out.print ("Geben Sie das deutsch Wort ein (ende fuer
Programmende):");
                        String deutsch = eingabe.readString();
                        if (deutsch.equals ("ende"))
                        System.exit (0);
                        String englisch = liste.vokabelSuchen (deutsch);
                        System.out.println ("Die Uebersetzung fuer "+deutsch + " ist "+
englisch);
                }
        }
}
```

Felder

Die primitivste Möglichkeit Objekte zu verwalten sind Felder. Die Verwendung von Feldern, die ja bekanntlich nur den gleichen Datentyp aufnehmen können, wird dadurch komfortabler, dass alle Objekte von der Klasse Object erben. Somit wäre es möglich, unterschiedlichste Objekte in einem Feld zuspeichern. Wie in anderen Programmiersprachen muss die Größe des Feldes beim Initialisieren angegeben werden. Die einzelnen Feldelemente sind indiziert, derIndex beginnt mit der Zahl 0 und erhöht sich mit jedem Feldelement um 1.

Felder werden wie folgt angelegt:

```
Object feld [];

feld = new Object [1000];
```

In diesem Beispiel wurde ein Feld mit 1000 Feldelementen angelegt, welche Objekte aus der Klasse Object aufnehmen können.

Im folgenden Beispiel wird eine eigene Klasse AdresseArray geschrieben, die ein Feld für Objekte aus der Klasse Adresse kapselt. Die Klasse Adresse wurde oben schon mehrfach verwendet und wird nicht weiter beschrieben.

Datei: AdresseArray.java

```
public class AdresseArray
{
	Adresse feld[];

	public AdresseArray (int anzahl)
	{
		feld = new Adresse [anzahl];
	}

	public void adresseEintragen (Adresse adr ,int position)
	{
		feld[position] = adr;
	}

	public Adresse adresseLaden(int position)
	{
		return feld[position];
	}

	public void adressenAnzeigen(int anzahl)
	{
		for (int i = 0; i < anzahl; i++)
		{
			System.out.println ("Index: "+i);
			feld[i].anzeigen();
		}
	}
}//end of class
```

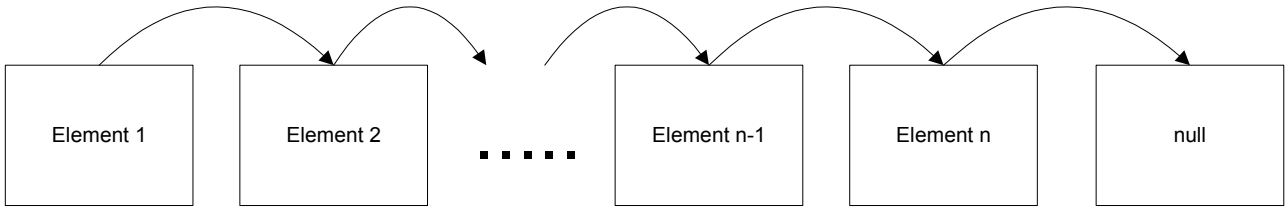
Datei: StartAdresseArray.java

```
public class StartAdresseArray
{
 public static void main (String args[])
 {
  AdresseArray feld = new AdresseArray (100);
  Adresse adr1 = new Adresse ("Herr","Jochen","Ferger","Altengarten 2", "65558",
"Langenscheid");
  feld.adresseEintragen (adr1,0);
  Adresse adr2 = new Adresse ("Frau","Michaela","Ferger","Altengarten 2", "65558",
"Langenscheid");
  feld.adresseEintragen (adr2,1);
  Adresse adr3 = new Adresse ("Herr","Paul","Ferger","Altengarten 2", "65558",
"Langenscheid");
  feld.adresseEintragen (adr3,2);
  Adresse adr4 = new Adresse ("Frau","Sophia","Ferger","Altengarten 2", "65558",
"Langenscheid");
  feld.adresseEintragen (adr4,3);
  feld.adressenAnzeigen(4);
 }
}
```

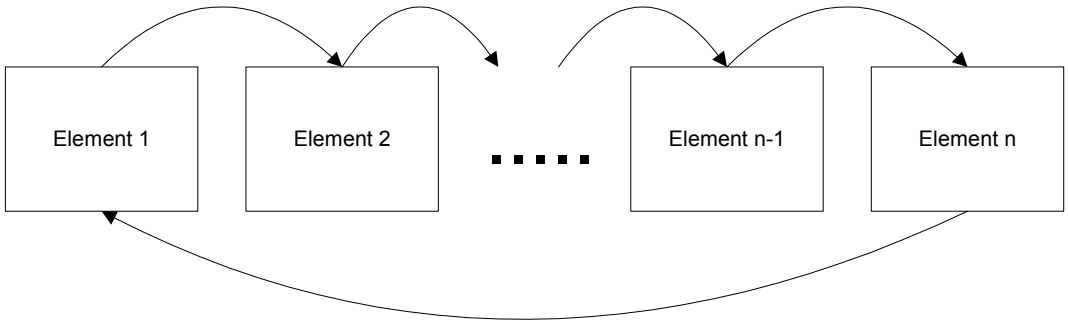
Einfach verkettete Liste

Diese Form der Liste besteht aus einer theoretisch unendlichen Anzahl von Elementen. Jedes Element besitzt einen Datenanteil und einen Zeiger auf das nächste Element der Liste. Das letzte Element zeigt auf einen definierten Wert, z.B. in JAVA auf den Wert null. Wird die Liste als Ring angelegt, so zeigt das letzte Element auf das erste Element der Liste.

Verkettete Liste mit n Elementen:



Verkettete Liste mit n Elementen als Ring:



Im folgenden Beispiel wird eine verkettete Liste realisiert. Der Wertebereich eines Elements enthält ein Objekt aus der Klasse String. Zusätzlich zum Zeiger aus das nächste Element bekommt jedes Element noch ein Attribut boolean erstes zugeordnet. Hiermit ist es möglich, das erste Element zu identifizieren. Diese Maßnahme ist jedoch nicht unbedingt notwendig:

Datei: EinfachVerkettestesElement.java

```
public class EinfachVerkettetesElement
{
	private String inhalt;
	private boolean erstes = false;
	private EinfachVerkettetesElement naechstes = null;

	public EinfachVerkettetesElement (String inhalt, boolean erstes)
	{
		this.inhalt = inhalt;
```

```
		this.erstes = erstes;
	}

	public EinfachVerkettetesElement (String inhalt)
	{
		this.inhalt = inhalt;
	}

	public EinfachVerkettetesElement (String inhalt, EinfachVerkettetesElement element)
	{
			this.inhalt = inhalt;
			this.naechstes = element;
	}
	public boolean istErstes()
	{
		return erstes;
	}

	public boolean istLetztes ()
	{
		if (naechstes == null)
		return true;
		else
		return false;
	}

	public String getInhalt()
	{
		return inhalt;
	}

	public void setNaechstes (EinfachVerkettetesElement element)
	{
		this.naechstes = element;
	}
	public EinfachVerkettetesElement getNaechstes ()
	{
		return naechstes;
	}

	public void elementEinfuegen (EinfachVerkettetesElement vorher)
	{
		this.naechstes = vorher.getNaechstes();
		vorher.setNaechstes(this);
	}
}//end of class
```

Datei: StartEinfachVerketteteListe.java

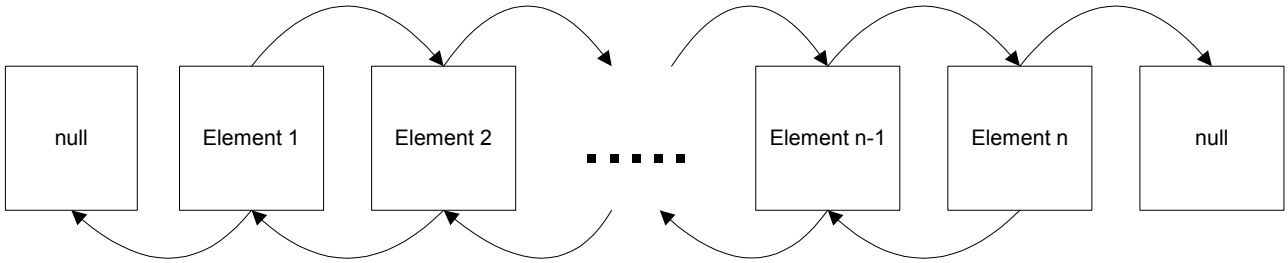
```
public class StartEinfachVerketteteListe
{
	public static void main (String args[])
	{
		EinfachVerkettetesElement element1 = new
EinfachVerkettetesElement("Dies",true);
		EinfachVerkettetesElement element2 = new EinfachVerkettetesElement("ist");
		element1.setNaechstes (element2);
		EinfachVerkettetesElement element3 = new EinfachVerkettetesElement("eine");
```

```
                element2.setNaechstes (element3);
                EinfachVerkettetesElement element4 = new
EinfachVerkettetesElement("verkettete");
                element3.setNaechstes (element4);
                EinfachVerkettetesElement element5 = new EinfachVerkettetesElement("Liste");
                element4.setNaechstes (element5);
                EinfachVerkettetesElement temp = element1;
                do
                {
                        System.out.println (temp.getInhalt());
                        temp = temp.getNaechstes();
                }while (!temp.istLetztes());
                System.out.println (temp.getInhalt());

                EinfachVerkettetesElement element6 = new EinfachVerkettetesElement("tolle");
                element6.elementEinfuegen(element3);

                temp = element1;
                do
                {
                        System.out.println (temp.getInhalt());
                        temp = temp.getNaechstes();
                }while (!temp.istLetztes());
                System.out.println (temp.getInhalt());
        }
}
```

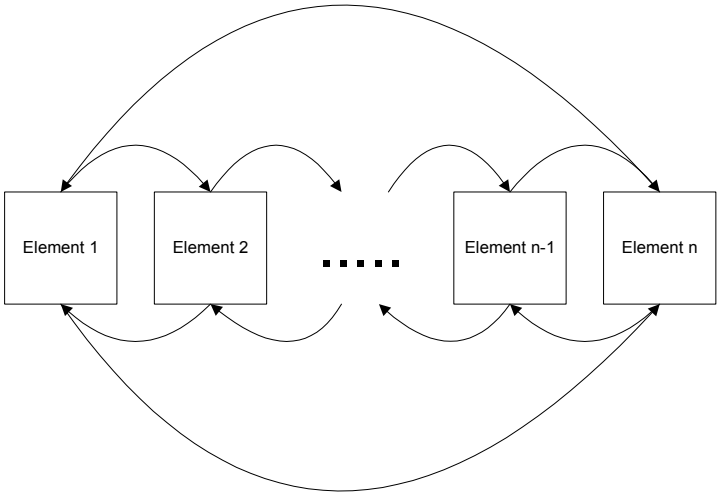
Die doppelt verkettete Liste (oder zweifach verkettete Liste)

Bei dieser Liste besitzt jedes Listenelement zusätzlich zum Zeiger auf das nächste Element einen Zeiger auf das vorherige Element. Ist ein Element das erste der Liste, so zeigt der zusätzliche Zeiger auf den Wert null. Selbstverständlich kann auch eine solche Liste als Ring programmiert werden:

Doppelt verkettete Liste:



Doppelt verkettete Liste als Ring



Programmbeispiel: Doppelt verkettete Liste:

Datei: ListenElement.java

```
public class ListenElement
{
	public static int anzahlElemente;
	public ListenElement vorher= null;
	public ListenElement nachher = null;
	public int elementNummer;

	public ListenElement(int nummer)
	{
		anzahlElemente++;
		this.elementNummer = nummer;
	}

	public ListenElement(ListenElement vorher,ListenElement nachher, int nummer)
	{
		anzahlElemente++;
		this.elementNummer = nummer;
		this.vorher = vorher;
		this.nachher = nachher;
	}

	public void setVorher(ListenElement vorher)
	{
			this.vorher= vorher;
	}

	public ListenElement getVorher()
	{
		return vorher;
	}

	public void setNachher(ListenElement nachher)
	{
			this.nachher = nachher;
	}
```

```
        public ListenElement getNachher()
        {
                return nachher;
        }

        public void setElementNummer (int i)
        {
                elementNummer = i;
        }

        public void elementNummerErhoehen()
        {
                elementNummer ++;
        }

        public void elementNummerErniedrigen ()
        {
                elementNummer --;
        }

        public void anzeigen ()
        {
                System.out.println();
                System.out.println ("Dies ist das "+elementNummer+ ".te Element" );
                System.out.println ("Es gibt "+anzahlElemente + " Elemnt(e) in der Liste");
        }
        public void vorherAnzeigen ()
        {
                System.out.println();
                System.out.println("Mein Vorgaenger:");
                if (vorher == null)
                System.out.println ("Ich bin das erste Element in der Liste");
                else
                this.vorher.anzeigen();
        }

        public void nachherAnzeigen()
        {
                System.out.println();
                System.out.println("Mein Nachfolger:");
                if (nachher == null)
                System.out.println ("Ich bin das letze Element in der Liste");
                else
                this.nachher.anzeigen();
        }
}
```

Datei: StartDoppeltVerketteteListe.java

```
public class StartDoppelVerketteteListe
{
        public static void main (String args[])
        {
                ListenElement element1 = new ListenElement (1);
                element1.anzeigen();
                element1.vorherAnzeigen();
                element1.nachherAnzeigen();
```

```
                ListenElement element3 = new ListenElement (element1,null,3);
                element3.anzeigen();
                element3.vorherAnzeigen();
                element3.nachherAnzeigen();

                ListenElement element2 = new ListenElement (element1,element3,2);
                element3.setVorher(element2);
                element2.anzeigen ();
                element2.vorherAnzeigen();
                element2.nachherAnzeigen();
        }
}
```

Sonderformen:

Als Sonderformen kann die sortierte verkettete Liste benannt werden, bei der die einfügen – Methode angepasst wird. Auch ein Ringpufferspeicher kann mit einer Liste oder aber auch mit einem Feld realisiert werden. Hier bietet es sich an, die mitgelieferten Klassen in der Dokumentation zu betrachten. Sie finden sich alle in oder unter dem Paket java.util.*!

Objekte identifizieren

Wenn man nun Objekte während der Laufzeit in Listen verwaltet, kann es zudem Problem kommen, dass Objekte unterschiedlicher Klassen in der gleichen Liste gespeichert sind. Um herauszufinden, welche Objekte zu welcher Klasse gehören, kann man sich über die Methode `Class getClass ()` ein Objekt aus der Klasse Class erzeugen. Diese Klasse besitzt u.a. die Methode `String getName()`, welche den Namen der Klasse zurückgibt und die Methode `boolean isInstance(Object o)`, die den Wert true liefert, sofern o aus der entsprechenden Klasse ist.

Beispiel (Die Klasse ist die obige, zusätzlich wurde ein Standardkonstruktor implementiert):

Datei: ObjekteIdentifizieren.java

```
public class ObjekteIdentifizieren
{
        public static void main (String args[])
        {
                Adresse adr= new Adresse();
                Class klasse= adr.getClass();
                String strKlasse = klasse.getName();
                System.out.println ("adr ist aus der Klasse: "+strKlasse);

                if (klasse.isInstance (adr))
                {
                        System.out.println ("Abfrage geht auch mit if!");
                }
        }
}
```

Der folgende Bildschirmplot wurde erzeugt:

Textdateien

Text in eine Datei schreiben:

Das Paket java.io.* liefert alle Klassen, um in eine Textdatei zu schreiben bzw. um aus ihr zu lesen. Um diese Klassen zu verwenden, muss die Bibliothek eingebunden, importiert werden:

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

Es wäre auch möglich, jede benötigte Klasse einzeln einzubinden. Gute Programmierer machen dies um genau anzugeben, welche Klassen aus welchem Paket sie verwenden (siehe auskommentierten Quellcode im Programm). Zur Vereinfachung wird aber das komplette Paket eingebunden.

Das Speichern in eine Textdatei funktioniert über Writer. Mittels eines FileWriter wird die Verbindung zu einer Textdatei hergestellt:

```
FileWriter fw = new FileWriter("test.txt");
```

Ein BufferedWriter hängt sich an den FileWriter und sorgt dafür, dass die Datenausgabe kontrolliert (gepuffert) stattfindet:

```
BufferedWriter bw = new BufferedWriter(fw);
```

Der PrintWriter hängt sich an den BufferedWriter und stellt Methoden zur Verfügung, Daten zeilenweise in die Datei zu schreiben:

```
pWriter = new PrintWriter(bw);
pWriter.println("Dies ist ein Text2");
```

Da das Schreiben in eine Datei ein Vorgang ist, der unsicher sein könnte (jemand könnte beim Schreiben diese Datei löschen), muss der komplette Vorgang in einen try-Catch-Block integriert werden (hierzu mehr im Kapitel try-catch-finally).

Achtung:

Existiert die Datei schon, so wird ihr ursprünglicher Inhalt gnadenlos überschrieben. Dies könnte man verhindern, indem man beim Erstellen des FileWriters einen boolschen Wert true mit übergibt. Dann werden die neuen Daten an das Ende der Datei angehängt:

```
FileWriter fw = new FileWriter("test.txt", true);
```

Hier nun das komplette Programm, um einen Text in eine Datei zu schreiben.

```
/*import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
*/

import java.io.*;

public class InTextdateiSchreiben {
  public static void main(String[] args) {
    PrintWriter pWriter = null;
    try {
```

```
      FileWriter fw = new FileWriter("test.txt");
      BufferedWriter bw = new BufferedWriter(fw);
      pWriter = new PrintWriter(bw);

      //pWriter = new PrintWriter(new BufferedWriter(new FileWriter("test.txt")));
      pWriter.println("Dies ist ein Text2");
      pWriter.println("Dies ist ein Text3");
    } catch (IOException ioe) {
      ioe.printStackTrace();
    } finally {
      if (pWriter != null){
        pWriter.flush();
        pWriter.close();
      }
    }
```

Text aus einer Datei lesen:

Analog zu den Writern gibt es Reader. Der FileReader hängt sich wieder an eine Textdatei. Ein BufferedReader hängt sich an den FileReader und stellt gleichzeitig schon Methoden zum zeilenweisen Einlesen von Strings zur Verfügung:

```
        BufferedReader in = new BufferedReader(new FileReader("test.txt"));
        String zeile = in.readLine();
```

Funktioniert das Einlesen einer Zeile nicht, so ist der Rückgabewert null. Dies ist beispilesweise der Fall, wenn das Ende der Textdatei erreicht wurde. Das macht man sich zu Nutze, indem man in einer kopfgesteuerten Schleife so lange Zeilen einliest, bis der Wert null erreicht wurde:

```
        while ((zeile = in.readLine()) != null) {
          System.out.println("Gelesene Zeile: " + zeile);
        }
```

Da auch hier wieder das Schreiben in eine Textdatei unsicher ist, wird der Quellcode in einen try-catch-Block eingefasst.

Hier nun der komplette Quelltext des Programms, welches Text aus einer Datei einliest und auf dem Bildschirm ausgibt:

```
import java.io.*;

public class VonTextdateiLesen {

  public static void main(String[] args) {

    try {
        BufferedReader in = new BufferedReader(new FileReader("test.txt"));
        String zeile = null;
        while ((zeile = in.readLine()) != null) {
          System.out.println("Gelesene Zeile: " + zeile);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
  } // end of main

} // end of class VonTextdateiLesen
  }
}
```

Datum : Joda-Time

Java bietet natürlich Möglichkeiten, mit dem Datum und der Zeit zu arbeiten. Beachtet man aber das Open-Source-Paket Joda-Time (http://joda.sourceforge.net/), so stellt man fest, dass der Umgang mit Datum und Zeit wesentlich vereinfacht wird.

Installation von Joda-Time

Man lädt das aktuelle Paket von Joda-Time von der obigen Webseite herunter. Mittels eines Pack-Programms (7Zip o.ä.) entpackt man die Datei, bis man im Ordner eine Datei names: joda-time-2.7.jar findet (die Versionsnummer kann natürlich differieren). Diese Datei kopiert man in das Verzeichnis der externen Pakete von Java. Dieses findet man unter C:\Program Files\Java\jdk1.8.0_40\jre\lib\ext. Hier kopiert man die .jar-Datei rein (der Programmpfad ist natürlich auf Ihre Umgebung anzupassen). Fertig!

Will man in der entsprechenden Entwicklungsumgebung auch die Dokumentation verwenden, so entpackt man die Datei joda-time-2.7-javadoc.jar. Hier findet man den Ordenr „joda“ unter dem Ordner „org“. Diesen kopiert man im jdk in den Ordner org (docs/api/org).

Verwendung von Joda-Time

Im nachfolgenden Beispielprogramm werden einige (es gibt noch viel mehr) Funktionen von der Verwendung mit Joda-Time dargestellt:

Zu Beginn muss die entsprechende Bibliothek eingebunden werden:

```
import org.joda.time.*;
```

Ein Objekt der Klasse DateTime wird erzeugt. Dieses Objekt enthält immer die aktuellen Informationen und aktualisiert sich selbst!

```
    DateTime datum = DateTime.now();
```

Der aktuelle Tag wird als Integerzahl ausgegeben:

```
    System.out.println("Tag");
    System.out.println(datum.dayOfMonth().getAsText());
```

Der aktuelle Tag wird als Text ausgegeben:

```
    System.out.println("Wochentag");
    System.out.println(datum.dayOfWeek().getAsText());
```

Der aktuelle Tag wird als Integerzahl ausgegeben:

```
    System.out.println("Monat:");
    System.out.println(datum.getMonthOfYear());
```

Der aktuelle Tag wird als Text ausgegeben:

```
    System.out.println("Monat:");
    System.out.println(datum.monthOfYear().getAsText());
```

Das aktuelle Jahr wird als Integerzahl ausgegeben:

```
    System.out.println("Jahr:");
    System.out.println(datum.getYear());
```

Die aktuelle Stunde wird als Integerzahl ausgegeben:

```
System.out.println("Stunden:");
System.out.println(datum.getHourOfDay());
```

Die aktuelle Minutenzahl wird als Integerzahl ausgegeben:

```
System.out.println("Minuten");
System.out.println(datum.getMinuteOfHour());
```

Ein neues Objekt der Klasse DateTime wird erzeugt. Das Datum wird als String übergeben:

```
DateTime naechstesWeihnachten = new DateTime("2015-12-24");
```

Die Differenz von heute bis zu dem Datum wird in Monaten angezeigt (Integerzahl):

```
System.out.println("Monate bis Weihnachten");
Period p = new Period(datum, naechstesWeihnachten, PeriodType.yearMonthDay());
System.out.println( p.getMonths());
```

Die Differenz von heute bis zu dem Datum wird in Tagen angezeigt (Integerzahl):

```
System.out.println("Tage bis Weihnachten");
p = new Period(datum, naechstesWeihnachten, PeriodType.yearDay());
System.out.println( p.getDays());
```

Die Differenz von heute bis zu dem Datum wird in Jahren angezeigt (Integerzahl):

```
System.out.println("Jahre bis Weihnachten");
p = new Period(datum, naechstesWeihnachten, PeriodType.yearMonthDay());
System.out.println(p.getYears());
```

Die Differenz von dem Datum bis heute wird in Jahren angezeigt (Integerzahl):

```
System.out.println("Tage vom naechsten Weihnachten bis heute!");
p = new Period(naechstesWeihnachten,datum, PeriodType.yearDay());
System.out.println(p.getDays());
```

Exceptions

Java ist eine Sprache, die sehr viele Sicherheitsvorkehrungen vorsieht. Ein Sicherheitskonzept sind die Exceptions. Exceptions sind Fehler, die im Programmablauf auftreten können, die man jedoch abfangen kann. So ist es in C z.B. nicht (oder nur mit viel Aufwand) möglich, auf der Konsolenebene statt eines Strings eine Integerzahl einzugeben, und diese Falscheingabe abzufangen. Das C - Programm schmiert im schlimmsten Falle ab. In Java können solche Probleme mit der folgenden Struktur abgefangen werden:

```
try
{}   //In diesem Block steht der gefährdete Code
catch (Exception e)
```

```
{}   //In diesem Block steht der Code für den Fehlerfall

finally

{}   //Dieser Block wird auf jeden Fall ausgeführt
```

Die finally – Anweisung ist nicht unbedingt notwendig. Im folgenden Beispiel soll gezeigt werden, wie die try – catch – Kombination zu verwenden ist. Es geht hierbei um die Division durch 0.

