Java für Java-Hasser

Michael Dienert *

6. Oktober 2011

Inhaltsverzeichnis

*Satz: unvermeidlich - LaTeX

Ist mir alles zu theoretisch hier ...

Diese kleine Anleitung ist für Leute gedacht, die mit wenig Programmier-Vorkenntnissen ein paar erste Schritte mit Java wagen möchten, ohne von der Objektorientierungs-Theorie erschlagen zu werden. Statt viel Theorie stehen deshalb im Text möglichst nur Beispiele und Anleitungen.

Da das gesamte Java-Paket aber ein *riesen* Ding ist, für dessen Erforschung man Monate wenn nicht Jahre braucht, kann diese Anleitung wirklich nur eine winzig kleine Starthilfe in die Java- Programmierung sein. Man kann mit ihr aber problemlos lernen, kleine Java-Programme zu schreiben, mit denen man z.B. eine Textdatei konvertieren oder eine kleine Datenbankanwendung schreiben kann.

Der Text endet mit einer kurzen Einführung in die Begriffe *Objekt* und *Klasse*. Die weiteren Feinheiten der objektorientierten Programmierung wie Subklassen, Vererbung

und Mehrfachverwendung gleicher Namen werden an dieser Stelle *nicht* erklärt.

> Und natürlich ist der Text nur zur Begleitung des Unterrichts des Autors gedacht. Wer Java bei jemand anderem lernt, darf ihn nicht lesen ...

1 Ein paar Grundbegriffe und Regeln

1.1 Was ist ein Programm

Ein Programm ist einfach eine Folge von *eindeutigen* Befehlen. In den allermeisten Programmiersprachen sind diese Befehle ein winzig kleiner Ausschnitt der englischen Sprache. Dieser Ausschnitt ist allerdings so gewählt, dass die damit formulierten Befehle eben *absolut eindeutig* sind.

Der Erfinder von Java (James Gosling) hat für Java zum grössten Teil die Befehle der Programmiersprache C übernommen. Wer also schon C kann, hat es mit Java etwas leichter.

1.2 Schlüsselwörter

Schlüsselwörter sind alle Wörter, die zum Sprachumfang von Java gehören, eben der oben erwähnte Ausschnitt der englischen Sprache.

1.3 Anweisungen

Damit der Computer die Befehlsfolge Schritt für Schritt abarbeiten kann, müssen die einzelnen Befehle eindeutig voneinander getrennt werden.

> Nach jedem Befehl eines Java-Programms *muss* ein Semikolon stehen. Statt des Begriffs *Befehl* wird meistens der Begriff *Anweisung* verwendet.

Hier ein Beispiel:

```
a = a+1;
b = b-1;
if (a > b) System.out.println("Max=" + a);
else System.out.println("Min=" + b);
```

Abbildung 1: Einige Anweisungen in Folge

1.4 Blöcke

Oft kommt es vor, dass mehrere Anweisungen zusammengefasst werden sollen. Das macht man, indem man Anweisungen zu *Blöcken* zusammenfasst:

> Ein Block beginnt und endet mit einer geschweiften Klammer: $\{ \cdots \}$. Zwischen diesen Klammern stehen eine oder mehrere Anweisungen.

Hier ein Beispiel:

```
if (a > b) {
   c = a;
   a = b;
   b = c;
}
```

Abbildung 2: Ein Block aus drei Anweisungen

1.5 Einrücken von Blöcken

Damit die Blöcke im Programmtext gut erkennbar sind, müssen sie *eingerückt* werden. Ein guter Programmier-Editor erledigt dieses Einrücken automatisch und selbstverständlich kann die Java-Programmierumgebung NetBeans das auch (und noch viel mehr).

Ich empfehle dringend, die Klammern dabei wie im obigen Beispiel gezeigt, anzuordnen.

Eine andere Möglichkeit sieht z. B. so aus:

```
if (a > b)
{
   c = a;
   a = b;
   b = c;
}
```

Abbildung 3:

Bei dieser Methode des Klammersetzens wird der Programmtext um eine Zeile länger ohne dass der eigentliche Block besser hervorgehoben wird. Das mag an dieser Stelle nicht weiter schlimm sein, aber bei längeren Programmen ist man froh um jede zusätzliche Zeile, die man ohne zu scrollen mit einem Blick auf dem Bildschirm sehen kann.