```
int a,b=0,c;

try

{  c = a/b; }

catch (ArithmeticException e)

{System.out.println ("Fehler");}
```

In diesem Programmbeispiel wird die Division durch 0 abgefangen und der Text "Fehler" ausgegeben. Von den Exceptions gibt es sehr viele. In der Dokumentation findet man die einzelnen Ausnahmen (Exceptions), die alle von der Hauptausnahme "Exception" erben.

Will man beim Programmieren erst einmal keine Rücksicht auf Exceptions nehmen, so kann man Methoden mit dem Schlüsselwort throws und anschließend den Namen der Exception im Methodenkopf versehen. Die entsprechende Exception ist dann bei der Verwendung der Methode abzufangen.

Das folgende Beispiel zeigt, wie mit dem Schlüsselwort throws umzugehen ist. Führt man das Programm aus, so wird bei dem markierten Aufruf eine Exception ausgelöst, da f nicht in eine Zahl umgewandelt werden kann!

Datei: BeispielException.java

```
public class BeispielException
{
 public void warten () throws Exception
 {
       Thread.sleep (1000);
 }
 public void umwandeln (String zahl)
 {
  System.out.println ("Die Zahl ins Quadrat ist: " + Integer.valueOf (zahl)*Integer.valueOf
(zahl));
 }
}//end of class
```

Datei: StartBeispielException.java

```
public class StartBeispielException
{
	public static void main (String args[])
	{
		BeispielException b = new BeispielException();
		try
		{
		b.warten ();
		}
		catch (Exception e)
		{
			System.out.println ("Fehler in der Methode warten()");
		}
		b.umwandeln ("2");
		b.umwandeln ("f"); //löst eine Exception aus
	}
} //end of class
```

Threads

Bei den unterschiedlichen Anforderungen an Software ist es oft notwendig, mehrere Abläufe parallel laufen zulassen. Z.B. muss ein Webserver mehrere Clientanfragen gleichzeitig beantworten oder es ist notwendig, dass ein Prozess im Hintergrund eine Hardwareschnittstelle beobachtet um Zustandsänderungen an das Hauptprogramm zu melden. Hierzu dienen Threads. Threads sind "kleinere Prozesse", die sich den gleichen Adressraum (nämlich den vom Hauptprozess, der die Threads ausführt) teilen[7]. Somit können Threads auch auf die gleichen Variablen zugreifen! Jeder Thread hat seinen eigenen Befehlszähler und seinen eigenen Stack[8].

Threads per Vererbung

Das Ausführen mehrerer parallel laufender Prozesse kann in JAVA über die Klasse Thread realisiert werden. Eine Klasse MeinThread erbt von Thread und überschreibt die Methode `void run()`. Nun können von einer main – Methode (oder von einer anderen während der Laufzeit) mehrere Instanzen erzeugt werden. Diese Instanzen werden über den Aufruf der Methode `start()` "gestartet", wodurch automatisch die Methode `void run()` aufgerufen wird. Beispiel:

Datei: MeinThread.java

```
public class MeinThread extends Thread
{
	String text;

	public MeinThread (String text)
	{
		super();
		this.text = text;
	}
```

[7] aus Krüger, Seite 442

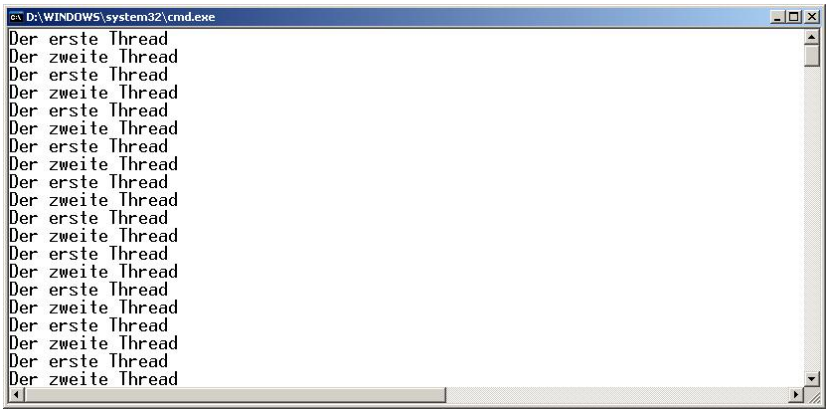
[8] Threads sind eigentlich viel umfangreicher und interessanter, jedoch würden noch mehr Informationen zu diesem Thema fast ein eigenes Skript nach sich ziehen!

```
        public void run ()
        {
                while (true)
                {
                        System.out.println (text);
                        try
                                {
                                        Thread.sleep (100);
                                }
                        catch (Exception e)
                                {}
                }
        }
}
```

Datei: StartMeinThread.java

```
public class StartMeinThread
{
        public static void main (String args[])
        {
                MeinThread thread1 = new MeinThread ("Der erste Thread");
                MeinThread thread2 = new MeinThread ("Der zweite Thread");
                thread1.start();
                thread2.start();
                try
                {
                        Thread.sleep (5000);
                }
                catch (Exception e)
                {}
                thread1.stop();
                thread2.stop();
        }
}
```

Die Ausführung der Datei StartMeinThread.java hat folgende Bildschirmausgabe zur Folge:



Threads per Interface

Über die Implementierung des Interfaces Runnable wird der gleiche Effekt erzeugt wie über die Vererbung der Klasse Thread. Hier muss nun in der Klasse, die Runnable implementiert die Methode `void run()` überschrieben werden:

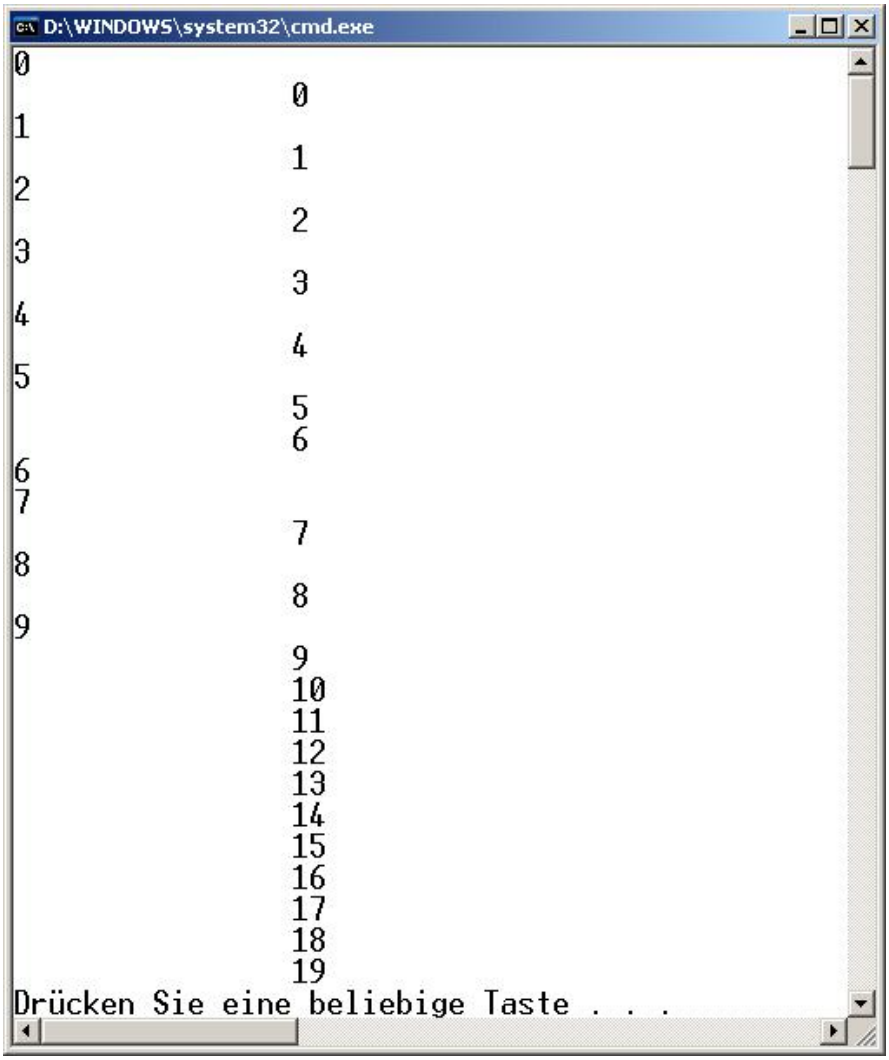
Datei: MeinThread.java

```
public class MeinThread implements Runnable
{
	int zahl,tabstop;

	public MeinThread (int zahl, int tabstop)
	{
		this.zahl = zahl;
		this.tabstop = tabstop;
	}

	public void run ()
	{
		for (int i = 0; i < zahl; i++)
		{
			for (int t = 0; t < tabstop; t++)
			{
				System.out.print ("\t");
			}

			System.out.println (i);
			try
				{
					Thread.sleep (100);
				}
			catch (Exception e)
				{}
		}
	}
}//end of class
```

Datei StartMeinThread.java

```
public class StartMeinThread
{
	public static void main (String args[])
	{
		MeinThread thread1 = new MeinThread (10,0);
		MeinThread thread2 = new MeinThread (20,2);
		Thread t1 = new Thread (thread1);
		Thread t2 = new Thread (thread2);
		t1.start();
		t2.start();
		try
		{
			Thread.sleep (2000);
		}
		catch (Exception e)
		{}
	}
}
```

Die folgende Bildschirmausgabe wird erzeugt:



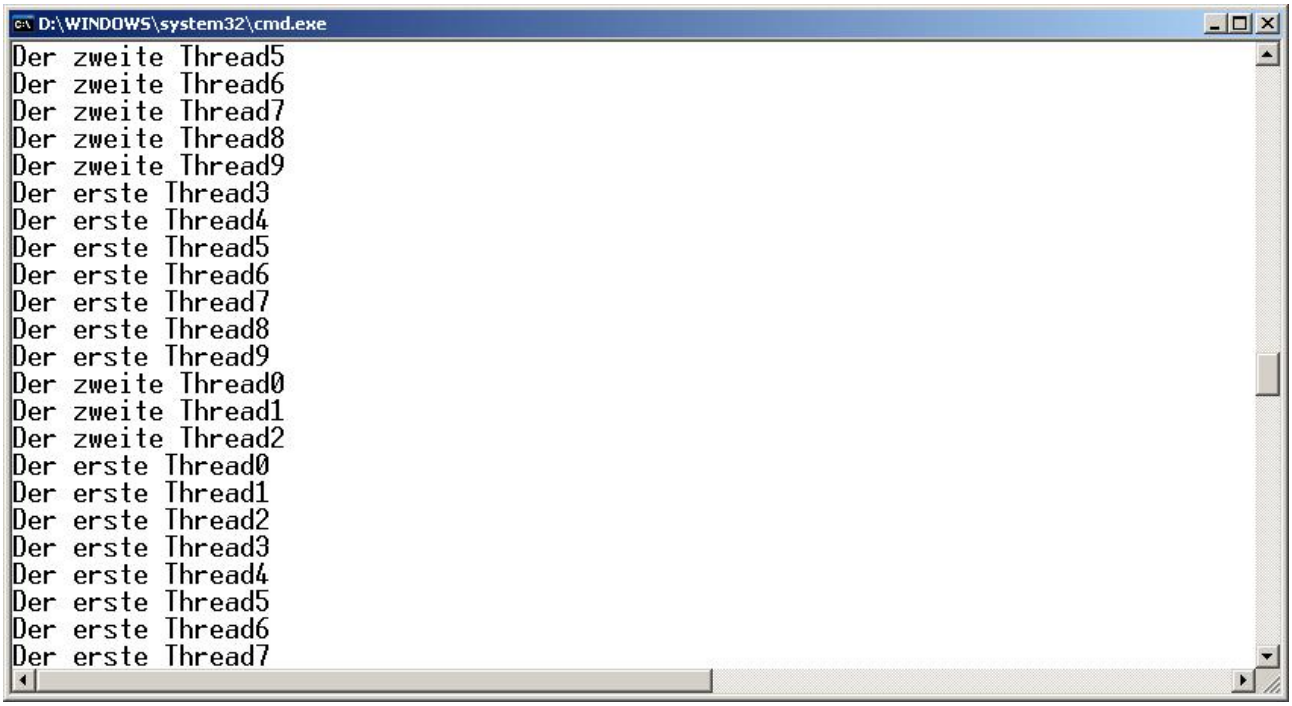
Abläufe Synchronisieren

Angenommen, in einem Thread wird von 0 bis 10 gezählt und der entsprechende Zählwert wird auf der Konsole ausgegeben. Dies kann durch die folgende Klasse realisiert werden:

```
public class MeinThreadUnSynchron extends Thread
{
	String text;

	public MeinThreadUnSynchron (String text)
	{
		super();
		this.text = text;
	}

	public void zaehlen()
	{
		for (int z = 0; z < 10; z++)
```

```
                {
                        System.out.println (text + z);
                }
        }

        public void run ()
        {
                for (int i =0; i < 10; i++)
                {
                        zaehlen();
                        try
                        {
                                Thread.sleep(50);
                        }
                        catch (Exception e){}
                }
        }
}
```

Werden nun zwei Instanzen dieser Klasse erzeugt und die beiden Threads mit der Methode `start()` gestartet, so kann es zu folgender Ausgabe kommen[9]:



Man sieht, je nach zugeteilter Prozessorzeit, gibt der eine oder der andere Thread seine Ausgaben auf der Konsole aus. Dies ist unter Umständen unerwünscht (z.B. bei bidirektionalen Kommunikationsprozessen). Zu lösen ist dieses Problem mit der Markierung eines Blockes oder einer ganzen Methode durch das Schlüsselwort `synchronized`.

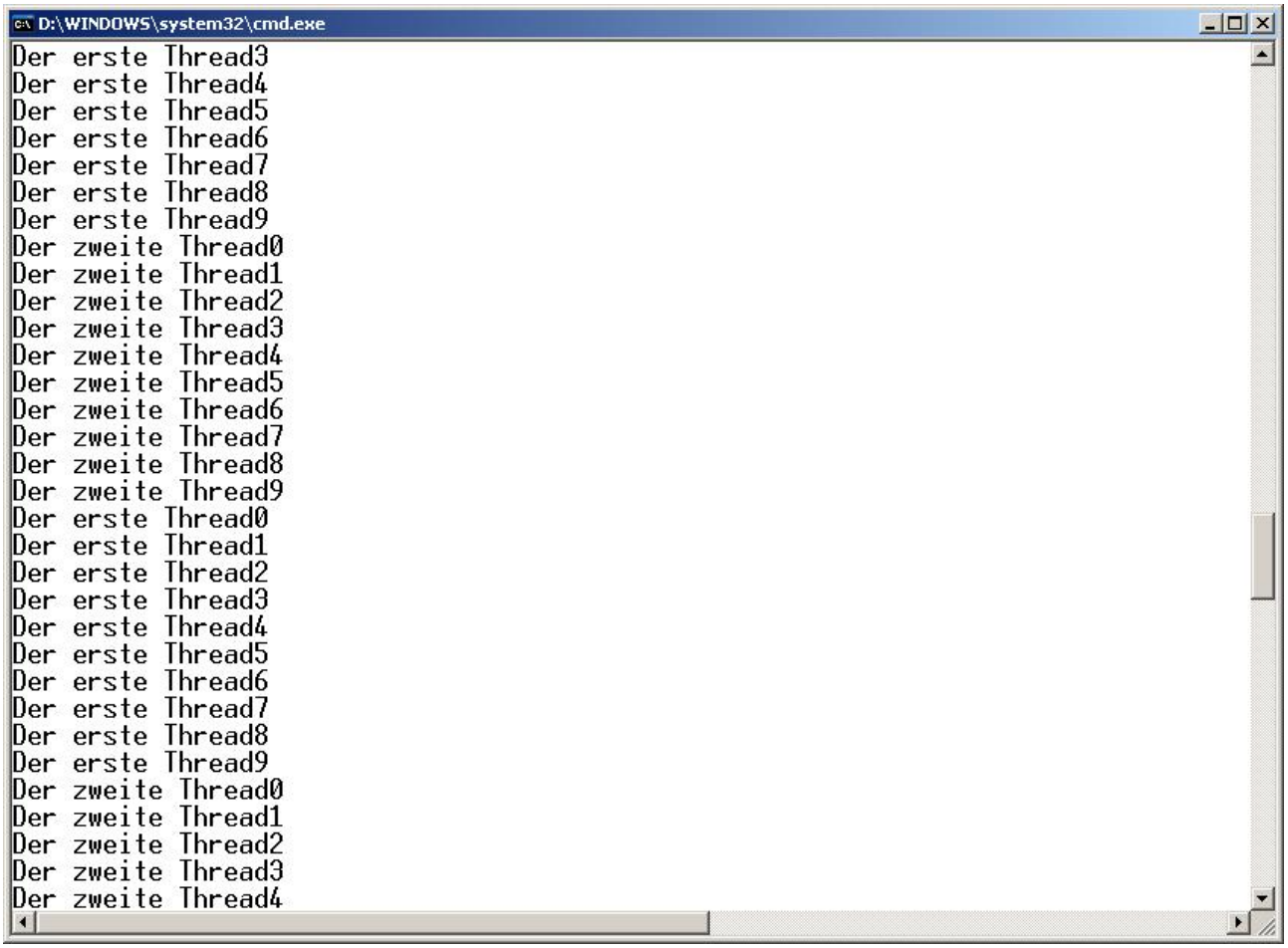
Die folgende Klasse nutzt diese Markierung:

```
public class MeinThreadSynchron extends Thread
{
        String text;
```

[9] Programmbeispiel zu finden unter Multithreading/UnsynchronisierteThreads

```
        public MeinThreadSynchron (String text)
        {
                super();
                this.text = text;
        }

        public void zaehlen()
        {
                for (int z = 0; z < 10; z++)
                {
                        System.out.println (text + z);
                }
        }

        public void run ()
        {
                for (int i =0; i < 10; i++)
                {
                        synchronized (getClass())
                        {
                        zaehlen();
                        try
                        {
                                Thread.sleep(50);
                        }
                        catch (Exception e){}
                        }
                }
        }
}
```

Es kommt zu folgender Bildschirmausgabe:



Man sieht, die Threads werden sauber nacheinander ausgeführt. Sie sind synchronisiert. Der synchrone Block wurde mit dem Schlüsselwort synchronized eingeleitet und mit geschweiften Klammern begrenzt. Dem Schlüsselwort synchronized wird in runden Klammern ein Objekt übergeben, welches beim Ablauf des Blockes geschützt wird. Es hilft hier nichts, this in die runden Klammern zu schreiben, da sich this immer auf das aktuelle Objekt bezieht, und davon gibt es im obigen Fall zwei! Mit getClass() wird ein Klassenobjekt beschafft, welches für beide Threads identisch ist.

Microsleep

Angenommen, in einem Programm muss ein Thread ständig den Zustand einer Schnittstelle abfragen (Polling), da ein externes Gerät an dieser Schnittstelle nicht in der Lage ist, eigenständig Daten zu senden. Würde man diese Abfrage in eine Endlosschleife in einem Thread auslagern, kommt es zu folgendem Effekt: Die Endlosschleife läuft (wie der Name schon sagt) pausenlos und belastet den Prozessor zu 100%. Im folgenden kleinen Programm wird dies simuliert:

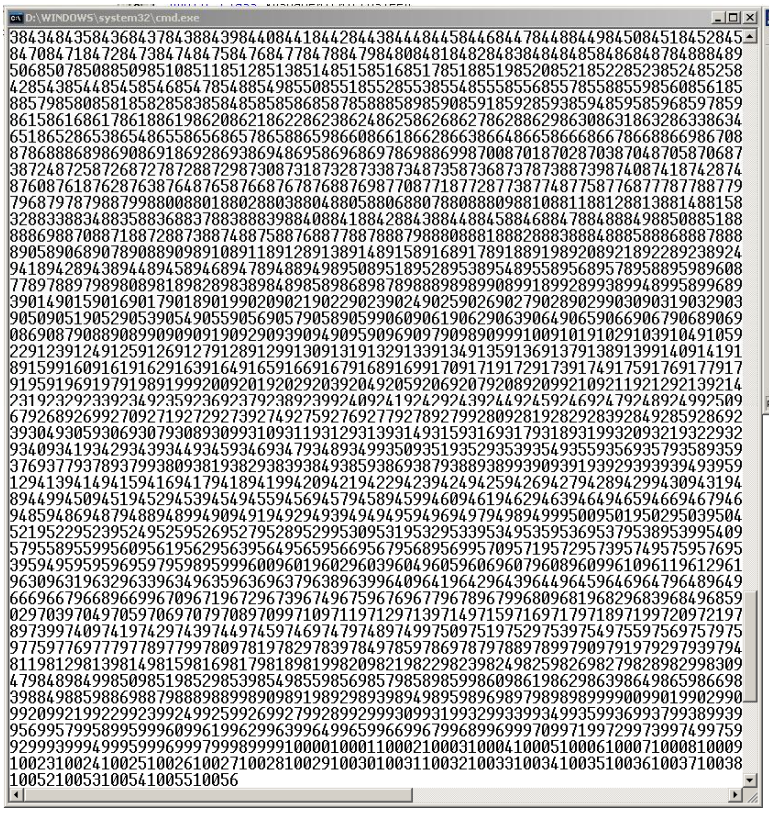
```
public class AusgabeOhneMicrosleep
{
	public static void main (String args[])
	{
		int i = 0;
		while (true)
		{
```

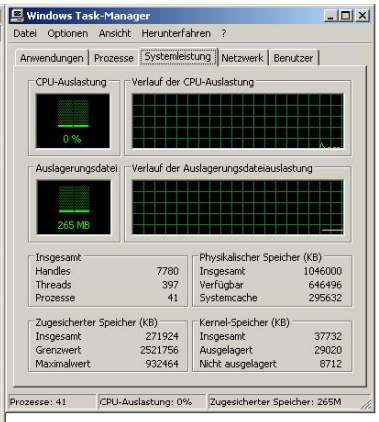
```
                System.out.print (i++);
            }
        }
}
```

Das Programm hat die folgende Bildschirmausgabe zur Folge (gleichzeitig wird der Taskmanager angezeigt):

Man sieht, dieses kleine Programm nimmt dem System fast die komplette Leistung des Prozessors weg.

Mit einem kleinen Trick kann dies verhindert werden. Durch das Einfügen einer minimalen Unterbrechung (hierbei beträgt die Unterbrechungszeit laut Dokumentation eine Nanosekunde, wobei der tatsächliche Zeitwert sicherlich höher sein wird) des Threads wird die Endlosschleife unmerklich verlangsamt. Für das gesamte System ist jedoch diese Unterbrechung ausreichend um wieder vollkommen stabil und performant zu laufen (siehe Screenshot).





Die Unterbrechung wird durch die Codezeilen:

```
try
{
 Thread.sleep (0,1);
}catch (Exception e){}
```

erreicht. Die übergebenen Werte der statischen Methode sleep() stehen für die Unterbrechungszeit in Millisekunden und Nanosekunden.

GUI

Die Grundlage eines jeden Programms mit grafischer Oberfläche ist ein Fenster, in JAVA Frame genannt. Seit dem Swing-Paket gibt es auch das JFrame, welches schicker aussieht und doch einige Funktionen mehr bietet.

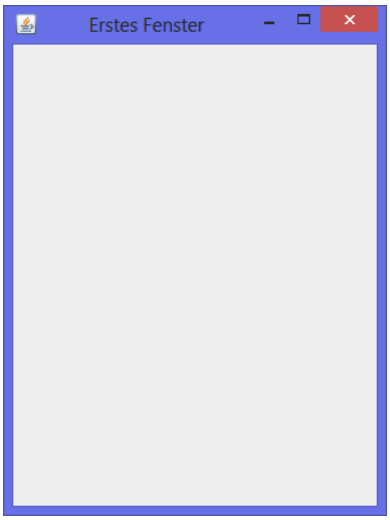
Um also ein Fenster zu erstellen, implementiert man eine eigene Klasse und lässt diese von JFrame erben. Das JFrame bringt jede Menge Methoden mit, deren man sich wegen der Vererbung einfach bedienen kann. Hinweis: Um das JFrame verwenden zu können, muss man das Paket javax.swing.* einbinden (alternativ kann man auch nur das javax.swing.JFrame importieren).

Im nachfolgenden Beispiel wird ein einfaches Fenster mit einer festen Fenstergröße erzeugt und angezeigt:

```
//einbinden des benötigten Paketes
import javax.swing.*;

//Die Klasse Fenster erbt von JFrame
public class Fenster extends JFrame{

  public Fenster(String titel){
```

```
//Der Konstruktor der Oberklasse wird aufgerufen
    super(titel);
//Die Größe des Fensters wird auf eine Breite von 300 Pixel und eine
//Tiefe von 400 Pixel gesetzt
    this.setSize(300,400);
//Das Fenster wird angezeigt
    this.setVisible(true);
  }

  public static void main (String args[]){
 //Ein neues Objekt aus der Klasse Fenster wird erzeugt
   new Fenster("Erstes Fenster");
  }
}
```

Das Programm hat die folgende Ausgabe zur Folge:



Führt man das Programm aus, und schließt das Fenster so beendet die virtuelle Maschine zwar das Fenster, aber das Programm läuft trotzdem „irgendwo“ noch weiter (im JAVA-Editor muss man beispielsweise über den roten Button das Programm beenden). Dies liegt daran, dass mehrere Prozesse beim Anzeigen eines Fensters ablaufen. Um alle Prozesse wirklich zu beenden, bekommt die Klasse einen Hinweis, was sie machen soll, wenn man das Fenster schließt:

```
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Aufgabenstellung:

Erstellen Sie ein Fenster der Größe 800 x 600 Pixel. Sorgen Sie dafür, dass das Fenster in seiner Größe nicht verändert werden kann.

Komponenten

Möchte man ein Fenster mit Leben füllen, so muss man Komponenten wie Text, Texteingabefelder, Buttons etc. darauf platzieren.

Hier gibt es im Grunde zwei Möglichkeiten:

Man platziert die einzelnen Komponenten pixelgenau aufs Fenster. Anschließend sorgt man noch dafür, dass diese Platzierung vom Fenster bzw. auch vom Anwender nicht verändert werden kann. Diese Möglichkeit wird hier behandelt.

Die andere Variante ist die Verwendung von Layoutmanagern, hierzu später mehr.

Da alle grafischen Komponenten von der gleichen Oberklasse erben, ist die Verwendung identisch.

Eine grafische Komponente wird durch einen Konstruktor erzeugt, beispielsweise beim Button:

```
JButton b = new JButton(„Text auf dem Button“);
```

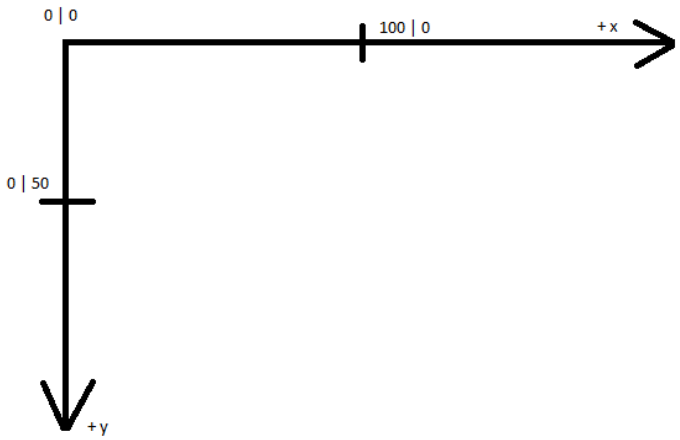
Anschließend setzt man die Position der linken oberen Ecke des Buttons, sowie seine Breite und seine Tiefe:

```
b.setBounds(10,20,100,30);
```

Der Button sitzt nun auf der Position 10,20, hat eine Breite von 100 Pixeln und eine Tiefe von 30 Pixeln.

Hinweis:

Das Koordinatensystem der Pixel auf dem Frame ist wie folgt angeordnet:



Man bemerke, dass die y-Koordinate nach unten hin positivere Werte erhält.

Letztlich muss die Komponente dem Frame (hier das Objekt f) zugeordnet werden:

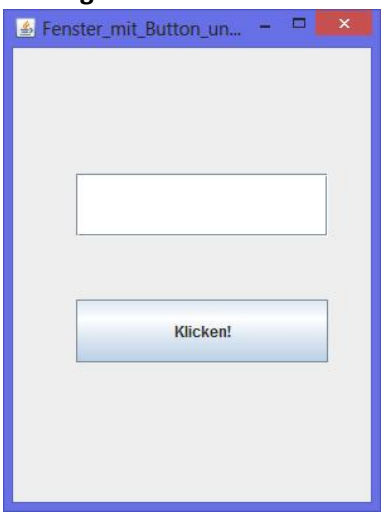
```
f.add(b);
```

Im nachfolgenden Beispiel wird ein Fenster mit einem Textfeld und einem Button erzeugt.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Fenster_mit_Button_und_Textfeld extends JFrame{
  JButton b;
  JTextField tf;

  public Fenster_mit_Button_und_Textfeld(String title) {
    super(title);
```

```
//Der Layoutmanager wird ausgeschaltet (hierzu später mehr)
    this.setLayout(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(300,400);
//Ein Button wird erzeugt
    b = new JButton("Klicken!");
//Der Button bekommt eine Koordinate (50,200) im Fenster sowie eine
//Breite (200) als auch eine Höhe
    b.setBounds(50,200,200,50);
//Der Button wird dem Fenster hinzugefügt
    this.add(b);

//Ein Textfeld wird erzeugt, der Rest in analog dem Button
    tf = new JTextField();
    tf.setBounds(50,100,200,50);
    this.add(tf);

    this.setVisible(true);
  }

  public static void main(String[] args) {
   new Fenster_mit_Button_und_Textfeld("Fenster_Button_Textfeld");
  }
}
```

Das Ergebnis:



Aufgabenstellung:

Erstellen Sie ein Fenster mit drei Textfeldern. Die Felder sind mit dem Text: „Zahl1“, „Zahl2“, „Ergebnis“ beschriftet. Hinweis: Die Beschriftung erfolgt über die Komponente JLabel.
Weiterhin hat das Fenster vier Buttons. Die Buttons sind mit den vier Grundrechenarten beschriftet.

Interfaces

Etwas graue Theorie muss an der Stelle sein.

Ein Interface definiert Methoden, implementiert sie aber nicht. Die Methoden werden mit ihrem Methodenkopf aufgeschrieben, allerdings wird der Methodenkörper nicht ausgefüllt.

Warum dies?

1. Eigenschaft eines Interfaces als Vorschrift für den Entwickler

Eine Klasse, welche ein Interface einbindet, muss alle im Interface definierten Methoden implementieren. Hier kann es zum Beispiel von Vorteil sein, dass mehrere Programmierer des gleichen Projektes sich darüber einigen müssen, wie Methoden heißen, was die Methoden übergeben bekommen und was sie zurückgeben. Beispiel:

Ein Entwicklerteam arbeitet an einer Banksoftware. Hier gibt es die Klassen `Konto`, `Sparbuch` etc. Alle diese Klassen müssen eine Methode einzahlen() besitzen. Es wird das folgende Interface geschrieben:

```
public interface KontoInterface{
       public void einzahlen(double betrag);
}
```

Erstellt nun eine Entwickler die Klasse `Konto` und implementiert (Schlüsselwort `implements`) das `KontoInterface`, so muss er die Methode `einzahlen()` mit in deiner Klasse implementieren. Erstellt er nun das folgende Konstrukt:

```
public class Konto implements KontoInterface{
    private double kontostand;

    public Konto(double kontostand){
      this.kontostand = kontostand;
    }
}
```

kommt es beim kompilieren zu folgender Fehlermeldung:

```
error: Konto is not abstract and does not override abstract method
einzahlen(double) in KontoInterface
```

Der Entwickler wird also darauf hingewiesen, dass er die Methode `einzahlen()` implementieren muss (und genauso würde es dem Entwickler der Klasse `Sparbuch` ergehen)!

Das Problem kann er wie folgt lösen:

```
public class Konto implements KontoInterface{
    private double kontostand;

    public Konto(double kontostand){
      this.kontostand = kontostand;
    }
  //Diese Methode wurde vom Interface vorgegeben!
    public void einzahlen(double betrag){
      kontostand = kontostand + betrag;
    }
}
```

Man sieht hier also einmal die Eigenschaft eines Interfaces als **Vorschrift**.

2. Interface zur Umsetzung von Funktionalität:

Bei Programmierung einer GUI muss man auf Eingaben, Klicks etc. reagieren können. Dies geschieht über Listener. Diese Listener sind Prozesse im Hintergrund, die dann, wenn ein definiertes Ereignis eingetreten ist (Enter-Taste beim Textfeld, Buttonklick etc.), eine vorher definierte Methode aufrufen. Da diese Methode vorhanden sein muss (siehe oben), muss man auch hier ein Interface implementieren. Beispielsweise achtet der ActionListener unter anderem darauf, ob Buttons geklickt

wurden. Ist dies passiert, so ruft er die Methode actionPerformed() des Frame auf. Um also sicherzustellen, dass es diese Methode überhaupt gibt, muss das Frame das Interface ActionListener implementieren.

Anmerkung:

Natürlich passiert beim Implementieren und Hinzufügen eines Listeners wesentlich mehr im Hintergrund, worauf hier aber nicht eingegangen werden soll. Ein Beispiel, wie man selbst einen Listener und ein solches dazugehörendes Interface programmiert findet sich im Kaiptel 5 des Buches MSR mit USB und JAVA.

Listener

Um nun einem Fenster etwas Leben einzuhauchen, muss ein Listener bemüht werden. In diesem Fall wird beschrieben, wie auf einen Klick auf einen Button beispielsweise reagiert werden kann.

Hierzu muss als erstes das Frame das Interface `ActionListener` implementieren:

```
public class FensterMitEinfachemActionListener extends JFrame
implements ActionListener
```

Um den `ActionListener` zu verwenden muss das `Paket java.awt.event.*` eingebunden werden.

Jetzt wird das Fenster herkömmlich aufgebaut. Einen kleinen Unterschied gibt es: die Komponenten, auf die reagiert werden soll, müssen beim Listener angemeldet werden. In diesem Falle ist das der Button b:

```
b.addActionListener(this);
```

Zur Erklärung:

Die Methode `addActionListener` meldet den Button an dem Listener an, der in der runden Klammer übergeben wird. Dies wiederum muss ein Objekt aus einer Klasse sein, welche den `ActionListener` als Interface eingebunden hat. Da hier das Frame selbst das passende Interface implementiert, wird `this` übergeben.

Jetzt muss nur noch in der Methode `public void actionPerformed()` auf den Klick reagiert werden. Die Methode wird jetzt automatisch aufgerufen, wenn der Button geklickt wurde. In diesem Fall werden die Anzahl der Klicks gezählt und dies in einem Label ausgegeben.

Nun die vollständige Klasse:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FensterMitEinfachemActionListener extends JFrame implements ActionListener{
  JButton b;
  JLabel l;
  int anzahlKlicks=0;

  public FensterMitEinfachemActionListener(String title) {
    super(title);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(300,200);
    setResizable(false);
    this.setLayout(null);
```

```
    //Ein Label wird erzeugt um Text anzuzeigen
    l = new JLabel();
    l.setBounds(10,10,250,30);
    this.add(l);

    b = new JButton("Klick mich an");
    b.setBounds(10,50,250,50);
    //Der Button bekommt einen Listener zugewiesen
    b.addActionListener(this);
    this.add(b);

    this.setVisible(true);
  }
  //Methode des Interfaces
  public void actionPerformed(ActionEvent ae){
    anzahlKlicks++;
    l.setText("Es wurde " +anzahlKlicks + " mal geklickt");
  }

  public static void main(String[] args) {
    new FensterMitEinfachemActionListener("FensterMitEinfachemActionListener");
  }

}
```

Aufgabenstellung:

Erstellen Sie ein Fenster mit zwei Textfeldern und einem Button. Beim Klick auf den Button soll der Text des ersten Textfeld in das zweite kopiert werden. Das erste Textfeld wird automatisch wieder gelöscht.

Hinweis:

Über die Methode `String getText()` aus der Klasse `JTextField` kann man den Text eines Feldes auslesen, die Methode `setText(String text)` setzt den Text eines Textfeldes.

Bis jetzt wurden Fenster fast immer absolut programmiert. Dies bedeutet, dass das Fenster eine feste Größe hat und auch die platzierten Komponenten fix gesetzt wurden. Dies hat den Nachteil, dass bei einer Veränderung der Fenstergröße Komponenten nicht mehr oder nur noch teilweise zu sehen waren. Alternativ konnte man das Fenster dazu verdonnern, keine Größenänderung zuzulassen.

Um die Komponenten absolut zu setzen bedarf es der Zeile (das Frame ist in diesem Dokument immer das Objekt f!):

```
f.setLayout(null);
```

Will man dies nicht, so kann man sich einem Layoutmanager bedienen. Hier gibt es einige von denen hier nur drei aufgegriffen werden sollen. Für den normalen Programmbereich sollte dies langen, für höhere Ansprüche wird auf die einschlägige Literatur verwiesen.

1. Der BorderLayoutManager

Dieser Layoutmanager teilt das Fenster in fünf Bereiche ein: Norden (`BorderLayout.NORTH`), Süden (`BorderLayout.SOUTH`), Osten (`BorderLayout.EAST`), Westen (`BorderLayout.WEST`), und Zentrum (`BorderLayout.CENTER`).

Der Layoutmanager wird über die Zeile:

```
f.setLayout(new BorderLayout());
```

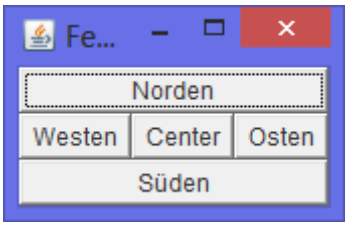
dem Frame zugewiesen.

Fügt man dem Frame nun ein Objekt hinzu, so wird in der Methode add() zusätzlich die Platzierung im Layout mit angegeben:

```
f.add(b,BorderLayout.CENTER); //setzt einen Button b in die Mitte
                              //des Layouts
```

Am Ende der Framegestaltung wird abschließend noch die Methode `pack()` aufgerufen (dies passiert bei allen Layoutmanagern). Diese sorgt dafür, dass das Fenster exakt die Mindestgröße bekommt, die das Fenster mit all seinen Komponenten benötigt.

Das folgende Beispiel hat den folgenden Screenshot zur Folge:



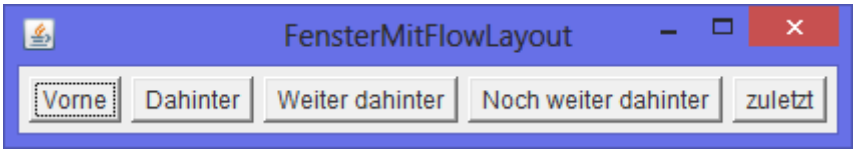
Der Quelltext:

```
import java.awt.*;
import javax.swing.*;

public class FensterMitBorderLayout extends JFrame {

  public FensterMitBorderLayout(String title) {
    super(title);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    //setSize(frameWidth, frameHeight);
    this.setLayout(new BorderLayout());
    Button bNorden = new Button("Norden");
    this.add(bNorden,BorderLayout.NORTH);
    Button bSueden = new Button("Süden");
    this.add(bSueden,BorderLayout.SOUTH);
    Button bOsten = new Button("Osten");
    this.add(bOsten,BorderLayout.EAST);
    Button bWesten = new Button("Westen");
    this.add(bWesten,BorderLayout.WEST);
    Button bCenter = new Button("Center");
    this.add(bCenter,BorderLayout.CENTER);
    this.pack();
    setVisible(true);
  } // end of public FensterMitBorderLayout

  public static void main(String[] args) {
    new FensterMitBorderLayout("FensterMitBorderLayout");
  }
}
```

2. Der FlowLayoutManager

Bei diesem Layoutmanager werden die Komponenten einfach nach und nach aneinandergesetzt.

Das folgende Beispiel hat die folgende Ausgabe zur Folge:



Der Quellcode:

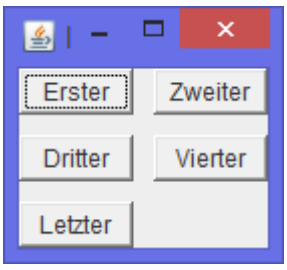
```
import java.awt.*;
import javax.swing.*;

public class FensterMitFlowLayout extends JFrame {

  public FensterMitFlowLayout(String title) {
    super(title);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLayout(new FlowLayout());
    Button bNorden = new Button("Vorne");
    this.add(bNorden);
    Button bSueden = new Button("Dahinter");
    this.add(bSueden);
    Button bOsten = new Button("Weiter dahinter");
    this.add(bOsten);
    Button bWesten = new Button("Noch weiter dahinter");
    this.add(bWesten);
    Button bCenter = new Button("zuletzt");
    this.add(bCenter);
    this.pack();
    setVisible(true);
  } // end of public FensterMitFlowLayout

  public static void main(String[] args) {
    new FensterMitFlowLayout("FensterMitFlowLayout");
  }

}
```

3. Der GridLayoutManager

Beim GridLayout wird eine Tabelle mit x Zeilen und y Spalten erzeugt (wie man später sieht (siehe Kommentar im Quelltext), ist die Anzahl der Zeilen uninteressant und kann auf 0 gesetzt werden).

Die dem Frame hinzugefügten Komponenten werden dann in der Reihenfolge von links oben nach rechts unten eingefügt, wobei immer erst die jeweilige Zeile gefüllt wird.

Das nächste Programm hat die folgende Bildschirmausgabe zur Folge:



Der Quellcode:

```
import java.awt.*;
import javax.swing.*;

public class FensterMitGridLayout extends JFrame {

  public FensterMitGridLayout(String title) {
    super(title);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//hier wird ein GridLayout mit 0 Zeilen und zwei Spalten erzeugt. Da
//sich die Anzahl der Zeilen aus den hinzugefügten Komponenten
//automatisch ergibt, kann hier 0 angegeben werden. Die beiden
//letzten Parameter setzen den Abstand der Komponenten in Pixel
//(vertikal und horizontal)
    this.setLayout(new GridLayout(0,2,10,10));
    Button bNorden = new Button("Erster");
    this.add(bNorden);
    Button bSueden = new Button("Zweiter");
    this.add(bSueden);
    Button bOsten = new Button("Dritter");
    this.add(bOsten);
    Button bWesten = new Button("Vierter");
    this.add(bWesten);
    Button bCenter = new Button("Letzter");
    this.add(bCenter);
    this.pack();
    setVisible(true);
  } // end of public FensterMitGridLayout

  public static void main(String[] args) {
    new FensterMitGridLayout("FensterMitGridLayout");
  }
}
```

Aufgabenstellung:

Aufgabe 1:

Erstellen Sie ein Fenster zwei Textfenstern und zwei Buttons mittels des Borderlayoutmanagers. Button 1 kopiert den Text von Textfeld 1 in Textfeld 2. Button 2 löscht die Textfelder.

Aufgabe 2:

Lösen Sie Aufgabe 1 mit dem FlowLayout.

Aufgabe 3:

Lösen Sie Aufgabe 1 mit dem GridLayout.

Panel

Ein Panel ist sowohl eine grafische Komponente als auch ein Container für weitere grafische Komponenten. Es eignet sicher daher sehr gut, um seine Applikation aus verschiedenen Bereichen

aufzubauen. Auch kann jedem Panel sein eigener LayoutManager zugewiesen werde, womit man im Prinzip jede Art von Fenster gestalten kann.

Das nachfolgende Beispiel führt zu folgendem Screenshot:

Diese Anordnung wäre mit den drei vorgestellten LayoutManagern nicht möglich. Unterteilt man das Fenster jedoch in zwei Hälften (oben drei Textfelder und unten zwei Buttons), so kann man mit zwei Panels, die jeweils ein FlowLayout besitzen, dieses Fenster aufbauen.

Das erste Panel sieht im Quelltext wie folgt aus:

```
import javax.swing.*;
import java.awt.*;

//Die Klasse erbt von JPanel
public class MeinObenPanel extends JPanel{
  public MeinObenPanel(){
//das FlowLayout wird gesetzt
    this.setLayout(new FlowLayout());
//drei Textfelder werden erzeugt und dem Panel hinzugefügt
    JTextField tf1 = new JTextField("Erste Zahl");
    JTextField tf2 = new JTextField("Zweite Zahl");
    JTextField tf3 = new JTextField("Ergebnis");
    this.add(tf1);
    this.add(tf2);
    this.add(tf3);
  }
}
```

Das untere Panel stellt sich im Quelltext folgend dar (da der Quelltext fast identisch ist, wird er hier nicht weiter kommentiert):

```
import javax.swing.*;
import java.awt.*;

public class MeinUntenPanel extends JPanel{
  public MeinUntenPanel(){
    this.setLayout(new FlowLayout());
    JButton b1 = new JButton("Addieren");
    JButton b2 = new JButton("Subtrahieren");
    this.add(b1);
    this.add(b2);
  }
}
```

Jetzt müssen dem Frame nur noch die beiden Panels passend positioniert hinzugefügt werden:

```
import java.awt.*;
import javax.swing.*;

public class FensterMitPanels extends JFrame {

  public FensterMitPanels(String title) {
    super(title);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLayout(new BorderLayout());
//Das obere Panel wird erzeugt und dem Frame hinzugefügt
    JPanel meinObenPanel = new MeinObenPanel();
    this.add(meinObenPanel,BorderLayout.NORTH);
//Das untere Panel wird erzeugt und dem Frame hinzugefügt
    JPanel meinUntenPanel = new MeinUntenPanel();
    this.add(meinUntenPanel,BorderLayout.SOUTH);
    this.pack();
    setVisible(true);
  } // end of public FensterMitPanels

  public static void main(String[] args) {
    new FensterMitPanels("FensterMitPanels");
  }

}
```

Aufgabenstellung:

Wandeln Sie den Quelltext so ab, dass die drei Textfelder jeweils ein Label als Beschriftung bekommen. Versuchen Sie durch geeignete Listener das Programm so mit Funktionalität auszustatten,

Weitere Listener

Nachfolgend werden vier verschiede Listener und speziellere Funktionen in vier Beispielen dargestellt. In den Klassen werden teilweise Layoutmanager verwendet, diese werden im nachfolgenden Dokument Layoutmanager erklärt und können hier einfach ignoriert werden. Die Klassen sind alle sehr ausführlich kommentiert, so dass man sich die Funktionalitäten sicher selbst erschließen kann.

Beispiel 1: ActionListener

```
import java.awt.*;
//Wird benötigt für den ActionListener
import java.awt.event.*;
//Wird benötigt für das JFrame
import javax.swing.*;

//Die Klasse erbt von JFrame (ein Fenster) und implementiert das Interface ActionListener
public class FensterActionListener extends JFrame implements ActionListener{

/*
```

```
 * Im Konstruktor wird das Fenster erzeugt, der Button gesetzt und dem Button der
ActionListener
 * aufgedrückt.
 * */

  public FensterActionListener(){
    //Der Konstruktor der Oberklasse wird aufgerufen
    super("Beispiel für ActionListener");
    //Wenn man auf das x des Fenster klickt, werden alle hiervon laufenden Prozesse beendet
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    //Der Button wird erzeugt
    JButton b = new JButton("Klick mich an");
    //Der Button bekommt eine Größe zugewiesen
    b.setSize(300,200);
    //Der Button bekommt einen ActionListener aufgedrückt. In diesem Falle ist this das Objekt
    //der Fensterklasse selbst, welches ja einen ActionListener implementiert hat.
    b.addActionListener(this);
    //Der Button wird platziert. Hierbei wird er in der Mitte des Layoutes gesetzt
    this.add(b,BorderLayout.CENTER);
    //Das Fenster wird "passend" gemacht
    this.pack();
    //Das Fenster wird sichtbar gemacht
    this.setVisible(true);
  }
  /*
   * Die Methode muss implementiert werden, sie wird durch das Interface definiert.
   * Gleichzeitig wird diese Methode immer dann aufgerufen, wenn ein Event entstanden ist,
   * beispielsweise ein Mausklick auf den Button
   */
  @Override
  public void actionPerformed(ActionEvent ae) {
    //Es wird ausgegeben, welche Komponente (beispielsweise welcher Button) gedrückt wurde
    System.out.println(ae.getSource());
    //Es wird ausgegeben, welches Kommando die geklickte Komponente hat
    //In diesem Fall: Klick mich an
    System.out.println(ae.getActionCommand());
    //Die ID des Objektes wird ausgegeben
    System.out.println(ae.getID());
    //Die Zeit in Millisekunden wird ausgegeben, an dem der Button gedrückt wurde
    System.out.println(ae.getWhen());

  }

  public static void main (String args[]){
    FensterActionListener f = new FensterActionListener();
  }

}
```

Beispiel 2: TextListener

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.event.*;
import javax.swing.*;


public class FensterKeyListener extends JFrame implements KeyListener {

  public FensterKeyListener(){
    //Der Konstruktor der Oberklasse wird aufgerufen
    super("Beispiel für KeyListener");
    //Wenn man auf das x des Fenster klickt, werden alle hiervon laufenden Prozesse beendet
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    //Das Textfeld wird erzeugt
    JTextField t = new JTextField();
    //Das Textfeld bekommt eine Größe zugewiesen
    t.setSize(300,200);
    //Der Textfeld bekommt einen KeyListener aufgedrückt. In diesem Falle ist this das Objekt
    //der Fensterklasse selbst, welches ja einen KeyListener implementiert hat.
```

```
    t.addKeyListener(this);
    //Das Textfeld wird platziert. Hierbei wird es in der Mitte des Layoutes gesetzt
    this.add(t,BorderLayout.CENTER);
    //Das Fenster wird "passend" gemacht
    this.pack();
    //Das Fenster wird sichtbar gemacht
    this.setVisible(true);
  }

  //Wird aufgerufen, wenn eine Taste gedrückt wird
  @Override
  public void keyPressed(KeyEvent ke) {
    System.out.println("Taste gedrueckt");
    //Mit der folgenden Methode können auch die Pfeiltasten abgefragt werden
    System.out.println(ke.getKeyCode());

  }

  //Wird aufgerufen, wenn eine Taste losgelassen wird
  @Override
  public void keyReleased(KeyEvent arg0) {
    System.out.println("Taste losgelassen");
  }

  //Wird aufgerufen, wenn eine Taste gedrückt und wieder losgelassen wird
  @Override
  public void keyTyped(KeyEvent ke) {
    System.out.println("Taste gedrueckt und wieder losgelassen");
    //Gibt die Taste aus
    System.out.println(ke.getKeyChar());

  }

  public static void main(String[] args) {
    FensterKeyListener f = new FensterKeyListener();

  }

}
```

Beispiel 3: MouseListener

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.JFrame;

public class FensterMouseListener extends JFrame implements MouseListener {

  public FensterMouseListener(String title)  {
    super(title);
    //Der Layoutmanager wird ausgeschaltet
    this.setLayout(null);
    this.setSize(400, 300);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    //Dem Fenster wird ein MouseListener aufgedrückt
    this.addMouseListener(this);
    this.setVisible(true);
  }

  //Wird aufgerufen, wenn die Maus geklickt wurde (Maus drücken und loslassen).
  @Override
  public void mouseClicked(MouseEvent me) {
    System.out.println("Maus wurde geklickt");
    //Gibt aus, welcher Button gedrückt wurde
    System.out.println("Button: "+me.getButton());
    //Gibt die x-Koordinate des Mauszeigers aus
    System.out.println(me.getX());
    //Gibt die y-Koordinate des Mauszeigers aus
    System.out.println(me.getY());
```

```
    //Gibt die Anzahl der Klicks aus (beispielsweise 2 beim Doppelklick)
    System.out.println(me.getClickCount());

  }

  //Wird aufgerufen, wenn die Maus das Fenster betritt
  @Override
  public void mouseEntered(MouseEvent arg0) {
    System.out.println("Maus hat das Fenster betreten");
  }

  //Wird aufgerufen, wenn die Maus das Fenster verlässt
  @Override
  public void mouseExited(MouseEvent arg0) {
    System.out.println("Maus Maus hat das Fenser verlassen");
  }

  //Wird aufgerufen, wenn die Maus gedrückt wurde
  @Override
  public void mousePressed(MouseEvent arg0) {
    System.out.println("Maus wurde gedrückt");
  }

  //Wird aufgerufen, wenn die Maus losgelassen wird
  @Override
  public void mouseReleased(MouseEvent arg0) {
    System.out.println("Maus wurde losgelassen");
  }

  public static void main(String[] args) {
    FensterMouseListener f = new FensterMouseListener("Beispiel für einen MouseListener");
  }
}
```

Beispiel 4: MouseMotionListener

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;


public class FensterMouseMotionListener extends JFrame implements MouseMotionListener {


  public FensterMouseMotionListener(String title)  {
    super(title);
    //Der Layoutmanager wird ausgeschaltet
    this.setLayout(null);
    this.setSize(400, 300);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    //Dem Fenster wird ein MouseMotionListener aufgedrückt
    this.addMouseMotionListener(this);
    this.setVisible(true);
  }

  //Wird aufgerufen, wenn mit der Maus gezogen wird (bei gedrückter linker oder rechter
Maustaste)
  @Override
  public void mouseDragged(MouseEvent arg0) {
    System.out.print("Maus ziehen (Taste gedrueckt halten) ");
    System.out.print(" X-Koordinate: "+arg0.getX());
    System.out.println(" Y-Koordinate: "+arg0.getY());