Einrücken von Blöcken:
Das richtige Einrücken von Blöcken ist absolut wichtig für die Lesbarkeit eines Programms.

Blöcke müssen unbedingt eingerückt werden !!

Im Anhang ist ein Beispielprogramm, bei dem mehrere Blöcke ineinandergeschachtelt sind (Abb. 35). Der Quelltext währe ohne Einrückungen absolut unlesbar.

2 Aufbau eines Java-Programms

Hier das Beispiel eines sehr einfachen Java-Programms:

```
class Hello{
  public static void main(String[] args){
    System.out.println("Hello world!");
  }
}
```

Abbildung 4: Das unvermeidliche Hello-world-Programm

2.1 Klassenname

Das Programm beginnt mit dem Schlüsselwort `class`, das von einem beliebigen Namen und einem **Block** gefolgt wird. Dabei ist es absolut wichtig, dass der Name der nach `class` steht mit einem *Grossbuchstaben* beginnt! Diese Name wird im Weiteren als *Klassenname* bezeichnet.

Klassenname:

- **Der Klassenname *muss* grossgeschrieben werden!**
- **Klassennamen sind die *einzigen* Namen in Java, die mit einem einzelnen Grossbuchstaben anfangen.**

Das gilt für jedes Java-Programm! Die Struktur eines einfachen Java-Programms sieht zunächst einmal so aus:

```
class Name{
  anweisung;
  anweisung;
     ...
}
```

Abbildung 5: Struktur eines jeden Java-Programms

2.2 Klasse

Genauer gesagt, ist das Beispiel aus Abb. 5 eine *Klasse*. Ein Java-*Programm* dagegen kann aus **einer oder mehreren Klassen** bestehen. Zunächst beschränken wir uns aber auf Programme, die nur aus einer Klasse bestehen.

Nochmal zur Wiederholung: eine Klasse schreibt man, indem man mit dem Schlüselwort `class` beginnt, dann kommt der Klassenname (grossgeschrieben) und dann ein Block (der wird mit geschweiften Klammern eingefasst).

2.3 Die `main`-Methode

Wer schon mal in C programmiert hat, kennt bereits die `main`-Funktion. Die `main`-*Methode* ist bis auf winzige Unterschiede identisch. *Methode* ist nur ein anderes Wort

für *Funktion*. Für ein Java-Programm muss nun folgendes gelten:

> In einem Java-Programm *muss* es **mindestens** eine Klasse geben, die die main-Methode enthält.
> Diese Klasse wird ab jetzt als *Startklasse* bezeichnet.

Besteht das Programm nur aus einer Klasse, muss eben diese Klasse die main-Methode enthalten.

Abb. 31 zeigt die main-Methode. Was die Schlüsselwörter `public`, `static`, `void`

```
  public static void main(String[] kzp){
    anweisung;
        ...
  }
```

Abbildung 6: Die `main`-Methode

und der Name `main` bedeuten, wird an dieser Stelle noch nicht erklärt. Das kommt erst, wenn wir untersuchen, wie man eigene Methoden schreibt.

`kzp` ist einfach ein frei wählbarer *Name*. Ich habe hier `kzp` gewählt, das steht für das etwas längliche Wort *kommandoZeilenParameter*.

2.4 Allgemeine Form eines einfachen Java-Programms

Wer also nun ein einfaches Java-Programm schreiben möchte, verwendet folgende Vorlage und muss nur den eigentlichen Programmblock selbst schreiben (Abb. 7):

```
class Name{
   public Name(<liste mit deklaration der parameter>){
      anweisung;
      anweisung;
         ...
  }
  public static void main(String[] kzp){
     new Name(<liste mit parametern>);
  }
}
```

Abbildung 7: Grundgerüst einer Java-Klasse mit main-Methode

3 Übersetzen und Ausführen eines Java-Programms

3.1 javac

Oben habe ich behauptet, ein Java-Programm besteht aus einer Folge englischer Befehlswörter.