  }

  //Wird aufgerufen, wenn die Maus bewegt wird
  @Override
  public void mouseMoved(MouseEvent arg0) {
    System.out.print("Maus bewegen ");
    System.out.print(" X-Koordinate: "+arg0.getX());
```

```
    System.out.println(" Y-Koordinate: "+arg0.getY());
  }

  public static void main(String[] args) {
    FensterMouseMotionListener f = new FensterMouseMotionListener("Beispiel für
MouseMotionListener");

  }
}
```

Aufgabenstellungen:

Aufgabe 1:

Erstellen Sie ein Fenster mit zwei Textfeldern und zwei Buttons. Der erste Button kopiert den Text aus dem ersten Textfeld in das zweite. Der zweite Button löscht beide Textfelder.

Aufgabe 2:

Erstellen Sie ein Fenster mit einem Textfeld. Wird in dem Textfeld der Buchstabe c eingegeben, so wird der Text im Textfeld komplett gelöscht. Wird der Buchstabe x eingegeben, so wird das Fenster geschlossen und das Programm beendet.

Aufgabe 3:

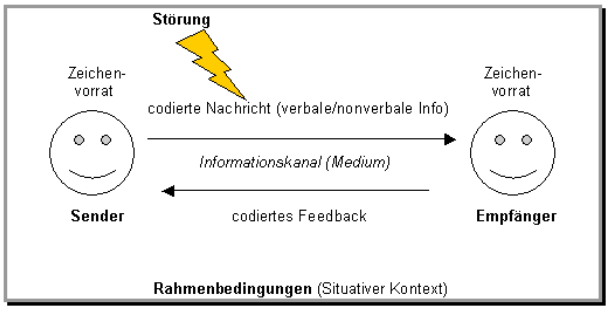
Erstellen Sie ein Fenster mit einem Textfeld und einem Button. Jeder Buchstabe im Textfeld soll nicht angezeigt werden, sondern durch das Zeichen * ersetzt werden. Wird der Button geklickt, so wird der Originaltext des Textfeldes auf die Konsole ausgegeben. Anschließend werden die Zeichen im Textfeld gelöscht.

Hinweis:

Aufgaben zu den beiden Listener die für Mausaktionen zuständig sind, folgen im Abschnitt „Zeichen“.

Kommunikation im Netzwerk

Die Kommunikation zweier Rechner über eine der Standardschnittstellen ist langsam, aufwendig und vor allem an örtliche Bedingungen gebunden. Im Zeitalter von Internet, Intranet etc. ist es wichtig, dass PCs über das Netzwerk miteinander kommunizieren lernen. Hierbei ist es auch für den Anwendungsentwickler wichtig, technische Hintergrundinformationen über die Kommunikation im Netz zu erlernen. Es muss beispielsweise das geeignete Protokoll für seine Software auswählen können oder aber muss er genau über die Normierung von Diensten im Netz informiert sein.



Kommunikation ist allgemein nicht definiert, was nicht heißt, dass es keine wissenschaftlichen Untersuchungen gibt. Das Gegenteil ist der Fall[10]. Im Bereich der Datenkommunikation muss das Sender-Empfänger-Modell der Kommunikation von Stuart Hall (1970) genannt werden:

Wichtig ist hierbei, dass Sender und Empfänger den gleichen Zeichenvorrat verwenden. Der Sender muss die Nachricht so absenden können, dass sie genau für den Empfänger bestimmt ist (die Nachricht muss also adressiert sein!). Der Informationskanal (das Informationsmedium) muss dafür sorgen, dass eine versendete Nachricht den Empfänger vollständig erreichen kann. Die Kodierungsart des Senders muss dem Empfänger bekannt sein. Erfolgt eine Rückantwort, wird der Empfänger zum Sender!

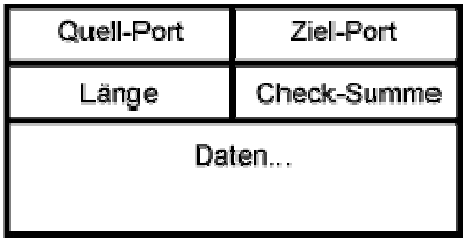
Bezüglich der Genauigkeit der Übertragung muss man Abwägungen treffen:

Der Satz: "In der Not schmeckt die Wurst ach ohne Brot." ist inhaltlich eindeutig zu verstehen. Hierbei handelt es sich nicht um einen Rechtschreibfehler sondern um einen möglichen Fehler bei der Übertragung dieses Satzes[11]. Es ist also durchaus möglich, Fehler bei der Informationsübertragung zuzulassen, sofern der Zweck der Information nicht beeinträchtigt wird. Wird Beispielsweise ein Video mit einer Auflösung von 800 x 600 Pixel über das Netzwerk übertragen, so sind einige Pixelfehler durchaus in Kauf zu nehmen (der Betrachter wird diese kaum bemerken), ruckeln jedoch die laufenden Bilder aufgrund von Geschwindigkeitsproblemen im Netz, so wird dies vom Betrachter als unangenehm empfunden. Werden jedoch die Daten von 1000 Bankkunden über das Netzwerk übermittelt, so darf nicht der kleinste Fehler bei der Datenübertragung passieren. Hinsichtlich dieser Betrachtung beschreibe ich für die Programmierung zwei verschiede Protokolle:

[10] Insbesondere ist hier Paul Watzlawick, ein Kommunikationsforscher zu nennen.
[11] Experiment: Streichen Sie aus einem beliebigen Satz Buchstaben heraus und geben Sie den Satz jemand anderem zu Lesen. Überprüfen Sie, wie viel Buchstaben Sie weglassen können.

UDP (User Datagram Protocol)

Das UDP-Protokoll ist ein verbindungsloses Protokoll. Das bedeutet, dass bei der Nachrichtenübertragung keine Rückmeldung vom Empfänger erfolgt. Wurde ein Datenpaket fehlerhaft übertragen, so bekommt der Empfänger dies nicht mit (er weiß ja nicht, wie die Information korrekt lautet), es sei denn, eine Prüfsumme kann dem Datenpaket entnommen werden. Auch ist bei UDP die Reihenfolge der einzelnen Pakete nicht unempfindlich, da es je nach Übertragung einige Pakete schneller zum Ziel gelangen als andere. Dies kann dazu führen, dass die empfangene Reihenfolge nicht mehr richtig ist und vom Empfänger korrigiert werden muss. Allen diesen Einschränkungen steht als Vorteil des UDP gegenüber, dass der Overhead[12] im Vergleich zu anderen Protokollen sehr gering ist.



[13]UDP-Pakete setzen sich aus dem Header-Bereich und dem Daten-Bereich zusammen. Im Header sind alle Informationen enthalten, die eine einigermaßen geordnete Datenübertragung zuläßt und die ein UDP-Paket als ein solches erkennen lassen. Der UDP-Header ist in 32-Bit-Blöcke unterteilt. Er besteht aus zwei solcher Blöcke, die den Quell- und Ziel-Port, die Länge des gesamten UDP-Paketes und die Check-Summe enthalten. Der UDP-Header ist mit 8 Byte sehr schlank und verbraucht damit wenig Rechenzeit und Netzlast. Der Datenanteil sollte aus technischen Gründen auf maximal 508 Bytes begrenzt werden.

Bedeutung der Felder im UDP-Header

Feldinhalt	Bit	Beschreibung
Quell-Port (Source-Port)	16	Hier steht der Quell-Port, von der die Anwendung das UDP-Paket verschickt. Bei einer Stellenanzahl von 16 Bit beträgt der höchste Port 65535.
Ziel-Port (Destination-Port)	16	Hier steht der Ziel-Port, über welchen das UDP-Paket der Anwendung zugestellt wird. Bei einer Stellenanzahl von 16 Bit beträgt der höchste Port 65535.
Länge	16	In diesem Feld wird angegeben, wie lang das gesamte UDP-Paket ist. Über diesen Wert kann die Vollständigkeit des UDP-Paketes ermittelt werden.
Check-Summe	16	Über dieses Feld wird kontrolliert, ob das UDP-Paket fehlerfrei übertragen wurde. Die Check-Summe bietet keinen Schutz vor Datenverlust.

Netzwerkprogrammierung über Sockets (UDP)

Die Klasse DatagramSocket stellt eine Softwareschnittstelle für den Programmierer dar. Hierbei muss sich der Programmierer nicht mehr über den Header der einzelnen Pakete Gedanken machen. Lediglich

[12] Als Overhead bezeichnet man alle Daten, die notwendig sind, um die eigentlichen Nutzdaten an das richtige Ziel zu bringen.

[13] aus http://www.elektronik-kompendium.de/sites/net/0812281.htm

die IP – Adresse des Empfängers und der Port[14], auf dem die Daten übertragen werden müssen einmalig gesetzt werden.

Beispiel: Kommunikation von einem Rechner über das Netz zu einem anderen Rechner per UDP

In diesem Beispiel soll nur kurz angerissen, wie die Kommunikation zwischen zwei Rechnern per UDP möglich ist.

Die Klasse UDPClient[15]

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

public class UDPClient
{
	public static void main (String args[])
	{
		String host = "127.0.0.1";
		int port = 5000;
		try
		{
			DatagramSocket socket = new DatagramSocket ();
			InetAddress adr = InetAddress.getByName (host);
			DatagramPacket paket = new DatagramPacket (new byte[1], 1, adr, port);
			BufferedReader in = new BufferedReader (new
InputStreamReader(System.in));
			System.out.println ("Programmende bei Eingabe von 'ende'");
			while (true)
			{
				System.out.print (">");
				String text = in.readLine();
				if (text.equals ("ende"))
				{
					break;
				}
				byte [] daten = text.getBytes();
				paket.setData (daten);
				int laenge = (daten.length < 508) ? daten.length : 508;
//Bedingungsoperator
				paket.setLength (laenge);
				socket.send (paket);
			}
			in.close ();
			socket.close();
		}
		catch (Exception e){e.printStackTrace();}
	}
}
```

Erklärungen:

[14] Der Begriff "Port" wird weiter unten erklärt.

[15] Klasse leicht modifiziert aus Dietmar Abts, Aufbaukurs Java, Vieweg-Verlag, 1. Auflage 2003, Braunschweig/Wiesbaden

Es wird ein Objekt aus der Klasse DatagramSocket instanziert. Anschließend wird mit Hilfe eines Objektes aus der Klasse DatagramPacket die Verbindung zum Empfänger unter Angabe von Portnummer und IP – Adresse (in Form eines Objektes aus der Klasse InetAddress) hergestellt. Dem Paket muss nun nur noch mitgeteilt werden wie viele Bytes gesendet werden und vor allem welche. Dies passiert über ein Byte – Array. Die Größe des übertragenen Bytes wird auf 508 Bytes begrenzt. Gibt der Benutzer auf der Konsole das Wort 'ende' ein, so werden die Eingabeströme und der Socket geschlossen.

Die Klasse UDPServer[16]

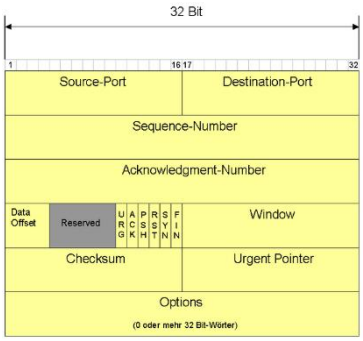
```
import java.net.*;

public class UDPServer
{
	public static void main (String args[])
	{
		int port = 5000;
		try
		{
		DatagramSocket socket = new DatagramSocket (port);
		byte daten[] = new byte[508];
		DatagramPacket paket = new DatagramPacket (daten, daten.length);
			while (true)
			{
				socket.receive (paket);
				String text = new String (paket.getData(), 0,paket.getLength());
				System.out.println (text);
				paket.setLength (daten.length);
			}
		}
		catch (Exception e){}
	}
}
```

Erklärung:

Der Verbindungsaufbau ist ähnlich, nur wird hier das Objekt aus der Klasse DatagramPacket nicht an eine IP – Adresse ausgerichtet. Über die Methode recieve kann der Socket Pakete empfangen (in diesem Fall wird das Objekt aus der Klasse DatagramPacket übergeben und dabei verändert). Aus dem Paket können die Länge der Daten in Bytes und die Daten in Form eines Byte-Arrays gewonnen werden. Aus dem Byte-Array wird ein Objekt aus der Klasse String erzeugt, welches auf der Konsole ausgegeben wird.

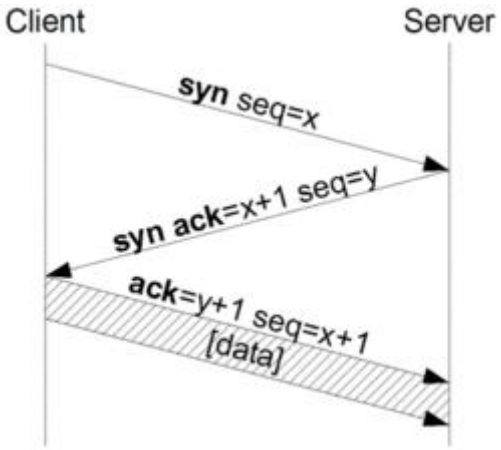
[16] Klasse leicht modifiziert aus Dietmar Abts, Aufbaukurs Java, Vieweg-Verlag, 1. Auflage 2003, Braunschweig/Wiesbaden

TCP



Das wichtigste Protokoll ist sicherlich das TCP (Transmission Control Protocoll). Im Gegensatz zu UDP ist TCP ein verbindungsorientiertes Protokoll. Die verschiedensten Informationen innerhalb des Protokolls sowie definierte Ablaufmechanismen sorgen dafür, dass Daten auf Senderseite getrennt werden können und auf Empfängerseite in der richtigen Reihenfolge zusammengesetzt werden. Auch werden auf dem Übertragungsweg verloren gegangene Daten erneut angefordert. Das Protokoll eignet sich demnach sehr gut, um wichtige Daten zu übertragen, deren komplette Übertragung vorausgesetzt wird. Natürlich erfordert diese Sicherheit ein höheres Datenvolumen im Übertragungsheader[17]. Der Header wird an dieser Stelle nicht erklärt, im Internet finden sich reichliche Erklärungen zu dieser Thematik.

Beim Aufbau einer TCP-Verbindung kommt der so genannte Drei-Wege-Handshake zum Einsatz. Der Rechner, der die Verbindung initiieren will, sendet dem anderen ein *SYN*-Paket mit einer Sequenznummer **x**. Es handelt sich also um ein Paket, dessen *SYN-Bit* im Paketkopf gesetzt ist (siehe TCP-Header). Die initiale Sequenznummer ist beliebig und wird vom jeweiligen Betriebssystem festgelegt.



Die Gegenstelle (siehe Skizze) empfängt das Paket und sendet in einem eigenen SYN-Paket im Gegenzug seine initiale Sequenznummer **y**. Zugleich bestätigt sie den Erhalt des ersten SYN-Pakets, indem sie die Sequenznummer inkrementiert und x+1 im ACK-Teil des Headers zurückschickt.
Der Client bestätigt zuletzt den Erhalt des SYN/ACK-Pakets durch das Senden eines eigenen ACK-Pakets mit der Sequenznummer x+1 und dem ACK-Wert y+1. Die Verbindung ist damit aufgebaut.

Ein TCP-Segment hat typischerweise eine Größe von 1500 Bytes. Es darf nur so groß sein, dass es in die darunter liegende Übertragungsschicht passt, das Internetprotokoll IP. Das IP-Paket ist theoretisch bis 65535 Bytes (64 kB) spezifiziert, wird aber selbst meist über Ethernet übertragen, und dort ist die Rahmengröße auf 1500 Bytes festgelegt. TCP und IP Protokoll definieren jeweils einen Header von 20 Bytes Größe. Für die Nutzdaten bleiben in einem TCP/IP-Paket also 1460 Bytes übrig. Da die meisten Internet-Anschlüsse DSL verwenden, gibt es dort noch das Point-to-Point Protocol (PPP) zwischen IP und Ethernet, was noch einmal 8 Bytes für den PPP-Rahmen kostet. Dem TCP/IP-Paket verbleiben im Ethernet-Rahmen nur 1492 Bytes MTU, die Nutzdaten reduzieren sich auf insgesamt 1452 Bytes MSS. Dies entspricht einer Auslastung von 96,8%.
Über die meisten dieser Informationen braucht sich der Programmierer nicht zu kümmern. Er kann auf Sockets zugreifen.

[17] Abbildung aus http://de.wikipedia.org/wiki/Transmission_Control_Protocol

Netzwerkprogrammierung über Sockets (TCP/IP)

Sockets und TCP/IP

Die Kommunikation über ein Netzwerk wäre für den Programmierer ein schwieriges Unterfangen wenn er alle technischen Details der Netzwerkkommunikation realisieren müsste. Hier hilft dem Programmierer das Prinzip der Sockets:

"Ein **Socket** (wörtlich übersetzt "Sockel" oder "Steckverbindungen") ist eine bidirektionale Software-Schnittstelle zur Interprozess- (IPC) oder Netzwerk-Kommunikation. **Sockets** sind vollduplex fähig (es können Daten gleichzeitig in beide Richtungen gesendet werden)"[18].

Bei einem Socket handelt es sich um ein Ende einer Kommunikationsschnittstelle zwischen zwei Programmen, welche Daten über ein Netzwerk austauschen. Eine Applikation fordert einen Socket vom Betriebssystem an, und kann über diesen anschließend Daten verschicken und empfangen. Das Betriebssystem hat die Aufgabe, alle benutzten Sockets sowie die zugehörigen Verbindungsinformationen zu verwalten. Verschiedene Socket-Klassen repräsentieren die Verbindung auf der Client- wie auf der Serverseite. Eine Socket-Adresse kann z.B. definiert sein durch:

- Identifikationsnummer des Remote-Host
- Portnummer des Remote-Host
- Identifikationsnummer des Local-Host
- Portnummer des Local-Host

Diese Informationen sind allerdings vom verwendeten Protokoll abhängig, so ist die Adress-Information für einen UNIX Domain Socket (wird benutzt für Interprozesskommunikation) ein Dateipfad. Typischerweise handelt es sich bei der Adress-Information im Internet um die IP-Adresse und den Port. Die Vergabe der Portnummern erfolgt beim Verbindungsaufbau. Die Portnummern werden größten teils vom System beliebig vergeben. Ausnahmen sind die so genannten Well-Known-Ports welche von bekannten Applikationen fix verwendet werden.

Well-Known-Ports

In der folgenden Tabelle finden sich einige Ports, die fest definiert Netzwerkdiensten zugeordnet sind[19]:

Portnummer	Dienst	Beschreibung
7	Echo	Zurücksenden empfangener Daten
20	FTP-Data	Dateitransfer (Datentransfer vom Server zum Client)
21	FTP	Dateitransfer (Initiierung der Session durch Client)

[18] http://de.wikipedia.org/wiki/Socket
[19] Eine komplette Liste befindet sich auf: http://www.iana.org/assignments/port-numbers

22	SSH	Secure Shell
23	Telnet	Terminalemulation
25	SMTP	E-Mail-Versand
43	WhoIs	WhoIs-Anfragen
53	DNS	Auflösung von Domainnamen in IP-Adressen
70	Gopher	Gopher-Server
80	HTTP	Webserver
104	DICOM	DICOM Service Class Provider
110	POP3	E-Mail-Abholung
113	ident	Identifikations-Daemon
119	NNTP	Usenet
123	NTP	Network Time Protocol (Protokoll zur Zeitübermittlung)
143	IMAP	E-Mail-Zugriff und -Verwaltung
161	SNMP	Simple Network Management Protocol (Protokoll zur Verwaltung von Netzwerkgeräten)
443	HTTPS	Webserver mit SSL-Verschlüsselung
445	CIFS	SMB/CIFS über TCP
465	SSMTP	Cisco URL Rendesvous Directory for SSM (registriert); Simple Mail Transfer Protocol with SSL (nicht standardisiert)
548	AFP	Apple File Transfer Protocol
587	TLS	email message submission nach RFC 2476
666	DOOM	Doom Gameserver
706	SILC	Secure Internet Live Conferencing
993	IMAPS	IMAP (siehe auch Port 143) mit SSL-Verschlüsselung
995	POP3S	POP3 (siehe auch Port 110) mit SSL-Verschlüsselung
1433	MSSQL	Microsoft SQL-Server

1494	ICA	Citrix ICA Protocol
1701	L2TP	Layer 2 Tunneling Protocol
1719	L2TP	H323
1723	PPTP	Point-to-Point Tunneling Protocol
1863	MSN Messenger	MSN Instant Messaging
1900	SSDP	Simple Service Discovery Protocol
3000	HBCI	
3306	MySQL	MySQL
3389	RDP	Remote Desktop Protocol
5050	YIM	Yahoo! Instant Messaging
5190	ICQ	ICQ Instant Messaging
5222	Jabber	Jabber Instant Messaging
6667-6669	IRC	Internet Relay Chat
8767	TeamSpeak	TeamSpeak VoiceChat

Bei der Kommunikation unterscheidet man das Peer – to – Peer – und Das Client – Server – Prinzip. Beim Peer – to – Peer – Prinzip arbeiten alle Rechner in einem Netzwerk mit den gleichen Rechten, was die Netzstruktur und vor allem z.B. die Struktur einer Firmen – EDV unübersichtlich, ungeordnet und fehleranfällig macht. Aus diesem Grund soll hier nur das Client – Server – Prinzip angesprochen werden. Bei diesem Prinzip arbeitet ein zentraler Rechner als Server (Diener) und bietet einen bzw. mehrere Dienste an. Selbstverständlich können auch mehrere Server in einem Cluster oder aber auch wiederum in einem Netzwerk verteilt Dienste anbieten. Der Server ist ständig dienstbereit und wartet auf Anfragen. Der Client stellt eine Anfrage an den Server und schickt bzw. empfängt Daten.

Client – Programmierung

Die Klasse Socket bietet eine Reihe von Konstruktoren um einen Socket zu initialisieren. Hat man ein Objekt aus der Klasse Socket instanziert, so kann man über geeignete Methoden die entsprechenden Datenströme erzeugen. Nach den Datenoperationen schließt man die Streams und den Socket. Das folgende Programmbeispiel stellt einen HTML – Client dar, der den komplettem HTML – Code einer Webseite auf die Konsole ausgibt:

```
/*Idee nach Guido Krüger: Goto Java 2
*/
import java.net.*;
import java.io.*;

public class Start_Socket_Webseite
{
	public static void main (String args[])
	{
		KonsoleIn eingabe = new KonsoleIn();
		String url,seite;
		do
		{
			System.out.println ("Geben Sie eine Webadresse ein (ende fuer Ende):");
			url = eingabe.readString();
			try
			{
			Socket sock = new Socket (url,80);
			InputStream in = sock.getInputStream();
			OutputStream out = sock.getOutputStream ();
			String s = "GET " + "index.html " + "HTTP/1.0" + "\r\n\r\n";
			out.write(s.getBytes());
			int len;
			byte [] b = new byte [1000];
			while ((len = in.read (b))!=-1)
			{
				System.out.println (len);
				System.out.write (b,0,len);
			}
			out.close();
			in.close();
			sock.close();
			}
				catch (IOException io)
				{
					System.out.println (io.toString());
				}
		}
		while (!url.equalsIgnoreCase ("ende"));
	}
}
```

Serverprogrammierung

Die Serverprogrammierung unterscheidet sich von der Clientprogrammierung nur während des Aufbaus der Kommunikation. Auch sollte ein Server in der Lage sein, mehrere Client gleichzeitig zu bedienen, jedoch hierzu später mehr.

Die Kommunikation von einem Client und einem Server läuft (bewusst nicht technisch betrachtet) folgendermaßen ab: Der Server läuft und wartet darauf, dass ein Client auf ihn zugreift. Hierbei wartet der Server nicht auf alle Clients, sondern nur auf diejenigen, die über ein entsprechendes Tor (port) Anragen stellen. Ein Client öffnet eine Datenverbindung in Richtung des Servers und schickt eine

Anfrage an diesen (diese Anfrage ist meistens genormt, z.B. durch die RFC - Sammlungen[20]). Der Server muss als erstes die Anfrage des Clients akzeptieren. Ist dies geschehen, so kann der Server einen Socket erzeugen und über die entsprechenden Methoden die Datenströme generieren. Nun kann der Server mit dem Client kommunizieren. Auch die meisten Serverantworten, zumindest deren Form sind durch die entsprechenden RFCs genormt. Ausgenommen sind natürlich Server, die man für eigens definierte Zwecke erstellt hat.

Technisch läuft die Socketkommunikation eines Servers wie folgt ab[21]:

1) Server-Socket erstellen
2) binden (bind) des Sockets an eine Adresse (Port) über welche Anfragen akzeptiert werden
3) auf Anfragen warten (listen)
4) Anfragen akzeptieren und damit einen neuen Socket für den entsprechenden Client erstellen
5) Kommunikation auf Basis des erstellten Sockets
6) Socket schließen

Im folgenden Beispiel wird ein Server realisiert, der auf Anfrage die Systemzeit inklusive dem Datum zurückliefert. Die Datenausgabe wird nach der RFC 867 implementiert. Die Klasse DayTimeServer:

```
import java.net.*;
import java.io.*;
public class DaytimeServer
{
	public DaytimeServer ()
	{
		ServerSocket serverSocket = null;
		try
		{
			serverSocket = new ServerSocket (13);
			while (true)
			{
				System.out.println ("Auf neue Verbindung wird gewartet!");
				Socket socket = serverSocket.accept();
				OutputStream ausgabe = socket.getOutputStream();
				String datum = this.getZeit();
				byte daten [] = datum.getBytes();
				ausgabe.write(daten);
				System.out.println ("Daten wurden gesendet!");
				ausgabe.close();
				socket.close();
			}//end f while
		}catch (IOException io){}
	}//end of constructor

	public String getZeit()
	{
		DatumZeit datum = new DatumZeit();
		String zeit = datum.getWochentag() + ", " + datum.getMonat() + " ";
		zeit = zeit + datum.getZahlWochentag() + ", " + datum.getJahr() + " ";
		zeit = zeit + datum.getStunden() + ":" + datum.getMinuten() + ":";
```

[20] http://www.rfc-editor.org/ , hier kann man eine Reihe von RFCs herunterladen und vor allen Dingen suchen!
[21] http://de.wikipedia.org/wiki/Socket

```
                zeit = zeit + datum.getSekunden() + "-" + datum.getZone();
                return zeit;
        }
}
```

Erklärung:

Der Serversocket wird mit dem entsprechenden Konstruktor erstellt. Dem Konstruktor wird der Port übergeben, was dem Binden entspricht. Über die Methode accept() blockiert der Serversocket solange, bis ihn eine Anfrage von einem Client erreicht (listen + accept). Ist dies geschehen, gibt die Methode ein Objekt aus der Klasse Socket zurück. Hier kann nun mit den bekannten Methoden geschrieben und gelesen werden. Über die Methode close() können die Sockets wieder geschlossen werden.

Nachteil dieser Variante ist zum einen, dass der Server während er auf eine Verbindung wartet das Programm blockiert. Zum anderen kann dieser Server immer nur einen Client gleichzeitig bedienen. Um den Server so zu erweitern, dass er mehrere Clients gleichzeitig bedienen kann, muss man für jeden anfragenden Client einen eigenen Thread starten. Das folgende Beispiel befähigt den Server nun mehrere Client gleichzeitig zu bedienen. Der Server blockiert allerdings immer noch solange er auf eine Verbindung wartet. Will man auch dies vermeiden, so muss man den kompletten Server auch in einen eigenen Thread legen. Die Klasse, welche dann den eigentlichen Serverthread startet kann dann weiter arbeiten.

Die modifizierte Klasse DayTimeServer:

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

public class DaytimeServer
{
        public DaytimeServer ()
        {
                ServerSocket serverSocket = null;
                try
                {
                        serverSocket = new ServerSocket (13);
                        while (true)
                        {
                                System.out.println ("Auf neue Verbindung wird gewartet!");
                                Socket socket = serverSocket.accept();
                                String datum = this.getZeit();
                                DaytimeThread thread = new DaytimeThread(socket,datum);
                                thread.start();

                        }//end f while
                }catch (IOException io){}
        }//end of constructor

        public String getZeit()
        {
                DatumZeit datum = new DatumZeit();
                String zeit = datum.getWochentag() + ", " + datum.getMonat() + " ";
                zeit = zeit + datum.getZahlWochentag() + ", " + datum.getJahr() + " ";
```

```
                zeit = zeit + datum.getStunden() + ":" + datum.getMinuten() + ":";
                zeit = zeit + datum.getSekunden() + "-" + datum.getZone();
                return zeit;
        }
}
```

Die Klasse DayTimeThread:

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

public class DayTimeThread extends Thread
{
        private Socket socket;
        private String info;

        public DayTimeThread(Socket socket, String info)
        {
                this.socket = socket;
                this.info = info;
        }

        public void run()
        {
                try
                {
                        OutputStream ausgabe = socket.getOutputStream();
                        byte daten [] = info.getBytes();
                        ausgabe.write(daten);
                        System.out.println ("Daten wurden gesendet!");
                        ausgabe.close();
                        socket.close();
                }catch (IOException io){}
        }
}
```

Erklärung:

In der Endlosschleife wartet der Server auf eine eingehende Verbindung. Greift ein Client auf den Server zu, so legt der Server eine neue Instanz der Klasse DayTimeThread an und startet diesen. Dieser Thread öffnet die Streams und gibt die entsprechenden Daten aus. In dieser Zeit kann der eigentliche Server neue Anfragen entgegen nehmen.

Schnittstellen

Im Rahmen des Lehrplanes BG12, Leistungskurs 12.2: Datenkommunikation ist vorgesehen, mit Hilfe der parallelen und seriellen Schnittstelle Kommunikationsprogramme zu erstellen. mit diesen Programmen kann man entweder über ein entsprechendes externes Gerät, welches an einer Schnittstelle angeschlossen ist, steuern bzw. regeln oder mit einem anderen Computer kommunizieren, Daten austauschen etc.

Hierbei sind laut Lehrplan die parallele Schnittstelle und die serielle Schnittstelle zu besprechen. Warum die USB – Schnittstelle nicht berücksichtigt wurde ist nicht nachvollziehbar. Aus diesem Grund werde ich hier auch auf diese Schnittstelle eingehen.

Die parallele Schnittstelle

Der Computer ist über eine Schnittstelle mit der Außenwelt bzw. dem peripheren Ein- oder Ausgabegerät verbunden, mit dessen Hilfe dem Anwender ein Datenaustausch mit dem PC erst möglich wird. Die Regeln für den Datenaustausch werden durch die Schnittstelle genau festgehalten und technisch umgesetzt. Die Datenübertragung erfolgt parallel. Bei der parallelen Übertragung können acht Bits als ein komplettes Byte über die parallelen Leitungen transportiert werden. Kontrolliert wird der Austausch der Daten zwischen Rechner und Peripherie mit Hilfe von Steuerleitungen (dieses Verfahren wird auch Handshakingverfahren genannt). Die parallele Schnittstelle erreicht bei der Datenübertragung höhere Geschwindigkeiten als die serielle Schnittstelle, sie ist jedoch nicht zur Überbrückung längerer Entfernungen geeignet. Zunächst wurde die parallele Schnittstelle hauptsächlich zum Anschluss von Druckern verwendet. Hier hat lange Zeit die so genannte Centronics - Schnittstelle des gleichnamigen Druckerherstellers ohne Normierung einen Quasi-Standard dargestellt. Bei den ersten parallelen Schnittstellen war nur die Übertragung der Daten vom Rechner an die angeschlossene Peripherie möglich. Nach und nach haben sich jedoch bidirektionale Schnittstellen durchgesetzt, um auch Geräte wie z.B. Scanner anschließen zu können.

Für parallele Schnittstellen verwendet man oft (besonders in der Microsoft Welt) die Bezeichnung „LPT1“ , „LPT2“ usw. Die Kontakte werden Pins genannt. Man kann sich eventuell mit einem Zwischenadapter helfen, wenn der Anschluss nicht die gleiche Anzahl von Verbindungen aufweist. Vom System werden Portadressen vergeben, über die die Schnittstellen adressiert werden. Die Schnittstellen, englisch I/O-Ports, werden von einem I/O-Controller gesteuert, der sich entweder direkt auf dem Motherboard befindet oder als Steckkarte vorhanden ist. Der I/O-Controller ist über das Bussystem direkt mit der CPU verbunden und übersetzt die CPU-Befehle in entsprechende Signale. Die Schnittstellenkarten werden über DIP - Schalter oder Jumper konfiguriert.

Man sollte sich über die bisherige Konfiguration des PC immer vor einer Anwendung informieren.

Schnittstellenarten

Zurzeit sind folgende Schnittstellenarten mehr oder weniger verbreitet im Einsatz:

- **SPP**

  Diese Schnittstellen sind unidirektional und es ist keine Übertragung von Daten zum Rechner möglich. Die Steuerleitungen haben eine feste Bedeutung, die auf die Anwendung mit Druckern angepasst ist.

- **BPP**

  BPP steht für **bidirectional parallel port**. Diese Schnittstellen entsprechen den herkömmlichen unidirektionalen Modellen, es können aber auch externe Signale an den Datenleitungen von der Software abgefragt werden. Ein Übertragungsprotokoll für den bidirektionalen Betrieb wurde nicht spezifiziert und musste bei entsprechenden Anwendungen selbst definiert werden.

- **PS/2**

Diese Schnittstellen wurden von IBM in den gleichnamigen PS/2-Rechnern eingesetzt. Im Vergleich zu den BPP - Schnittstellen wurden hier Tri – State - Ausgänge für die Datenleitungen verwendet; das macht das Umschalten zwischen Ein- und Ausgabemodus einfacher.

- **EPP**

Die EPP - Schnittstelle wird in der IEEE-1284 Norm spezifiziert, wobei EPP für **enhanced parallel port** steht. Hiermit wurde erstmals ein festes Übertragungsprotokoll definiert, welches für einen Datenaustausch allgemein gehalten wurde. Die Steuerleitungen bekamen hierbei eine neue Bedeutung, denn das Handshaking wurde in die Hardware der Treiberbausteine integriert.

- **ECP**

ECP steht für **enhanced capabilities port**. Hierbei wird die gesamte Datenübertragung in die Hardware implementiert. Der Datenaustausch mit der Software erfolgt über FIFOs mit bis zu maximal 1024 Bytes. Zusammen mit dem EPP - Port bildet der ECP-Port den zurzeit aktuellen Standard. Da zurzeit die EPP- und ECP-Schnittstellen die aktuellsten und am weitesten verbreitete Versionen sind, sollten alle Neuentwicklungen sich ausschließlich auf diese beziehen. Aus Kompatibilitätsgründen sollte die Ansteuerungssoftware jedoch so ausgelegt werden, dass auch alle anderen Schnittstellen emuliert werden können. Das EPP - Protokoll kann sehr einfach von Mikrokontrollen emuliert werden, so dass für diese Schnittstelle entwickelte Geräte auch an Systeme angeschlossen werden können, die nicht auf einem PC basieren. Als Anschlüsse werden drei verschiedene Stecker verwendet, die in der aktuellen IEEE1284-Norm neue Bezeichnungen erhielten.

- **IEEE 1284-A**

In den heute üblichen PCs wird als Anschluss ein 25-poliger Sub – D - Stecker benutzt. Dieser besitzt aufgrund der geringen Polzahl nicht für jede Daten- und Signalleitung eine Masseleitung zur Abschirmung, was eine geringere Störfestigkeit ergibt. Für die meisten Anwendungen reicht dies jedoch vollkommen aus.

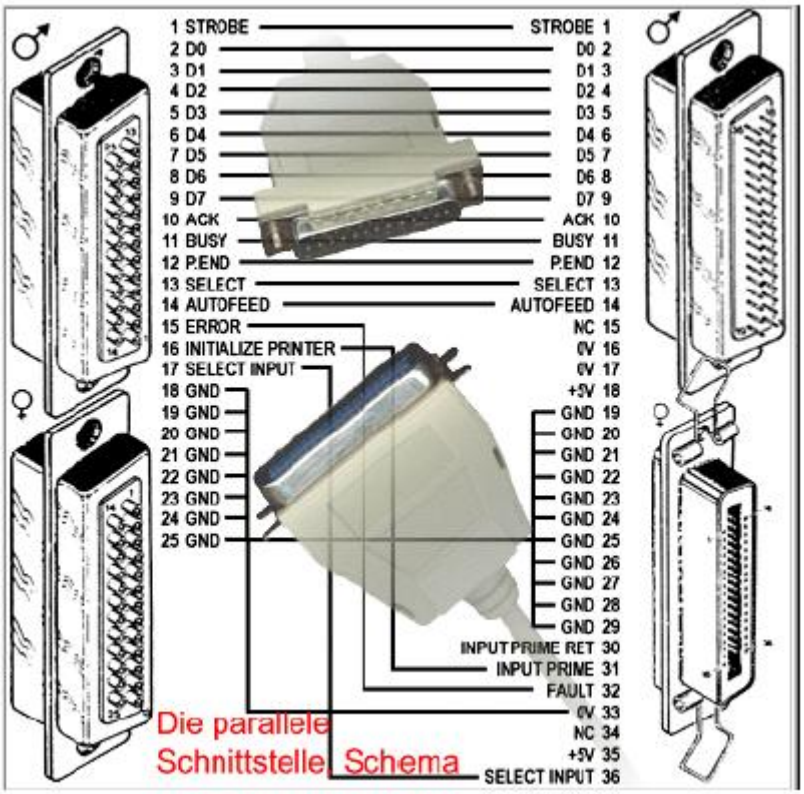
- **IEEE 1284-B**

Ursprünglich wurden 36-polige Stecker verwendet, wie sie von der Firma Centronics eingeführt wurden. Beim hierbei verwendeten Flachbandanschlusskabel wurde zwischen jede Daten- und Signalleitung eine Masseleitung gelegt, was das Übersprechen der Signale sowie äußere Einflüsse erheblich reduziert.

- **IEEE 1284-C**

Die modernste Form des Anschlusses verwendet wieder einen 36-poligen Stecker, der aufgrund einer extrem kleinen Bauform trotz der hohen Zahl der Anschlüsse kleiner als der herkömmliche Sub-D-Stecker ist. Wie beim IEEE 1284-B Stecker besitzt jede Daten- und Steuerleitung eine eigene Masseleitung zur Abschirmung.

Schema der Pinbelegung einer parallelen Schnittstelle



Die Pinbelegung der 25 PIN SUB-D Buchse am PC

Pin-Nr.	Signalbezeichnung	Funktionsbeschreibung
1	STROBE	Datenübernahmeimpuls
2	DATA 1	Signale für Paralleldatenbits 1 bis 8
3	DATA 2	
4	DATA 3	
5	DATA 4	
6	DATA 5	
7	DATA 6	

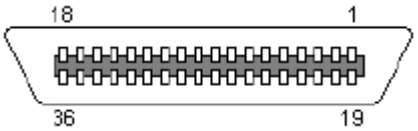
8	DATA 7	
9	DATA 8	
10	ACK	Daten werden übernommen
11	BUSY	Keine neuen Daten
12	PAPER OUT	Dem Drucker fehlt Papier
13	SELECTED	Der Drucker meldet ON-LINE
14	AUTO FEED	Automatischer Zeilenvorschub
15	FAULT	Störung beim Drucker
16	RESET	Druckerinitialisierung
17	SELECT IN	Den Drucker ON-LINE schalten
18 - 25	GND	Masse-Rückleitungen der verschiedenen Signale.

Die Pinbelegung einer parallelen Schnittstelle am Drucker

Pin-Nr.	Signalbezeichnung	Richtung	Funktionsbeschreibung
1	STROBE	Zum Drucker	Zum Einlesen der Daten! (normalerweise H-Pegel). Bei L-Pegel werden Daten gelesen
2	DATA 1	Zum Drucker	Signale für Paralleldatenbits 1 bis 8 H = HIGH = logisch 1 L = LOW = logisch 0
3	DATA 2	Zum Drucker	
4	DATA 3	Zum Drucker	
5	DATA 4	Zum Drucker	
6	DATA 5	Zum Drucker	
7	DATA 6	Zum Drucker	
8	DATA 7	Zum Drucker	
9	DATA 8	Zum Drucker	
10	ACK	Vom Drucker	Ein L-Pegel bestätigt den Datenempfang bzw. Die Ausführung einer Funktion.
11	BUSY	Vom Drucker	Bei H-Pegel des Signals ist kein Datenempfang möglich. Ein L-Pegel signalisiert, daß der Drucker empfangsbereit ist.
12	PAPER OUT	Vom Drucker	Ein H-Pegel des Signals zeigt an, daß der Papiervorrat erschöpft ist.
13	[SELECTED]	Vom Drucker	Ein H-Pegel des Signals zeigt an, daß der Drucker ON-LINE geschaltet ist.
14	[AUTO FEED]	Auf Masse gelegt	Nicht benutzt!
15	(NC)	--	Nicht belegt!
16	SIGNAL GND (0 V)	--	Signalerdung
17	CHASSIS GND	--	Gehäuseerdung (Masse)
18	[+5 V EXT.]	Vom Drucker	Externe Gleichspannung +5 Volt
19-30	GND	--	Masse-Rückleitungen der verschiedenen Signale.
31	RESET	Zum Drucker	Signal zur Rückstellung der Druckerlogik. Bei L-Pegel wird der Drucker initialisiert.
32	FAULT	Vom Drucker	Dieses Signal wechselt auf L-Pegel, wenn der Drucker eine Störung erkennt.
33	EXT GND (0 V)	--	Externe Erdung
34	(NC)	--	Nicht benutzt!
35	[+5 V]	--	H-Pegel
36	[SELECT IN]	Zum Drucker	Immer H-Pegel!

[ ] = Eingeklammerte Signale werden nicht von jedem Drucker unterstützt!

(NC) = **N**ot **C**onnected



36 PIN Centronics Buchse am Drucker

25 PIN SUB-D Buchse am Computer

Die Centronics - Schnittstelle

Die Centronics -Schnittstelle wurde ursprünglich als Druckerschnittstelle konzipiert, dient beim PC aber auch zum Anschluss anderer Peripheriegeräte wie Streamer usw. Die Schnittstelle arbeitet mit TTL-Pegel und unidirektional. Der bidirektionale Betrieb ist nur möglich, wenn eine Schnittstelle gemäß IEEE 1284 vorhanden und im SETUP/BIOS aktiviert ist. Ansonsten kann ein bidirektionaler Betrieb auch durch Verwendung der Eingänge FAULT, SELECT, PAPER EMPTY und BUSY, bei entsprechender Programmierung, realisiert werden.

Beispiel einer Pinbelegung der Centronics - Schnittstelle und Funktionsbeschreibung

Um das Kommunikationssystem zu verstehen, wollen wir uns die Funktionen etwas genauer ansehen, die die verschiedenen über die Steckerpins übertragenen Signale

erfüllen. Pin 1 überträgt das STROBE - Impulssignal vom Computer zum Drucker.

Es wird normalerweise vom H-Pegel gehalten. Stehen Daten zur Übertragung an den Drucker bereit, wird dieses Signal für mindestens 0,5 Mikrosekunden auf L-Pegel gesetzt. Erkennt der Drucker diesen Impuls am STROBE - Pin , liest er die über Pin 2 bis 9 kommenden Daten ein. Jede Signalleitung trägt ein Informationsbit. Eine logische „1“ wird durch ein Signal mit H-Pegel, eine logische „0“ durch L-Pegel dargestellt. Der Computer hält diese Signalpegel mindestens 0,5 Mikrosekunden und

auch nach der Einspeisung des STROBE - Impulses aufrecht. Hat der Drucker die Daten richtig empfangen, setzt er Pin 10 ca. 5 Mikrosekunden lang auf L-Pegel. Dieses Signal bestätigt den Datenempfang und wird auch als ACK - Signal bezeichnet

(acknowledge = bestätigen). Das Signal an Pin 11 gibt an, ob der Drucker für den Datenempfang bereit ist. Wenn dieses BUSY - Signal L-Pegel hat, ist der Drucker empfangsbereit. Es wechselt auf H-Pegel während der Datenübertragung, während des Druckens, im OFF – LINE - Betrieb oder bei einer Störung. Das Papierende wird durch ein Signal (PAPER OUT) mit H-Pegel an Pin 12 angezeigt. Dieser Pin wird jedoch auf L-Pegel gehalten, wenn DIP - Schalter 1-5 ausgeschaltet ist. Im ONLINE - Betrieb liegt an Pin 13 ein Signal mit H-Pegel an. Dieses SELECTED – Signal zeigt dem Computer, wann der Drucker empfangsbereit ist. Pin 14, 15, 34 und 35 werden nicht benutzt. Pin 16, 17, 19 – 30 und 33 sind geerdet. Pin 18 ist an die Drucker-Gleichspannung +5 V angeschlossen. Pin 31 wird für die Druckerinitialisierung benutzt. Wenn dieses Signal (RESET) L-Pegel hat, wird der Drucker initialisiert. Mit Pin 32 werden Druckerstörungen angezeigt. Liegt eine Störung vor, wechselt das ERROR - Signal auf L-Pegel.

Zugriff unter JAVA auf die parallele Schnittstelle

Es gibt unter JAVA zwei Möglichkeiten, auf die parallele Schnittstelle zuzugreifen:

1. Über eine spezielle DLL (Dynamic Link Library) und das JAVA Native Interface

2. Über die Communication – API der Firma SUN

Die 2. Möglichkeit ist sicher die sauberere, da hier der Programmierer nicht über direktes Ansprechen einer Adresse Zugriff auf die Hardware eines Systems erhält. Will man jedoch wirklich einzelne Bits der parallelen Schnittstelle ansprechen oder abfragen und entspricht die angeschlossene Hardware nicht den Vorgaben der API (z.B. muss über das Acknowlege – Bit eine Bestätigung des externen Gerätes erfolgen), so bleibt dem Programmierer keine andere Möglichkeit, als eine DLL zu verwenden. Die in diesen Beispielen verwendete DLL ist im Internet unter der Adresse http://web.bvu.edu/faculty/schweller/ frei erhältlich[22].

Installationsanleitung zur DLL: jnpout32pkg.dll

Die Datei jnpout.zip unter http://www.jochen-ferger.de oder unter http://web.bvu.edu/faculty/schweller/ herunter laden. Beim Auspacken entsteht ein Verzeichnis jnpout32. Dieses Verzeichnis ist in das aktuelle Arbeitsverzeichnis zu kopieren. Die Datei jnpout32pkg.dll kopieren Sie bitte in das system32 – Verzeichnis des Systems (funktioniert unter Windows 2000 und Windows XP, alternativ kann die DLL auch im aktuellen Arbeitsverzeichnis liegen). Anschließend kann man seine Applikationen im Arbeitsverzeichnis erstellen.

Die Klasse pPort stellt unter anderem einen Standardkonstruktor zur Verfügung. Nach dem Erzeugen des Objektes kann man über die Methoden: `public void output(short port, short value)` ein Byte an die Adresse port schicken und über die Methode `public short input(short port)` ein Byte von der Adresse port einlesen.

Hinweis:

Die Klassen sind vorkompiliert, es ist teilweise notwendig, die Klassen neu zu kompilieren. Die Klasse pPort ist nicht unbedingt notwendig. Prinzipiell reicht es, die einzelnen Register der parallelen Schnittstelle über die Methoden der Klasse ioPort anzusprechen. Hierfür wird ein Objekt über den Standardkonstruktor dieser Klasse erzeugt. Anschließend bieten die Methoden `public native void Out32(short PortAddress, short data)` und `public native short Inp32(short PortAddress)` Möglichkeiten Bytes zu senden oder auszulesen.

Die drei Leitungsgruppen des Druckerportes[23]

Angenommen die parallele Schnittstelle ist im BIOS unter der Adresse 378hex eingetragen.

Nun kann man unter dieser Adresse die 8 Datenleitungen des Druckerportes ansteuern.

Name	Pin	Wertigkeit
D0	2	1

[22] In diesem Skript wird nicht beschrieben, wie man diese DLL´s und die entsprechenden JAVA – NATIVE – Klassen erzeugt. Eine Anleitung hierzu findet man im Buch: **Java ist auch eine Insel** von Christian Ullenboom.

[23] Diese Information stammt von einem Arbeitsblatt: Interfacetechnik des Kollegen Friedhelm Leber von der Friedrich Dessauer Schule, Limburg

D1	3	2
D2	4	4
D3	5	8
D4	6	16
D5	7	32
D6	8	64
D7	9	128

Unter der Adresse 379hex kann man die Eingabesignale: ERROR, SLCT, PE, ACK und BUSY abfragen. Wichtig hier ist, dass die Eingabesignale unterschiedliche Wertigkeiten im Abfragebyte haben:

Name	**Pin**	**Wertigkeit**	**Invertierung**
ERROR	15	8	nein
SLCT	13	16	nein
PE	12	32	nein
ACK	10	64	nein
BUSY	11	128	ja

Die ersten drei Bytes sind undefiniert. Will man also wissen, ob z.B. das ERROR – Bit gesetzt ist, muss man einen Algorithmus entwickeln, der aus einem abgefragten Byte genau dieses Bit herausfiltert. Auch ist zu beachten, dass das BUSY – Bit invertiert geliefert wird!

Unter der Adresse 37A hex kann man den 4 – bit – breiten Ausgabeport der parallelen Schnittstelle ansprechen:

Name	**Pin**	**Wertigkeit**	**Invertierung**
STROBE	1	1	ja
AF	14	2	ja

INIT	16	4	nein
SLCT IN	17	8	ja

Warnung:

Die parallele Schnittstelle ist nicht gegen Kurzschlüsse abgesichert, daher kann es beim unvorsichtigen Umgang mit dieser Schnittstelle schnell zu Defekten an derselben kommen.

Das HIBS – Interface für die parallele Schnittstelle[24]; Zugriff über die jnpout32 - DLL

Als externes Gerät verwenden wir in unserer Schule das HIBS – Interface zu erhalten bei der Firma Knobloch. Dieses Interface ermöglicht es, neben der Ausgabe eines Datenbytes mit Hilfe eines kleinen Programmiertricks auch ein Datenbyte einzulesen.

Die Interfaceeingänge E1, E2, E3 und E4 kann man von Adresse adr + 1 abfragen, wenn man vorher auf der Adresse adr + 2 das Strobe – Bit auf 1 gesetzt hat. Die anderen vier Eingangssignale E5 bis E8 erreicht man auf Adresse adr + 1, wenn man vorher auf der Adresse adr + 2 das Strobe – Bit auf den Wert 0 gesetzt hat.

Die folgende Klasse kapselt den Zugriff auf ein HIBS – Interface:

```
import jnpout32.*;

public class HIBS
{
        pPort port;
        short adresse;
        public HIBS (short adresse)
        {
                port = new pPort ();
                this.adresse = adresse;
        }

        public void datenSenden (short daten)
        {
                port.output (adresse,daten);
        }

        public short datenEinlesen ()
        {
                adresse ++;
                adresse ++;
                port.output (adresse,(short)1);
                adresse --;
                short e1e4daten =  port.input (adresse);
```

[24] Die Grundadresse der parallelen Schnittstelle wird von nun an als adr bezeichnet. Das Interface kann unter der Adresse: http://www.knobloch-gmbh.de erworben werden

```
            adresse ++;
            port.output (adresse,(short)0);
            adresse --;
            short e5e8daten =  port.input (adresse);
            adresse --;

            return this.datenUmrechnen (e1e4daten,e5e8daten);
        }

        private short datenUmrechnen (short e1e4daten,short e5e8daten)
        {
            int daten = 0;
            if ((e1e4daten & 8)== 8)
            daten = daten+1;

            if ((e1e4daten & 32)== 32)
            daten = daten+2;

            if ((e1e4daten & 64)== 64)
            daten = daten+4;

            if ((e1e4daten & 128)== 0)
            daten = daten+8;

            if ((e5e8daten & 8)== 8)
            daten = daten+16;

            if ((e5e8daten & 32)== 32)
            daten = daten+32;

            if ((e5e8daten & 64)== 64)
            daten = daten+64;

            if ((e5e8daten & 128)== 0)
            daten = daten+128;
            return (short)daten;
        }
}
```

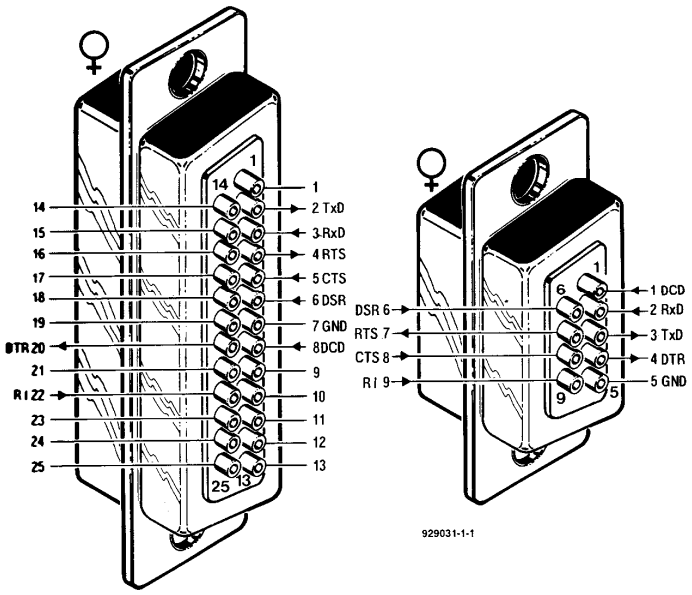
Die serielle Schnittstelle

Die serielle Schnittstelle oder RS232 - Schnittstelle ist in jedem PC vorhanden. Sie wird oft als COM1, COM2 usw. bezeichnet. Ihr ursprünglicher Zweck liegt in der Datenübertragung mit einem Modem, jedoch werden vielfach auch Geräte wie Maus, Messgeräte etc. an die serielle Schnittstelle angeschlossen.

Die serielle Schnittstelle hat gegenüber anderen Schnittstellen einige Vorteile:

- Die Schnittstelle ist relativ sicher gegen versehentliche Zerstörung.
- Geräte dürfen im eingeschalteten Zustand des Computers angeschlossen werden.
- Über die Schnittstelle kann eine Stromversorgung einfacher externer Geräte erfolgen.

Die Anschlussbelegung der RS232 – Schnittstelle in 25poliger und 9poliger Ausführung von der Lötseite her gesehen.



Die elektrischen Eigenschaften der Schnittstelle sind durch die RS232 – Norm festgelegt: Der logische Zustand 1 (mark) wird durch –12V der logische Zustand 0 (space) durch +12V definiert. Alle Ausgänge sind kurzschlussfest und können bis zu 10mA liefern. Die Eingänge besitzen eine Eingangs-widerstand von 10kΩ.

Tabelle der Anschlüsse und deren Bedeutung

Anschluss	**Bezeichnung**	**Funktion**
TXD	Transmit Data	Sendedaten
RXD	Receive Data	Empfangsdaten
RTS	Request To Send	Sendeteil einschalten
CTS	Clear To Send	Sendebereitschaft
DSR	Data Set Ready	Betriebsbereitschaft
GND	Ground	Betriebserde
DCD	Data Carrier Detect	Empfangssignalpegel
DTR	Data Terminal Ready	Endgerät betriebsbereit
RI	Ring Indicator	Ankommender Ruf

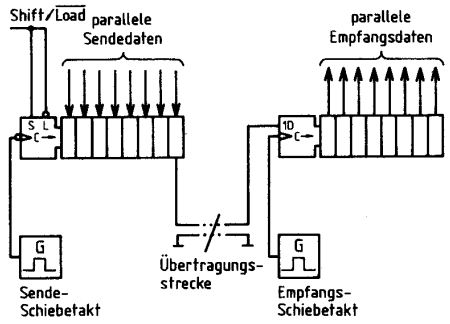
Die eigentlichen Daten werden über die TXD - und RXD - Leitungen übertragen. Zusätzlich ist für eine bidirektionale Datenübertragung noch die Masseleitung (Betriebserde) als Bezugsgröße erforderlich. Die anderen Leitungen werden üblicherweise als Handshake – Leitungen bezeichnet, da sie bei Datenübertragungen die Rolle der Vorbereitung und Quittierung von Datenübertragungen spielen.

Serielle Datenübertragung

Bei der seriellen Datenübertragung werden die einzelnen Bits wie der Name schon aussagt zeitlich nacheinander über eine Leitung übertragen. Hierbei entsteht das Problem, dass der Senderechner mit dem Empfangsrechner synchronisiert werden muss. Das heißt, Der Empfangsrechner muss wissen, in welchen zeitlichen Abständen der Senderechner Bits sendet.

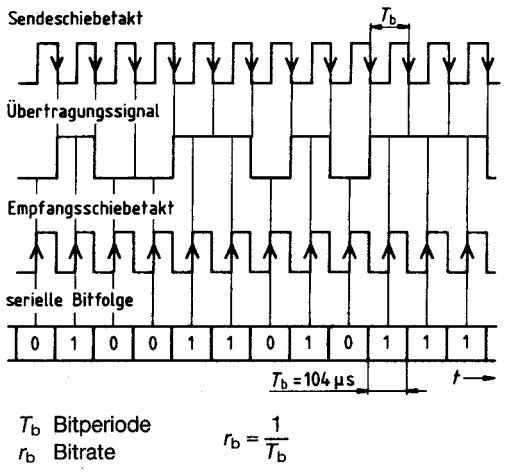
Synchrone Datenübertragung

Bei der Synchronen Datenübertragung senden und empfangen die jeweiligen Rechner mit exakt dem gleichen Takt, welcher festgelegt werden muss. Zusätzlich muss noch gewährleistet sein, dass der Takt des Empfangsrechners eine Phasenverschiebung hat, so dass der Empfangsrechner nicht genau während eines Bitwechsels des Senders empfängt. Das folgende Bild zeigt das Prinzip einer solchen Datenübertragung:



Die Daten werden im Senderechner parallel in ein Schieberegister geladen. Mit Hilfe des Sendetaktes werden die Daten nacheinander über die Übertragungsstrecke zur Empfangsstation geschickt.

Dort werden die Bits mit Hilfe des Empfangstaktes (der die gleiche Frequenz des Sendetaktes hat) in das entsprechende Schieberegister geladen. Die Daten können dann vom Empfangsrechner parallel abgerufen werden. Wie man im folgenden Diagramm sieht,



wird durch den Empfangstakt das ankommende serielle Signal genau in der Mitte abgetastet. Die Bitperiode ist die Zeit, die benötigt wird um ein Bit abzutasten. Beträgt die Zeit z.B. genau 104µs, so könne in einer Sekunde 9615 Bits übertragen werden.

Vorteil:

Diese Übertragung erlaubt es unbegrenzt Bits vom Sender zu empfangen, so fern die anschließende Verarbeitung dieser Daten schnell genug erfolgt.

Nachteile:

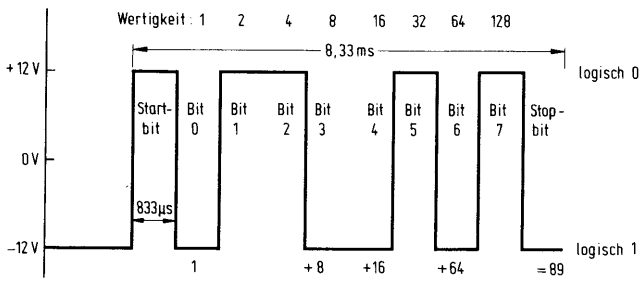
Die technische Realisierung dieser Übertragungsart erweist sich als sehr schwer. Außerdem ist oft nicht bekannt, wann an welchen Rechner oder welches System Daten versendet werden, womit ein zeitlicher Standart zu schaffen wäre, welcher sich nicht ändern dürfte (d.h. es müsste feste Zeiten geben, an denen eine Datenübertragung beginnen darf).

Es erscheint also sinnvoll, eine Datenübertragung zu realisieren, bei der Senderechner und Empfangsrechner unabhängig voneinander laufen und die unabhängig von Übertragungsstartzeitpunkten funktional ist.

Die asynchrone Datenübertragung

Um Daten zu übertragen, muss ein Empfänger wissen, wann die Übertragung beginnt. Hierzu ist es notwendig festzulegen, welchen logischen Zustand die Datenleitung führt, wenn keine Daten übertragen werden. Bei der RS232 – Schnittstelle wird der Ruhezustand der Datenleitung mit dem logischen Signal 1 (-12V) festgelegt. Sollen Daten übertragen werden, so wird dieser Vorgang mit einem Bit,. Welches den logischen Zustand 0 (+12V) hat, eingeleitet. Diese Bit wird als **Startbit** bezeichnet. Nach dem Startbit erfolgt die eigentliche Datenübertragung. Ausgegangen von zu übertragenen 8 – Bit – Blöcken, folgt nach dem Startbit die 8 Bit, welche die eigentliche Information beinhalten. Danach wird für die Dauer von mindestens einer Bitzeit die Leitung wieder auf logisch 0 gesetzt. Dieses Bit wird als **Stopbit** bezeichnet. Anschließend kann eine neue Übertragung, eingeleitet mit einem Startbit, durchgeführt werden. Dieses Verfahren nennt man Start – Stop – Übertragung. Hierdurch ist der Anfang einer Zeichenübertragung für den Empfänger erkennbar, weshalb man dieses Verfahren auch **Zeichensynchronisation** nennt.

Beispiel:

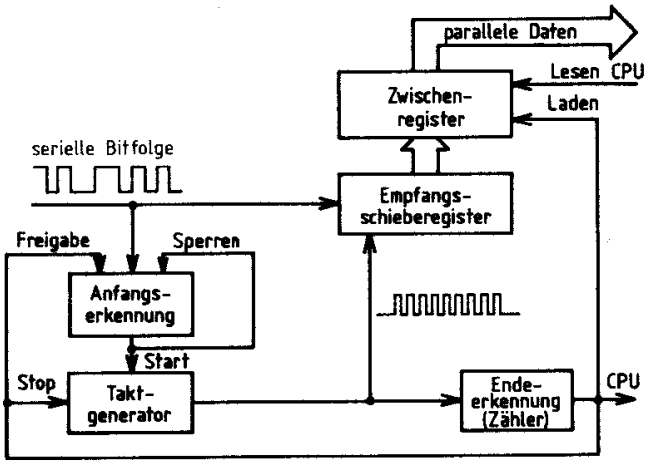


In diesem Beispiel wird das Zeichen Y (ASCII 89) übertragen.

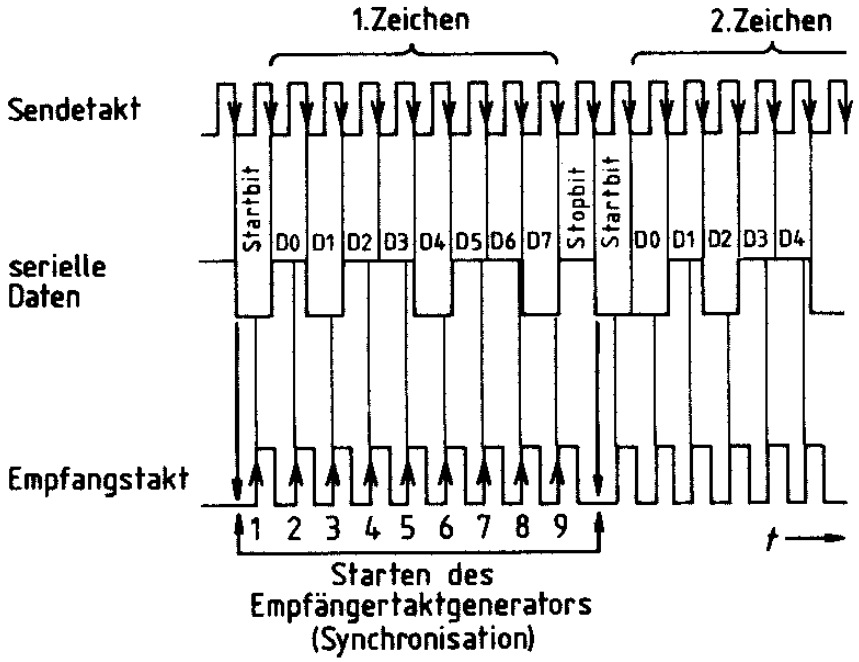
Der Sende- und Empfangstakt wird dadurch synchronisiert, dass der Taktgenerator des Empfängers mit der ersten Flanke des Startbits gestartet wird. Der Taktgenerator startet mit dem logischen Zustand 0. Nach halber Bitrate wechselt das Signal des Taktgenerators auf den logischen Zustand 1. Mit dieser Flanke wird das erste Bit (in diesem Fall das Startbit) genau in der

Mitte abgetastet. Nach neun Abtastungen ist das zu empfangende Zeichen im Schieberegister geladen und der Taktgenerator wird wieder gestoppt. Anschließend beginnt der Vorgang von vorne.

Prinzip der Synchronisation zwischen Sende- und Empfangstakt:

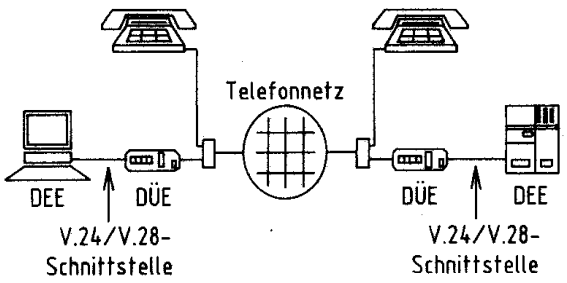


Das asynchrone Übertragungsverfahren funktioniert auch, wenn Sendetakt und Empfängertakt bis zu einem gewissen Grade voneinander abweichen. Die Frequenz-abweichung darf so groß sein, dass das letzte Bit gerade noch abgetastet werden kann. Die folgende Abbildung soll dies verdeutlichen:



Verbindung: Rechner - Rechner

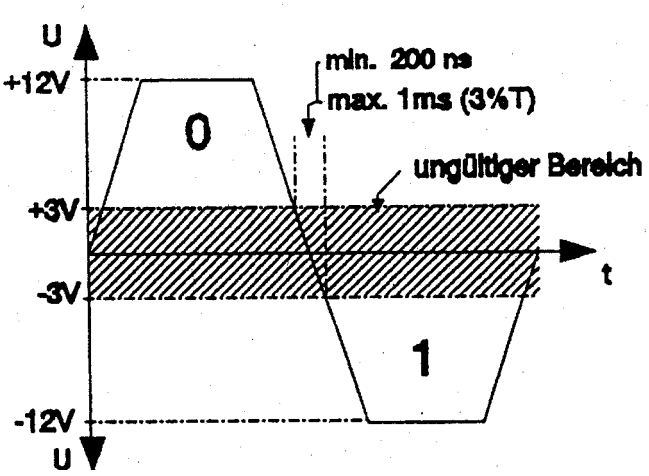
Eine der wichtigsten Anwendungen der seriellen Schnittstelle ist die der Datenkommunikation mit einem anderen Rechner über zwei Modems.



Die Grafik rechts zeigt den Aufbau einer solchen Verbindung. Die Bezeichnung DÜE steht für Datenübertragungseinrichtung (Englisch: DCE Data Circuit – Termination Equipment). Die Bezeichnung DEE steht für Datenendeinrichtung (Englisch DTE Data Terminal Equipment). Die Schnittstelle zwischen DEE und DÜE ist eine V.24 – Schnittstelle.

Die V.24 – Schnittstelle (RS – 232C – Schnittstelle)

Die Eigenschaften dieser Schnittstelle sind durch das CCITT (Comite Consultatif International Telegraphique et Telephonique) festgelegt. Bei dieser Schnittstelle entspricht ein Spannungssignal von 3V bis 12 V einer logischen 0, ein Signal zwischen –3V und –12V entspricht einer logischen 1. Zwischen den Spannungswerten –3V und 3V entsteht ein ungültiger Bereich, der in der Zeit von 200ns bis maximal 1ms durchschritten werden darf.



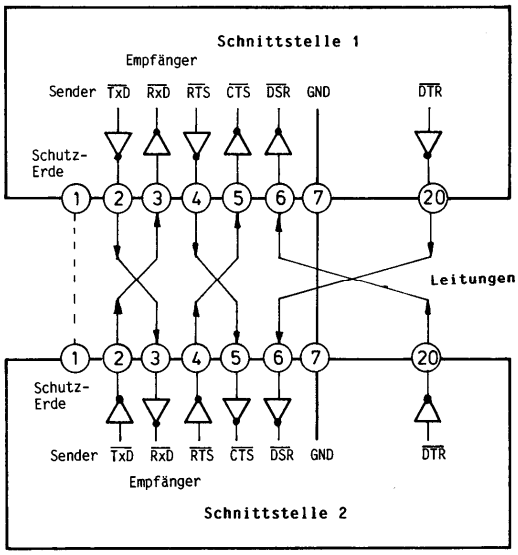
Die Anschlüsse der V.24 – Schnittstelle sind fest definiert. Auch hier gibt es Anschlüsse, die der reinen Datenübermittlung oder dem Datenempfang dienen.

Gleichzeitig gibt es Anschlüsse, die vordefinierte Zustände oder Quittierungssignale übertragen (Handshake – Leitungen). In der folgenden Tabelle ist ein Auszug aus der Liste der V.24 – Schnittstellenleitungen dargestellt, die auch in der DIN 66020 festgelegt sind:

Die direkte Verbindung zweier DEEs

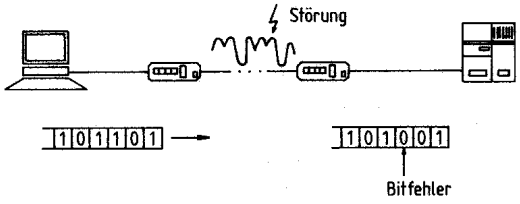
Auch ist es über diese Schnittstelle möglich, zwei DEEs direkt zu verbinden (Nullmodemverbindung).

Hierbei werden die entsprechenden Leitungen der beiden V.24 – Schnittstellen zum Teil gegenseitig über Kreuz angeschlossen. Man spricht von einer vollständigen Kopplung, die mit einem Überkreuzkabel oder auch Nullmodemkabel zu realisieren ist. Für eine Datenübertragung reicht es jedoch aus, wenn die Leitungen GND direkt, und die Leitungen TxD und RxD über kreuz angeschlossen werden (Drei – Draht – Verbindung).



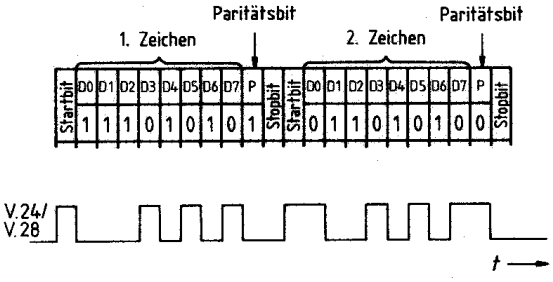
Fehlerüberwachung

Wie abgebildet, kann bei einer Datenübertragung ein Fehler auftreten. Nicht selten ändert sich bei einem solchen Fehler ein Bit. Da der Empfänger nicht erkennen kann, ob ein Fehler aufgetreten ist oder nicht, muss die Übertragung selber dafür sorgen. Eine Möglichkeit ist die Prüfung der Parität. Hierbei wird ein zusätzliches Bit übertragen, welches auf die Anzahl der High – Signale im



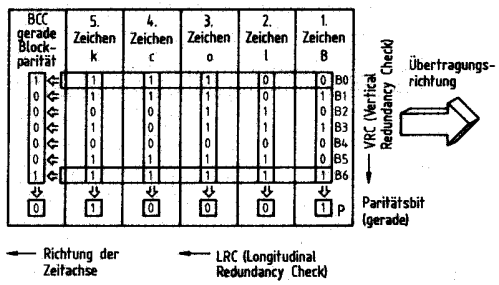
zu übertragenden Zeichen abgestimmt ist. Hierbei unterscheidet man zwischen **gerader (Parity even) Parität** und **ungerader (Parity odd) Parität**.

Bei der geraden Parität ergänzt das Parity – Bit die Summe aller Datenbits inklusive des Parity – Bits so, dass die Anzahl der High – Pegel gerade ist. Beim ungeraden Parity – Bit sorgt das Parity – Bit dafür, dass die Summe aller High – Signale von Datenwort und Parity - Bit selber ungerade ist. Problematisch ist hierbei sicher, dass es bei einer



gerade Parität (Parity Even):
Σ Datenbits + Paritätsbit ⇒ gerade

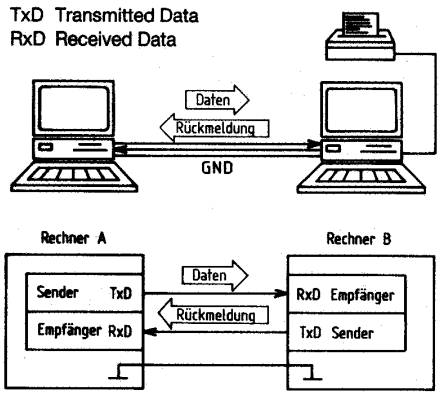
Störung auch zu einem Zwei – Bit – Fehler kommen kann, welchen dann das Parity – Bit nicht identifiziert. Eine Möglichkeit die Fehlerkorrektur zu verfeinern ist die **blockweise Fehlerüberwachung**: Bei der blockweisen Fehlerüberwachung wird wie im Bild eine



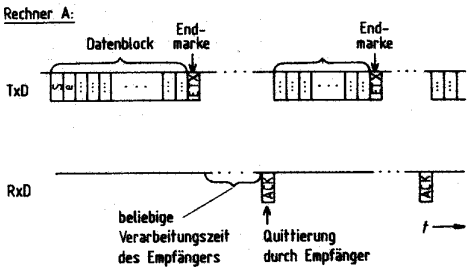
Längsprüfsumme aus mehreren Zeichen gebildet. Diese Summe wird zusätzlich nach dem Datenblock übertragen. Somit können auch einige Mehrfach – Bit – Fehler erkannt werden.

Datenflusskontrolle

Bei einer Datenübertragung kann es vorkommen, dass Daten die mit einer bestimmten Geschwindigkeit empfangen werden, nicht genau so schnell im Empfänger verarbeitet werden können. Zum Beispiel kann über eine Datenübertragung ein Text übermittelt werden, der anschließend ausgedruckt wird. Klappt aus einem bestimmten Grunde der Ausdruck nicht (Papier fehlt, Drucker zieht Papier falsch ein etc.), so werden weiter gesendete Daten evtl. nicht mehr berücksichtigt. Es kommt zu einem Datenüberlauf. Um dies zu verhindern, muss der Datenfluss kontrolliert werden (flow control). Dies kann zum Beispiel dadurch geschehen, dass nach einem bestimmten Kriterium der Empfänger eine Rückmeldung an den Sender gibt.



Hierfür benötigt man eine zweite serielle Datenübertragung, d.h. beide Rechner verfügen über eine seriellen Sender und einen seriellen Empfänger. Hierfür ist mindestens eine Verbindung über drei Leitungen (Daten, Rückmeldung, Masse) notwendig. Solche Verbindung werden „Drei – Draht – Verbindungen“ genannt.

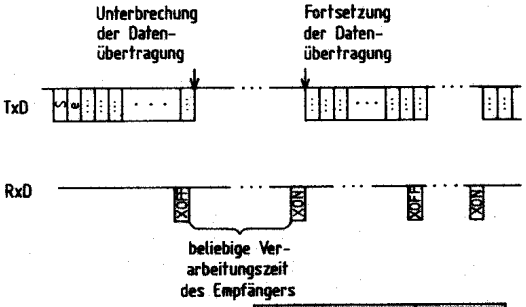


ETX und ACK sind im ASCII-Zeichensatz festgelegt:

ASCII	Hex
ETX	03
ACK	06

Wie funktioniert nun eine solche Datenflusskontrolle?

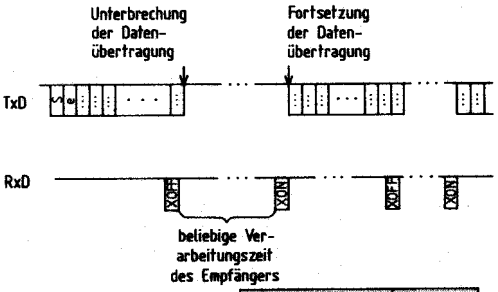
Übertragung von Daten in Blöcken



XON und XOFF sind im ASCII-Zeichensatz festgelegt:

ASCII	Hex
XON = DC1	11
XOFF = DC3	13

Datensender und Datenempfänger treffen die Vereinbarung, dass nach einer festgelegten Anzahl von Zeichen (ein Zeichenblock) die Datenübertragung unterbrochen wird. Die Anzahl der Zeichen ist so gewählt, dass der Empfangsrechner diese problemlos verarbeiten kann. Das Ende dieses Blockes wird mit einem definierten Zeichen markiert. Dieses Zeichen heißt **ETX** (End Of Text). Hat der Empfangsrechner die Daten verarbeitet, quittiert er die dem Senderechner auch wieder über ein definiertes Zeichen. Dieses Zeichen heißt **ACK** (Acknowledge). Anschießend beginnt der Senderechner mit der Übertragung des nächsten Blockes. Die Sende- und Empfangsabwicklung zwischen den beiden Rechnern ist also klar vereinbart. Eine Vereinbarung eines Datenübertragungsablaufs nennt man **Protokoll.** In diesem Fall wird das Protokoll nach den Steuerzeichen **ETX/ACK – Protokoll** genannt.



XON und XOFF sind im ASCII-Zeichensatz festgelegt:

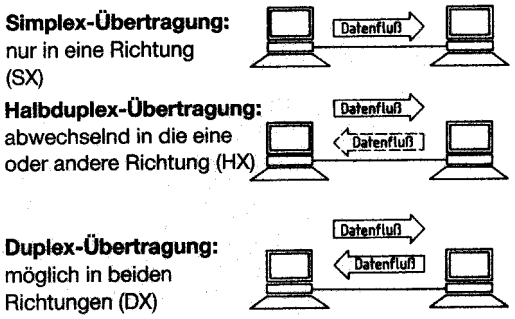
ASCII	Hex
XON = DC1	11
XOFF = DC3	13

Übertragung durch den Empfänger

Bei der Datenübertragung nach dem ETX/ACK – Protokoll wird der Datenfluss nach festgelegten Blöcken und damit nach festgelegten Zeichen unterbrochen. Damit kann die Leistungsfähigkeit eines Systems vermindert werden. Das Datenübertragungsverfahren kann durch das **XON/XOFF** – Protokoll beschleunigt werden. Der Sender überträgt solange Zeichen, bis der Empfänger seine Datenaufnahmegrenze erreicht hat. Dann überträgt der Empfänger ein definiertes Signal **(XOFF)** an den Sender. Nun hat der Empfänger Zeit die gesandten Daten zu verarbeiten. Ist er damit fertig, so überträgt er ein weiteres definiertes Zeichen **(XON)** an den Sender, welcher die Übertragung der Daten wieder aufnimmt.

Übertragungsarten:

Bei der Übertragungsart ohne Datenflusskontrolle werden Daten nur in eine Richtung übermittelt. Dieses Verfahren wird Simplex – Übertragung genannt. Bei der Übertragungsart nach dem ETX/ACK – Protokoll werden Daten zwar in beide Richtungen übermittelt, jedoch geschieht dies abwechselnd. Diese Art der Übertragung wird Halbduplex – Übertragung genannt. Als Duplex - Übertragung wird eine bezeichnet, bei der Daten gleichzeitig in beide Richtungen übermittelt werden. Dies geschieht bei dem XON/XOFF – Protokoll.



Zugriff auf die serielle Schnittstelle mit der Communication – API

Die API erhält man per Download auf http://java.sun.com/products/javacomm/ . Man entpackt die Datei javacomm20-win32.zip in ein Verzeichnis. Anschließend kopiert man die Datei: win32com.dll in das Verzeichnis: c:\jdk15\jre\bin[25], die Datei comm.jar nach c:\jdk15\jre\lib\ext und die Datei javax.comm.properties nach c:\jdk15\jre\lib.

Die nachfolgenden Beispiele sollen den Umgang mit der API erläutern.

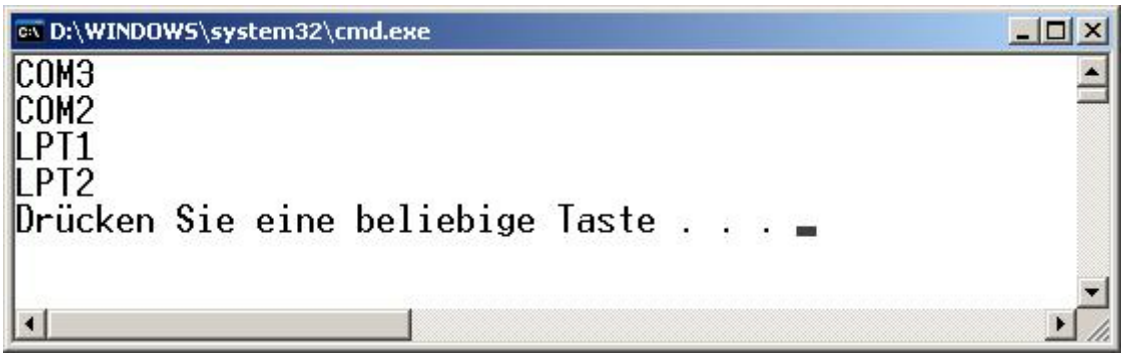
Beispiel: Schnittstellen auflisten

```
import java.util.*;
import java.io.*;
import javax.comm.*;

public class Schnittstellen_abfragen
{
  public static void main(String[] args)
  {
    Enumeration en = CommPortIdentifier.getPortIdentifiers();
    while (en.hasMoreElements())
    {
              CommPortIdentifier cpi = (CommPortIdentifier)en.nextElement();
              System.out.println (cpi.getName());
        }
  }
}
```

Erklärung:

Die statische Methode `CommPortIdentifier.getPortIdentifiers()` liefert eine Liste (Enumeration) von Objekten der Klasse CommPortIdentifier. Mit der Methode getName() liefert jedes dieser Objekte einen Repräsentanten der parallelen und seriellen Schnittstellen als String. Die folgende Bildschirmausgabe wäre möglich:

[25] Ich gehe hier davon aus, dass als JAVA – Installationsverzeichnis jdk15 auf dem Laufwerk c gewählt wurde! Ansonsten ist eine Anpassung notwendig!



Beispiel: Zwei Rechner werden über ein Nullmodemkabel verbunden.

Der Sender sendet über den Port COM2, der Empfänger empfängt die Daten auf dem Port COM1. Es werden Texte übertragen, wobei Sonderzeichen in Strings nicht korrekt übertragen werden.

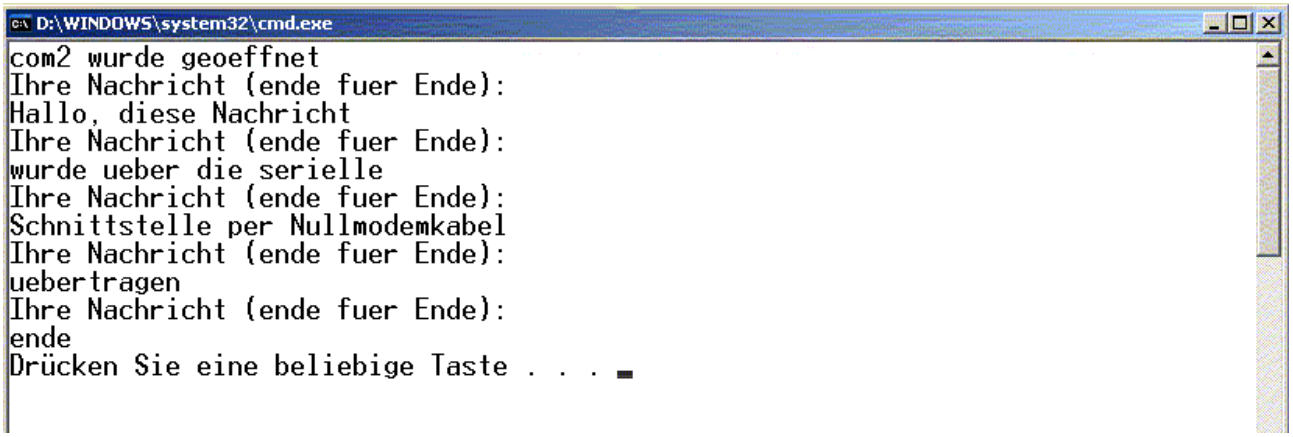
Die Datei NullModemVerbindungSender.java:

```
import java.util.*;
import java.io.*;
import javax.comm.*;

public class NullModemVerbindungSender
{
 public static void main (String args[])
 {
  KonsoleIn eingabe = new KonsoleIn();
  SerialPort com2= null;
  Enumeration en = CommPortIdentifier.getPortIdentifiers();
  while (en.hasMoreElements())
   {
    CommPortIdentifier cpi = (CommPortIdentifier)en.nextElement();
    if (cpi.getName().equals ("COM2"))
    {
     try
      {
       com2 = (SerialPort)cpi.open("COM2",1000);
       System.out.println ("com2 wurde geoeffnet");
com2.setSerialPortParams
(19200,SerialPort.DATABITS_8,SerialPort.STOPBITS_2,SerialPort.PARITY_NONE);
       DataOutputStream ausgabe = new DataOutputStream(com2.getOutputStream());
while (true)
        {
         System.out.println ("Ihre Nachricht (ende fuer Ende):");
         String text = eingabe.readString ();
         ausgabe.writeUTF (text);
         if (text.equalsIgnoreCase ("ende"))
         break;
       }//end of while
       ausgabe.close();
com2.close();
   }catch (Exception e){System.out.println("Fehler"); e.printStackTrace();}
  }//end of if
 }//end of while
}//end of main
}//end of class
```

Erklärung:

Sofern die Methode getName() den Namen "COM2" zurückgibt, wird ein Objekt der Klasse SerielPort über die open(String name, int timeout)- Methode erzeugt. Über die Methode setSerialPortParams () werden die Eigenschaften der Datenübertragung gesetzt. In diesem Beispiel wird die Übertragungsgeschwindigkeit auf 19200 bit/s gesetzt, die Anzahl der Datenbits auf 8 festgelegt, 2 Stopbits vereinbart und kein Paritybit gesetzt. Über die Methode getOutputStream() bekommt man einen Datenstrom geliefert an den man über die entsprechenden write – Methoden Daten senden kann. Der Rest der Klasse sollte mittlerweile bekannt sein!

Ein Programmdurchlauf könnte folgendermaßen aussehen:



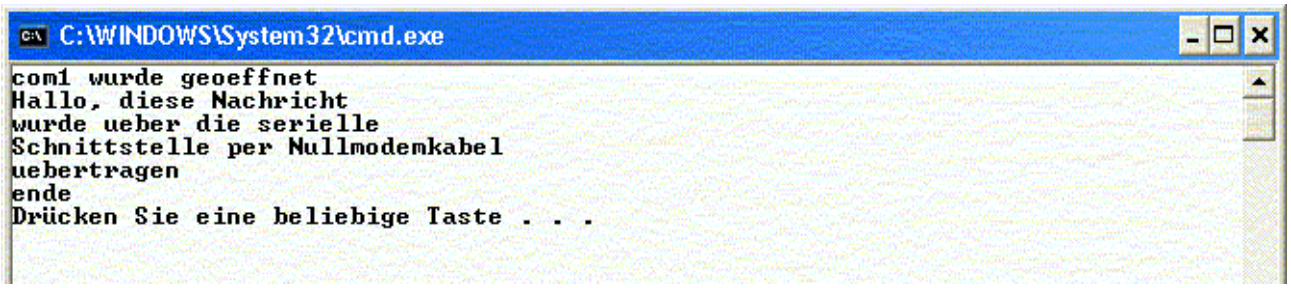
Die Datei NullModemVerbindungSender.java:

```
import java.util.*;
import java.io.*;
import javax.comm.*;

public class NullModemVerbindungEmpfaenger
{
public static void main (String args[])
{
 KonsoleIn eingabe = new KonsoleIn();
 DataInputStream daten =null;
 String nachricht = null;
 SerialPort com2= null;
 Enumeration en = CommPortIdentifier.getPortIdentifiers();
 while (en.hasMoreElements())
  {
  CommPortIdentifier cpi = (CommPortIdentifier)en.nextElement();
   if (cpi.getName().equals ("COM1"))
   {
   try
    {
     com2 = (SerialPort)cpi.open("COM1",1000);
        System.out.println ("com1 wurde geoeffnet");
        com2.setSerialPortParams
(19200,SerialPort.DATABITS_8,SerialPort.STOPBITS_2,SerialPort.PARITY_NONE);
        daten = new DataInputStream(com2.getInputStream());
```

```
        do
        {
                nachricht = daten.readUTF();
                System.out.println (nachricht);
        }while (!nachricht.equalsIgnoreCase("ende"));//end of while
        daten.close();
        com2.close();
        }catch (Exception e){System.out.println("Fehler"); e.printStackTrace();}
     }//end of if
    }//end of while
   }//end of main
}//end of class
```

Erklärung:

Die Datei ist analog zur Senderdatei aufgebaut. Statt OutputStreams werden InputStreams verwendet. Ein Programmdurchlauf könnte wie folgt aussehen:



Nachrichten über das Eventhandling empfangen

Eine Schnittstelle ständig abzufragen ist nichtsonderlich komfortabel, da das Programm immer warten muss, ob Daten anliegen, um diese anschließend zu verarbeiten. Schöner wäre es, das Programm reagiert nur für den Fall, dass Daten an der Schnittstelle anliegen. Dies kann bei Verwendung der Communication – API mit einem EventListener gelöst werden. Dieser stellt eine Schnittstelle dar, die der entsprechenden Klasse beigefügt wird. Im Gegensatz zu den Eventhandling – Interfaces der grafischen Oberfläche, muss bei der seriellen Schnittstelle noch mitgeteilt werden, auf welches Ereignis der Listener aufpassen soll. Im folgenden Beispiel wird dem Objekt SerialPort der SerialPortEventListener hinzugefügt. Die Methode public void serialEvent (SerialPortEvent ev) wird implementiert, im Falle des Dateneingangs jedoch noch nicht aufgerufen. Zuerst muss über die Methode public void notifyOnDataAvailable (boolean wert) der Klasse SerialPort mitgeteilt werden, dass sie auf Dateneingänge Wert auf "erhöhte Aufmerksamkeit" legen soll. In dem folgenden Programmbeispiel wird der einfache Sender von oben verwendet. Der Empfänger liefert nur Konsolenausgaben, sofern Daten an der Schnittstelle anliegen:

Screenshot des Sendeprogramms

Die Klasse StartNullmodemEmpfaengerEvent.java:

```
public class StartNullmodemEmpfaengerEvent
{
	public static void main (String args[])
	{
		NullModemEmpfaengerEvent empfaenger = new NullModemEmpfaengerEvent ();
		for (int i = 0; i < 100000; i++)
		{
			System.out.print(i + "\t");
			try
			{
				Thread.sleep (500);
			}catch (Exception e){}
		}
	}
}
```

Die Klasse legt ein Objekt aus der Klasse NullModemEnpfaengerEvent an und produziert ansonsten nur Zahlen, die auf der Konsole ausgegeben werden!

Der Quellcode der Datei: NullModemEmpfaengerEvent.java

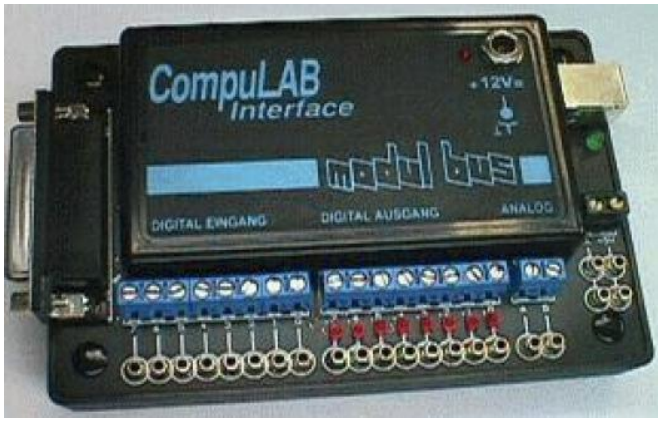
```
import java.util.*;
import java.io.*;
import javax.comm.*;

public class NullModemEmpfaengerEvent implements  SerialPortEventListener
{
 DataInputStream daten =null;
 String nachricht = null;
 SerialPort com1 = null;

public NullModemEmpfaengerEvent()
{
 System.out.println ("Konstruktor");
 Enumeration en = CommPortIdentifier.getPortIdentifiers();
 while (en.hasMoreElements())
  {
   CommPortIdentifier cpi = (CommPortIdentifier)en.nextElement();
```

```
   if (cpi.getName().equals ("COM1"))
   {
    try
     {
      com1 = (SerialPort)cpi.open("COM1",1000);
      System.out.println ("com1 wurde geoeffnet");
      com1.addEventListener(this);
      com1.setSerialPortParams
(19200,SerialPort.DATABITS_8,SerialPort.STOPBITS_2,SerialPort.PARITY_NONE);
      com1.notifyOnDataAvailable(true);
     }catch (Exception e){System.out.println("Fehler"); e.printStackTrace();}
   }//end of if
  }//end of while
}

public void serialEvent(SerialPortEvent ev)
  {
       boolean ende = false;
       System.out.println ("Serial Event eingetreten");
       try
       {
          daten = new DataInputStream(com1.getInputStream());
          do
           {
            nachricht = daten.readUTF();
            if (nachricht.equalsIgnoreCase ("ende"))
            ende = true;
            System.out.println (nachricht);
           }while (daten.available() > 0);//end of while
          daten.close();
          if (ende)
           {
            com1.close ();
            System.exit (0);
           }
       }catch (Exception e){}
  }//end of serialEvent
}//end of class
```

Startet man die Startklasse, so wird (bei Eingabe der Daten entsprechend der Abbildung des Empfängers) die folgende Ausgabe erzeugt:

Man sieht, die Ausgabe wird immer nur dann unterbrochen, sofern Daten an der seriellen Schnittstelle anliegen.

Messen, Steuer und Regeln mit JAVA und dem CompuLAB – Interface



Mit Hilfe dieses Interfaces kann man digital 8 Bits ausgeben, digital 8 Bits einlesen und an zwei analogen Eingängen Daten im Bereich von 0 bis 5 V einlesen. Das Interface ist bei der Firma AK-Modul-Bus Computer GmbH erhältlich (http://www.ak-modul-bus.com). Eine Beschreibung der Ansteuerungsbefehle findet sich unter http://www.elexs.de/sware.htm. Unter Java werden die Eingaben und die Ausgaben zur seriellen Schnittstelle wieder mit dem Communication - API realisiert. Aus der Datei CompuLABdok.pdf kann man die folgenden Funktionen entnehmen:

Daten an die digitalen Ausgänge ausgeben:

Zuerst wird das Byte mit dem Wert 81 an das Interface geschickt. Das nächste gesendete Byte enthält die Daten für die digitalen Ausgänge (die einzelnen Ausgänge haben jeweils die Wertigkeit: 2^x ,wobei x für die Nummer des Ausgangs steht).

Einlesen der digitalen Eingänge:

Gibt's man das Byte mit dem Wert 211 aus, so liefert das nächste eingelesene Byte den Zustand der digitalen Eingänge. Unter JAVA sollte man als Eingabedatenstrom den DataInputStream verwenden und mit der Methode int readUnsignedByte() den Wert einlesen. Ansonsten kommt es zu Datenfehlern (war zumindest bei mir so).

Einlesen der Daten der analogen Eingänge:

Schickt man den Wert 60 an das Interface so liefert das nächste eingelesene Byte den umgerechneten Wert des analogen Eingangs A (Werte von 0 bis 255 für 0 bis 5V).

Schickt man den Wert 58 an das Interface so liefert das nächste eingelesene Byte den umgerechneten Wert des analogen Eingangs B (Werte von 0 bis 255 für 0 bis 5V).

Beispielprogramm zur Ansteuerung des CompuLAP – Interfaces unter JAVA

Im folgenden Beispielprogramm werden zuerst alle Zahlen von 0 bis 255 an die digitalen Ausgänge gesendet. Anschließend werden 100 mal die digitalen Eingänge eingelesen. Zuletzt werden die analogen Eingänge 100 mal eingelesen.

Der Quellcode der Datei Testklasse_Compulab.java

```
import java.util.*;
import java.io.*;
import javax.comm.*;

public class Testklasse_Compulab
{
  public static void main(String[] args)
  {
    SerialPort com2= null;
    Enumeration en = CommPortIdentifier.getPortIdentifiers();
    while (en.hasMoreElements())
    {
CommPortIdentifier cpi = (CommPortIdentifier)en.nextElement();
       if (cpi.getName().equals ("COM2"))
       {
       System.out.println ("Digitale Ausgabe wird getestet");
       try
       {
       com2 = (SerialPort)cpi.open("COM2",1000);
       System.out.println ("com2 wurde geoeffnet");
       com2.setSerialPortParams
(19200,SerialPort.DATABITS_8,SerialPort.STOPBITS_2,SerialPort.PARITY_NONE);
       for (int i = 0; i < 256; i++)
       {
              DataOutputStream ausgabe = new DataOutputStream(com2.getOutputStream());
              ausgabe.writeByte (81);
              ausgabe.writeByte (i);
              ausgabe.close();
              Thread.sleep (80);
       }
       com2.close();
       }catch (Exception e){System.out.println("Fehler"); e.printStackTrace();}

       System.out.println ("Digitale Eingabe wird getestet");
       try
       {
       com2 = (SerialPort)cpi.open("COM2",1000);
       com2.setSerialPortParams
(19200,SerialPort.DATABITS_8,SerialPort.STOPBITS_2,SerialPort.PARITY_NONE);
       int i = 0;
       while (i < 100)
              {
                     OutputStreamWriter ausgabe = new
OutputStreamWriter(com2.getOutputStream());
                     ausgabe.write (211);
                     ausgabe.close();
```

```
                        DataInputStream eingabe = new DataInputStream(com2.getInputStream());
                        int daten = eingabe.readUnsignedByte();
                        System.out.println (daten);
                        eingabe.close();
                        i++;
                }
        com2.close();
        }catch (Exception e){}

        System.out.println ("Analoge Eingabe wird getestet");
        try
        {
        com2 = (SerialPort)cpi.open("COM2",1000);
        com2.setSerialPortParams
(19200,SerialPort.DATABITS_8,SerialPort.STOPBITS_2,SerialPort.PARITY_NONE);
        int i = 0;
        while (i < 100)
                {
                        DataOutputStream ausgabe = new DataOutputStream(com2.getOutputStream());
                        ausgabe.writeByte (60);
                        ausgabe.close();
                        DataInputStream eingabe = new DataInputStream(com2.getInputStream());
                        int datena = eingabe.readUnsignedByte();
                        System.out.print ("Kanal A:" + datena + "\t");
                        eingabe.close();
                        DataOutputStream ausgabeb = new
DataOutputStream(com2.getOutputStream());
                        ausgabeb.writeByte (58);
                        ausgabeb.close();
                        DataInputStream eingabeb = new DataInputStream(com2.getInputStream());
                        int datenb = eingabeb.readUnsignedByte();
                        System.out.println ("Kanal B:" + datenb);
                        eingabeb.close();
                        i++;
                }
        com2.close();
        }catch (Exception e){}
        }//end of if
 }//end of while
}//end of main
} //end of class
```

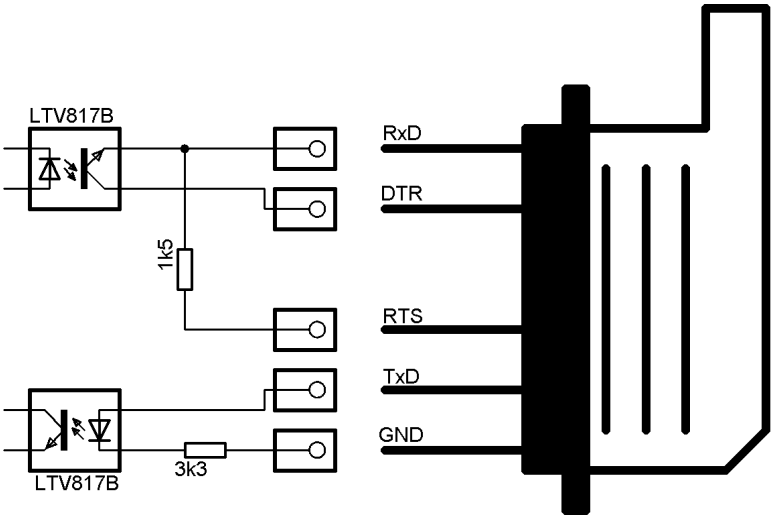
Ansteuern eines Digitalmutimeters

Derzeit befinden sich einige Digitalmultimeter[26] mit serieller Schnittstelle auf dem Markt die prinzipiell alle ähnlich anzusteuern sind. Leider reichen die vorhandenen Dokumentationen gerade für das Erstellen einer Java – Applikation kaum aus. Daher hier einige Hinweise:

Aus den Dokumentationen und dem Internet[27] entnimmt man, dass an das Multimeter der Buchstabe “D” gesendet werden muss. Anschließend bekommt man 7 Bytes zurück geschickt, worin die entsprechenden Messwerte enthalten sind. Nutzt man die Communication API, so muss man nach dem Senden des Buchstaben “D” noch das RTS – Bit auf den Wert 0 gesetzt werden, um dem

[26] Ich habe momentan Zugriff auf die Multimeter METEX M-3850D und McVoice M-345pro.
[27] http://www.b-redemann.de/auslesenDMM.shtml

Multimeter die Empfangsbereitschaft zu signalisieren. Dies kann man auch dem nebenstehenden Bild entnehmen.



Die Datenübertragungsrate liegt bei meinen Messgeräten bei 600 Baud bzw. 1200 Baud. Eventuell müssen auch noch einige Wartezeiten einprogrammiert werden, um dem Messgerät die entsprechende Antwortzeit zu geben. Das nachfolgende Beispielprogramm liest ständig Daten des Messgeräts McVoice M-345pro ein:

```
import java.util.*;
import java.io.*;
import javax.comm.*;

public class Messgeraet
{
  public static void main (String args[])
 {
  SerialPort com = null;
  Enumeration en = CommPortIdentifier.getPortIdentifiers();
  while (en.hasMoreElements())
  {
   CommPortIdentifier cpi = (CommPortIdentifier)en.nextElement();
   if (cpi.getName().equals ("COM2"))
   {
        try
        {
                com = (SerialPort)cpi.open("COM2",1000);
                System.out.println ("Com2 wurde geoeffnet");
                com.setSerialPortParams
(600,SerialPort.DATABITS_7,SerialPort.STOPBITS_2,SerialPort.PARITY_NONE);
                com.setFlowControlMode (SerialPort.FLOWCONTROL_NONE);
                OutputStream ausgabe = com.getOutputStream();
                DataInputStream daten = new DataInputStream(com.getInputStream());
                while (true)
                {
                        ausgabe.write(0x44);
                        com.setRTS(false);
                        while (daten.available()!=0)
                        {
                                Thread.sleep (500);
                                byte eingang  = daten.readByte();
                                System.out.print ((char)eingang+" ");
                        }
```

```
                    System.out.println ();
                }
            }
            catch (Exception e){System.out.println("Fehler"); e.printStackTrace();}
    }
   }
  }//end of main
}
```

Anmerkung: Für das METEX M-3850D (Voltcraft M-3850D) muss die Übertragungsgeschwindigkeit auf 1200 Baud herauf gesetzt werden, ansonsten funktioniert das Programm einwandfrei.

Aufgabensammlung

Aufgaben zu Variablen / Rechnen mit Variablen

Aufgabe 1 zu Variablen / Rechnen mit Variablen:

Bei einer Klassenarbeit wurde der folgende Schnitt erzielt:

Note	1	2	3	4	5	6
Anzahl	2	4	6	5	2	1

Erstellen Sie ein JAVA-Programm, welches den Klassenspiegel formatiert auf der Konsole ausgibt.

Zusätzlich soll unter dem Klassenspiegel noch der Notendurchschnitt ausgegeben werden.

Aufgabe 2 zu Variablen / Rechnen mit Variablen:

Von einer linearen Funktion sind die Punkte P (3|4) und Q (5|2) bekannt. Schreiben Sie ein JAVA-Programm LineareFunktion1.java, welches die Funktionsgleichung in der Form y = mx + b sowie die Nullstelle der Funktion ausgibt.

Aufgabe 3 zu Variablen / Rechnen mit Variablen:

Von einem geometrischen Körper ist der Radius r = 4 und die Höhe h = 7 bekannt.

Berechnen Sie per JAVA-Programm Geometrie1.java Volumen und Oberfläche des Körpers wenn dieser ein

a) Kegel
b) Zylinder

ist.

Aufgabe 4 zu Variablen / Rechnen mit Variablen:

Eine Firma verkauft PCs zum Preis von 800€. Die Firma kauft die PCs für 600€ ein. Schreiben Sie ein JAVA-Programm Firma1.java, welches folgende Ausgaben beim Verkauf von 30 PCs tätigt: der Umsatz,

der Gewinn, die Mehrwertsteuer von einem PC (19%) und die abzuführende Mehrwertsteuer beim Verkauf aller PCs.

Aufgaben zur Eingabe von der Konsole

Aufgabe 1 zur Eingabe von der Konsole:

Erstellen Sie ein Programm `Klassenarbeit1.java` in dem ein Lehrer die Noten einer Klassenarbeit eingeben kann. Das Programm soll anschließend den Klassenspiegel formatiert auf der Konsole ausgeben. Weiterhin wird der berechnete Schnitt auf der Konsole ausgegeben.

Aufgabe 2 zur Eingabe von der Konsole:

Erstellen Sie ein Programm LineareFunktion1.java. In dem Programm gibt der Anwender zwei Punkte einer linearen Funktion ein. Das Programm gibt anschließend die Funktionsgleichung in der Form y = mx + b aus. Weiterhin wird eine mögliche Nullstelle der Funktion ausgegeben. erstellen Sie von diesem Programm auch den Programmablaufplan sowie das Struktogramm.

Aufgaben zur bedingten Entscheidung

Aufgabe 1 zur bedingten Entscheidung

Erstellen Sie ein Programm `Klassenarbeit2.java` in dem ein Lehrer die Noten einer Klassenarbeit eingeben kann. Das Programm soll anschließend den Klassenspiegel formatiert auf der Konsole ausgeben. Ist der Notenschnitt schlechter als 4,5, so soll „`Die Klassenarbeit muss wiederholt werden.`“. Ansonsten wird „`Die Klassenarbeit kann gewertet werden.`“ auf der Konsole ausgegeben.

Aufgabe 2 zur bedingten Entscheidung

Ein Programm `Zahlenvergleich.java` soll erstellt werden. In dem Programm gibt der Anwender drei Zahlen ein. Das Programm ermittelt die größte der drei Zahlen und gibt sie aus. Erstellen Sie von dem Programm den Quelltext, den Programmablaufplan und das Struktogramm.

Aufgabe 3 zur bedingten Entscheidung

Erstellen Sie ein Programm `Dreieck1.java`. In dem Programm soll der Benutzer entweder die drei Seitenlängen eines Dreiecks, oder aber die Seitenlänge und zwei angrenzende Winkel eingeben. Ist das Programm rechtwinklig, so wird „`Das Dreieck ist rechtwinklig.`“ auf dem Bildschirm ausgegeben. Außerdem berechnet das Programm alle fehlenden Winkel und alle fehlenden Seitenlängen des Dreiecks und gibt diese aus. Erstellen Sie Quelltext sowie Struktogramm.

Hinweis:

Sie finden mathematische Funktionen in der Bibliothek Math der JAVA-Dokumentation.

Aufgabe 4 zur bedingten Entscheidung

Erstellen Sie ein Programm `SchnickSchnackSchnuck.java`. Das Programm soll einen Durchlauf dieses bekannten Spiels realisieren. Der Spieler spielt gegen den Computer, welcher per Zufallsgenerator Stein, Schere, Papier oder Brunnen auswählt.

Hinweis:

Über die Funktion `Math.random()` erzeugt das Programm eine Zufallszahl im Bereich von 0 bis 1.

Aufgaben zur Mehrfachentscheidung

Ein kleiner Taschenrechner soll programmiert werden. Dieser hat folgende Funktionen:

- zwei Zahlen addieren (1)
- zwei Zahlen subtrahieren (2)
- zwei Zahlen multiplizieren (3)
- zwei Zahlen dividieren (4).

Der Taschenrechner soll Menügesteuert ablaufen. Der Anwender bekommt diese vier Funktionen zur Auswahl gestellt. Gibt er die jeweilige Zahl ein, so kann er die Funktion verwenden. Gibt er im Menü die Ziffer 0 ein, so wird das Programm beendet. In allen anderen Fällen bekommt der Anwender einen Hinweis auf die Falscheingabe und das Menü wird erneut angezeigt.

Erstellen Sie von dieser Software ein Struktogramm. Das Programm ist strukturiert zu gestalten! Kodieren Sie den kompletten Taschenrechner in der Programmiersprache JAVA. Kommentieren Sie Ihr Programm ausführlich.

Aufgaben zur fußgesteuerten Schleife

Aufgabe 1 zur fußgesteuerten Schleife

Schreiben Sie das Programm, welches den Durchschnitt einer Klassenarbeit berechnet so um, dass der Anwender am Ende des Programms per Eingabe entscheiden kann, ob das Programm nochmal ablaufen soll (Dateiname: `Klassenarbeit3.java`).

Aufgabe 2 zur fußgesteuerten Schleife

Erstelle ein Programm, welches nur Zahlen ausgibt, die sich restlos durch 4 teilen lassen. 120 soll als letzte Zahl ausgegeben werden. Wie viele Male wird die integrierte Schleife durchlaufen?

Aufgabe 3 zur fußgesteuerten Schleife

Ein Zahlenratespiel soll erstellt werden. Der PC ermittelt eine Zufallszahl von 1 bis 100. Anschließend errät der Anwender seine erste Zahl. Ist die Zahl zu groß bekommt er die Meldung: „Zahl zu groß“

ausgegeben. Ist die Zahl zu klein, bekommt er die Meldung: „Zahl zu klein“ ausgegeben. In diesen beiden Fällen kann der Anwender erneut eine Zahl raten.

Hat er die richtige Zahl erraten, so bekommt er die Meldung „Bingo gewonnen, Sie haben x Versuche benötigt“ (x steht hierbei für die tatsächliche Anzahl von Versuchen). Der Anwender wird dann gefragt, ob er noch einmal spielen will und kann dies per Eingabe entscheiden.

- Erstelle Sie zu diesem Programm den Programmablaufplan.
- Erstellen Sie zu diesem Programm das Struktogramm.
- Implementieren Sie den Quellcode in der Datei `Zahlenratespiel1.java`.

Aufgabe 4 zur fußgesteuerten Schleife

Erstellen Sie ein Programm `QuadratischeFunktionen1.java`. Der Anwender soll die Koeffizienten a,b und c einer quadratischen Funktion eingeben. Anschließend soll der PC die Nullstelle der Funktion auf dem Bildschirm ausgeben. Hat die Funktion eine oder aber keine Nullstelle, so ist dies zu berücksichtigen. Am Ende wird der Anwender gefragt, ob er das Programm wiederholen möchte und kann dies per Eingabe auswählen.

Aufgaben zur Zählschleife

Aufgabe 1 zur Zählschleife

Erstellen Sie ein Programm EinMalEins1.java mit folgender Funktionalität:

Der Benutzer gibt einen Wert von 1 bis 10 ein (andere Werte sind nicht zugelassen). Das Programm gibt auf dem Bildschirm anschließend das entsprechende 1x1 aus. Beispiel:

Eingabe: 5

1 x 5 = 5

2 x 5 = 10

…

…

…

10 x 5 = 50

Aufgabe 2 zur Zählschleife

Erstelle ein Programm EinMalEins2.java mit folgender Funktionalität:

Das Programm gibt alle 10er-1x1 formatiert auf den Bildschirm aus:

1 2 3 4 5 6 7 8 9 10

2 4 6 8 ……

3 ..

4 ..

5 ..

6

7

8

9

10 20

Aufgabe 3 zur Zählschleife

Erstelle ein Programm Quadrat1.java welches folgende Funktionalität hat:

Der Benutzer gibt eine Zahl von 1 bis 10 ein (andere Werte sind nicht zugelassen). Das Programm zeichnet anschließend ein Quadrat mit dem Zeichen „*“. Beispiel:

Eingabe: 4

****

****

****

****

Aufgabe 4 zur Zählschleife

Erstellen Sie ein Programm Dreieck1.java mit folgender Funktionalität:

Der Benutzer gibt eine Zahl von 1 bis 10 ein (andere Werte sind nicht zugelassen). Das Programm zeichnet anschließend ein rechtwinkliges Dreieck mit dem Zeichen „*“. Beispiel:

Eingabe: 5

*

**

***

****

*****

Aufgaben zu break / continue

Aufgabe 1 zu break / continue

Erstellen Sie ein Programm Zugang1.java. Der Benutzer hat vier Versuche einen vorgegebenen vierstelligen Pin einzugeben. Gibt er den richtigen Pin ein, so erscheint die Ausgabe: „Zugang gewährt“. Nach vier Fehlversuchen erscheint die Ausgabe „Der Wachschutz kommt gleich“.

Aufgabe 2 zu break / continue

Das Programm EinMalEins2.java soll in das Programm EinMalEins3.java abgewandelt werden. Der Benutzer gibt eine Zahl von 1 bis 10 ein, deren 1x1 nicht ausgegeben wird!

Aufgabe 3 zu break / continue

Das bekannte Zahlenratespiel soll so abgewandelt werden, dass der Anwender maximal sechsmal eine Zahl raten darf.

Aufgaben zu Arrays

Aufgabe 1 zu Arrays:

Ein Speicherthermometer misst an einem Tag zu jeder Stunde die Temperatur. Es kommt folgende Tabelle zu Stande:

Uhrzeit	1.00 Uhr	2.00 Uhr	3.00 Uhr	4.00 Uhr	5.00 Uhr	6.00 Uhr	7.00 Uhr	8.00 Uhr
Grad	8°	8°	9°	9°	10°	13°	14°	16°
Uhrzeit	9.00 Uhr	10.00 Uhr	11.00 Uhr	12.00 Uhr	13.00 Uhr	14.00 Uhr	15.00 Uhr	16.00 Uhr
Grad	18°	19°	20°	23°	25°	26°	25°	25°
Uhrzeit	17.00 Uhr	18.00 Uhr	19.00 Uhr	20.00 Uhr	21.00 Uhr	22.00 Uhr	23.00 Uhr	24.00 Uhr
Grad	24	23	21	19	17	14	11	9

Erstellen Sie ein JAVA-Programm mit folgender Funktionalität:

- Ermittlung der Durchschnittstemperatur.
- Ermittlung des kältesten Wertes.
- Ermittlung des wärmsten Wertes.
- Ermittlung der Uhrzeit mit der stärksten Temperaturschwankung zur nächsten Stunde.

Erstellen Sie das Programm so, dass es mit festen Werten aber auch mit Werten von der Konsole gefüttert werden kann. Verwenden Sie möglichst wiederverwendbare Funktionen.

Aufgabe 2 zu Arrays:

Das Wort Rentner ist ein Palindrom, es kann sowohl vorwärts wie rückwärts gelesen werden.

Erstellen Sie ein JAVA-Programm mit folgender Funktionalität:

Der Anwender gibt ein Wort ein. Das Programm prüft, ob das eingegebene Wort ein Palindrom ist und gibt das Ergebnis aus.

Hinweis:

In der JAVA-Doku kann man der Klasse String einige Methoden entnehmen, unter anderem, eine Methode, um einen String in ein char-Array umzuwandeln.

Aufgabe 3 zu Arrays:

Das Programm aus Aufgabe 2 ist so zu verändern, dass ganze Sätze eingegeben werden und überprüft werden können. Erstellen Sie den Quellcode und das Struktogramm dieses Programms.

Aufgabe 4 zu Arrays:

Ein Programm fragt den Anwender nach sechs zu wählenden Lottozahlen (6 aus 49). Natürlich darf keine Zahl doppelt eingegeben werden.

Anschließend ermittelt das Programm 1000 Ziehungen. Eine Ziehung besteht aus sechs Zahlen (6 aus 49) wobei natürlich auch hier keine Zahl doppelt vorkommen darf.

Dann untersucht das Programm die Eingabe des Benutzers und die Ziehungen. Es gibt aus, wie oft der Benutzer wie viele Richtige gehabt hätte.

Aufgaben zu Sortieralgorithmen

Aufgabe 1 zu Sortieralgorithmen:

Schreiben Sie ein Programm BubbleSort.java, welches ein Feld mit 40 Zufallszahlen im Bereich 0 bis 100 füllt und anschließend auf dem Bildschirm ausgibt. Nun soll das Feld mit dem Bubble-Sort sortiert werden. Nach jedem Durchlauf ist das Feld auszugeben. Geben Sie auch abschließend die Anzahl der Durchläufe aus. Führen Sie das Programm mehrfach aus und beobachten Sie die Anzahl der Durchläufe.

Erstellen Sie von dem Programm ein normgerechtes Struktogramm.

Aufgabe 2 zu Sortieralgorithmen:

Schreiben Sie ein Programm Insertion.java, welches ein Feld mit 40 Zufallszahlen im Bereich 0 bis 100 füllt und anschließend auf dem Bildschirm ausgibt. Nun soll das Feld mit dem Insertion-Sort sortiert werden. Nach jedem Durchlauf ist das Feld auszugeben. Geben Sie auch abschließend die Anzahl der Durchläufe aus. Führen Sie das Programm mehrfach aus und beobachten Sie die Anzahl der Durchläufe. Programmieren Sie so, dass beide beschriebenen Varianten (Ein oder zwei Arrays) möglich sind.

Erstellen Sie von dem Programm ein normgerechtes Struktogramm.

Aufgabe 3 zu Sortieralgorithmen:

Schreiben Sie ein Programm Selection.java, welches ein Feld mit 40 Zufallszahlen im Bereich 0 bis 100 füllt und anschließend auf dem Bildschirm ausgibt. Nun soll das Feld mit dem Selection-Sort sortiert werden. Nach jedem Durchlauf ist das Feld auszugeben. Geben Sie auch abschließend die Anzahl der Durchläufe aus. Führen Sie das Programm mehrfach aus und beobachten Sie die Anzahl der Durchläufe.

Erstellen Sie von dem Programm ein normgerechtes Struktogramm.

Aufgabe 4 zu Sortieralgorithmen:

Schreiben Sie ein Programm Quicksort.java, welches ein Feld mit 40 Zufallszahlen im Bereich 0 bis 100 füllt und anschließend auf dem Bildschirm ausgibt. Nun soll das Feld mit dem Quick-Sort sortiert werden. Nach jedem Durchlauf ist das Feld auszugeben. Geben Sie auch abschließend die Anzahl der Durchläufe aus. Führen Sie das Programm mehrfach aus und beobachten Sie die Anzahl der Durchläufe.

Erstellen Sie von dem Programm ein normgerechtes Struktogramm.

Hinweis:

Der Algorithmus ist nicht ganz so leicht zu implementieren, Sie können daher das Internet als Hilfsmittel verwenden.

Aufgabe 5 zu Sortieralgorithmen:

Gegeben ist eine Klasse Konto mit Kontonummer, Betrag und Name und Vorname des Inhabers. Mehrere Konten werden in der Klasse Bank von einer verwaltet.

Erstellen Sie die Klasse `Konto`.

Implementieren Sie die Klasse `Bank`.

Erstellen Sie eine Applikation in der 10 Konten erzeugt werden. Die Bank bekommt vier Methoden:

`public ArrayList<Konto> sortiereNummer()`: Die Methode gibt eine nach Kontonummern sortierte ArrayList zurück. Zu verwenden ist der Bubble-Sort.

`public ArrayList<Konto> sortiereBetrag()`: Die Methode gibt eine nach Beträgen sortierte ArrayList zurück. Zu verwenden ist der Insertion-Sort.

`public ArrayList<Konto> sortiereNameVorname()`: Die Methode gibt eine nach Nachnamen und Vornamen sortierte ArrayList zurück. Zu verwenden ist der Selection-Sort.

`public void ausgeben(ArrayList<Konto>)`: Die Methode gibt die Informationen der Konten in der übergebenen ArrayList auf dem Bildschirm aus.

Aufgaben zu Klassen / Objekten

Aufgabe 1 zu Klassen / Objekten

Ein Haustier hat eine Rasse und einen Namen. Erstellen Sie das Klassendiagramm sowie den JAVA-Quelltext der Klasse Haustier.java.

Aufgabe 2 zu Klassen / Objekten

Ein Produkt hat einen Produktnamen, einen Nettopreis sowie das Merkmal food oder nonfood.

Erstellen Sie das Klassendiagramm sowie den JAVA-Quelltext der Klasse Produkt.java.

Aufgabe 3 zu Klassen / Objekten

Ein Handy hat diverse Eigenschaften. Sammeln Sie diese Eigenschaften indem Sie ein Klassendiagramm der Klasse Handy erstellen.

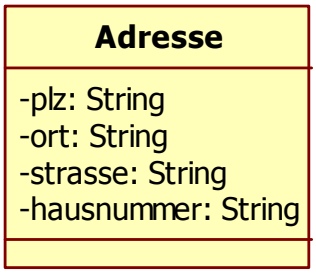
Aufgabe 4 zu Klassen / Objekten

Analysieren Sie den folgenden Quellcode. Vervollständigen Sie diesen und erstellen Sie hieraus das passende Klassendiagramm:

```
public class Buch
{
	private String titel;
	private String kategorie;
	private boolean ausleihStatus;
}
```

Aufgabe 5 zu Klassen / Objekten

Analysieren und vervollständigen Sie das folgende Klassendiagramm. Erstellen Sie den kompletten Quellcode.



Aufgabe 6 zu Klassen / Objekten

Situationsbeschreibung:

Ein Konto hat eine Kontonummer und einen Kontostand. Es ist möglich, einen Betrag auf das Konto einzuzahlen oder abzuheben. Das Konto darf beim Abheben nicht überzogen werden. Ein Kunde hat einen Namen und einen Vornamen. Weiterhin besitzt der Kunde ein Konto. Außerdem hat er eine Adresse (Strasse, Nr, Plz, Ort).

a) Erstellen Sie das Klassendiagramm der Situation.
b) Erklären Sie anhand Ihres Diagramms die Begriffe: Klasse, Attribut, Methode, Sichtbarkeit, Assoziation, Rolle, Multiplizität.
c) Implementieren Sie die Quellcodes Ihrer Klassen und testen Sie diese in einer Klasse Bank.java.

Aufgabe 7 zu Klassen / Objekten

Situationsbeschreibung:

In einem Fitnessstudio gibt es Mitglieder. Diese besitzen neben Name, Vorname und Adresse einen Vertrag. Der Vertrag hat eine Zahlungsmethode, einen Vertragsbeginn sowie eine Laufzeit.

Erstellen Sie das Klassendiagramm dieser Situation.

Aufgabe 8 zu Klassen / Objekten

Situationsbeschreibung:

Eine Bücherei soll objektorientiert modelliert werden. Ein Verlag hat eine Nummer, einen Namen und eine Adresse. Ein Buch besitzt eine ISBN-Nummer, einen Titel und einen Autor. Ein Buch wird von einem Verlag verlegt. Von einem Buch gibt es in der Bücherei Exemplare. Die Exemplare besitzen eine eindeutige ID, sowie ein Anschaffungsdatum. Es werden Leser verwaltet. Ein Leser besitzt einen Namen, einen Vornamen, eine Mailadresse, eine Handynummer sowie eine Adresse (Straße, Nr, Plz, Ort). Die Leser können ein Buch ausleihen. Gespeichert werden pro Ausleihe das Verleihdatum, das Rückgabedatum sowie die geplante Verleihdauer (in Tagen).

Aufgabenstellungen:

a) Erstellen Sie ein normgerechtes Klassendiagramm der Situationsbeschreibung.
b) Erklären Sie an Ihrem Klassendiagramm die Begriffe Klasse, Attribut, Methode, Sichtbarkeit, Assoziation, Multiplizität und Rolle.

c) Implementieren Sie alle Quellcodes und testen Sie diese in einer Klasse BuechereiMain.java.

Für Fortgeschrittene:

a) Aus der ISBN-Nummer kann man über einen XML-Request die Daten des Buches aus dem Internet abrufen. Erstellen Sie einen solchen Konstruktor der Klasse Buch.

Aufgaben zum Schlüsselwort static

Aufgabe 1 zum Schlüsselwort static

Situationsbeschreibung:

Ein Konto beinhaltet eine Kontonummer und einen Kontostand. Die Klasse soll so angelegt werden, dass eine Anwendung immer weiß, wie viele Konten gerade existieren. Außerdem soll die Kontonummer immer automatisch generiert werden und zwar ab der Nummer 1000. Dies bedeutet, dass das erste Konto die Nummer 1001 hat, das zweite Konto die Nummer 1002 usw.

Aufgabenstellung:

Erstellen Sie die Klasse Konto.

Erstellen Sie eine Anwendung in der 4 Konten erzeugt werden.

Geben Sie am Ende die Anzahl der vorhandenen Konten, und anschließend die Informationen aller Konten aus.

Aufgaben zur Vererbung, Überschreiben von Methoden, Polymorphismus, Late Binding

Aufgabe 1 zur Vererbung

Es gibt Fahrzeuge, Landfahrzeuge, Wasserfahrzeuge, Segelboote, Motorboote, Transportfahrzeuge und Personenfahrzeuge. Bringen Sie die Klassen in eine Vererbungshierarchie. Denken Sie sich je zwei Attribute pro Klasse aus. Jede Klasse bekommt einen geeigneten Konstruktor und eine toString-Methode. Erstellen Sie das Klassendiagramm und den Quellcode. Testen Sie Ihren Quellcode in einer geeigneten Main.

Aufgabe 2 zur Vererbung

Eine Schule hat Schüler, Lehrer und Klassen. Klassensprecher sind Schüler. Abteilungsleiter sind Lehrer. Klassen haben einen Klassenlehrer und einen Klassensprecher. Stellen Sie alle Klassen in einem Klassendiagramm dar. Benennen Sie pro Klasse je zwei Attribute und Implementieren Sie die Quellcodes. Testen Sie Ihre Quellcodes in einer Main.

Aufgaben zum Datum / Joda-Time

Aufgabe 1 zum Datum / Joda-Time

Erstellen Sie ein Programm bei dem der Anwender ein beliebiges Datum eingeben kann. Das Programm berechnet dann die Zeitspanne von heute bis zum eingegebenen Datum in Jahren, Monaten und Tagen und gibt diese auf dem Bildschirm aus.

Aufgabe 2 zum Datum / Joda-Time

Die folgenden beweglichen Feiertage sind am Datum von Ostern orientiert:

- Aschermittwoch ist 46 Tage vor Ostern.
- Pfingsten ist 49 Tage nach Ostern.
- Chr. Himmelfahrt ist 10 Tage vor Pfingsten.
- Fronleichnam ist 11 Tage nach Pfingsten.

Oster berechnet sich nach der „Gaußschen Osterformel“ wie folgt[28]:

Ostern fällt im Jahre J auf den (D + e + 1)sten Tag nach dem 21. März:
a = Rest von J / 19
b = Rest von J / 4
c = Rest von J / 7
*m = ganze Zahl von (8 * ganze Zahl von (J / 100) + 13) / 25 - 2*
s = ganze Zahl von (J / 100) - ganze Zahl von (J / 400) - 2
M = Rest von (15 + s - m) / 30
N = Rest von (6 + s) / 7
*d = Rest von (M + 19 * a) / 30*
D = 28 falls d = 29 oder
D = 27 falls d = 28 und a größer/gleich 11 oder
D = d für alle anderen Fälle
*e = Rest von (2 * b + 4 * c + 6 * D + N) / 7*

Ostern = 21. März + (D + e + 1)

Erstellen Sie ein Struktogramm, welches die „Gaußsche Osterformel“ darstellt.

Erstellen Sie ein JAVA-Programm, welches nach der Eingabe eines Jahres im Bereich von 1583 bis 8202 die Daten von Ostern, Faschingssamstag, Aschermittwoch, Pfingsten, Christi Himmelfahrt und Fronleichnam ausgibt.

Aufgabe 3 zum Datum / Joda-Time

Ein Geburtstagserinnerungsprogramm soll erstellt werden. Der Benutzer kann in einer graphischen Benutzeroberfläche den Namen, Vornamen und das Geburtsdatum einer Person eingeben (Klick auf einen Button „Geburtstag speichern“). Diese Daten werden in einer Textdatei so gespeichert, dass durch Kommata separiert in einer Zeile jeweils Name, Vorname und Geburtsdatum der jeweiligen Person stehen.

Durch Klick auf einen Button „Geburtstage anzeigen“ wird in einer Textarea angezeigt, in wie vielen Tagen die jeweilig gespeicherten Personen Geburtstag haben und wie alt sie werden. Sollte eine Person innerhalb der aktuell nächsten Woche Geburtstag haben, so soll dies in einem Dialog angezeigt werden.

[28] aus: http://www.igelnet.de/Dateien/xlostern.htm