Damit die CPU eines Rechners diese Folge abarbeiten kann, muss sie von einem Programm erst in Maschinencode übersetzt werden. Programme, die diese Übersetzung duchführen, heissen *Compiler*, der Übersetzungsvorgang selbst heisst *compilieren*. Um ein Java-Programm zu compilieren muss man also einen Java-Compiler haben. Der heisst `javac`.
Um das Hello-World-Programm (Abb. 4) zu compilieren, gehen wir so vor:

- Mit einem Texteditor wird der Programmtext aus Abb. 4 erzeugt.
- Dieser Text muss unter dem Namen `Hello.java` gespeichert werden.
- Den Programmtext, den wir mit dem Editor erzeugt haben, nenn man *Quelldatei*.
- Nun wechselt man in ein Terminal-Fenster (wenn's sein muss eine DOS-Box) und gibt folgende Kommandozeile ein:

  ```
  javac Hello.java
  ```

- Der Compiler startet und wenn die Quelldatei keine Fehler enthält, schreibt der Compiler ohne weitere Meldung die Datei `Hello.class` auf die Platte.

Für beliebige Programme sieht die Vorgehensweise so aus:

Aufruf des Java-Compilers:

javac *QuellDateiName*.**java**

- Bei der Angabe der Quelldatei muss man *genau auf Gross- und Kleinschreibung* achten!
- Der Compiler erzeugt aus der Quelldatei mit der Endung .java eine Klassendatei mit der Endung .class.
- Der Dateiname dieser Klassendatei ist dabei *nicht* der QuellDateiName, sondern der Name, der im Quelltext unmittelbar auf das Schlüsselwort **class** folgt!
- Damit das nicht zur Verwirrung führt, sollte man unbedingt die Quelldatei unter dem *gleichen Namen wie die Klasse* abspeichern!
- Bei Verwendung von NetBeans als IDE (Integrated Development Environment) ist es nicht notwendig, den Compiler einzeln aufzurufen. NetBeans erledigt das automatisch und verwendet dabei als Standard das build-Werkzeug **ant** (another neat tool).

3.2 Die virtuelle Maschine *java*

Die meisten Compiler übersetzen einen Quellcode direkt in den Maschinencode eines bestimmten Prozessors.
Bei Java ist das anders. *javac* erzeugt nicht irgendeinen speziellen Maschinencode, sondern einen Zwischencode:

Der Java-Quellcode wird nicht in die Maschinensprache eines bestimmten Prozessors übersetzt, sondern in die Befehle eines einfachen, *virtuellen* Prozessors (Virtual Machine VM). Dieser virtuelle Prozessor wird während der Programmlaufzeit von einer Software, der virtuellen Maschine

java

simuliert.

Die Rolle dieser virtuellen Maschine verdeutlicht nochmals Abb. 8.

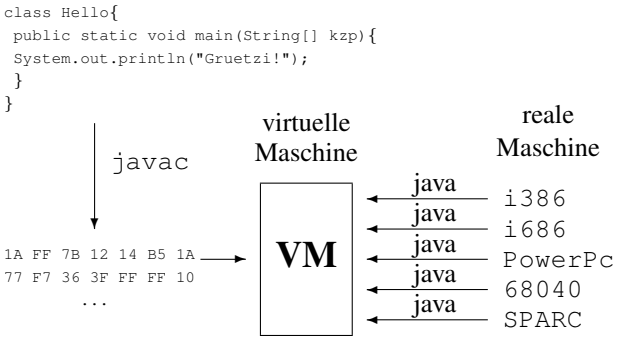

Abbildung 8: Modell der virtuellen Maschine

Wenn wir unser Hello-World-Programm nun endlich ausführen wollen, geben wir im Terminal-Fenster (oder in unsäglich DOS-Box) diese Kommandozeile ein:

- `java Hello`

Das Programm *java* sucht nun im *aktuellen* Verzeichnis nach der Klassendatei `Hello.class`. Das Ganze funktioniert also nur, wenn wir im richtigen Verzeichnis sind.

Im allgemeinen Fall sieht der Start der virtuellen Maschine so aus:

Aufruf der virtuellen Maschine:

java *Klassenname*

- Achtung: Den Klassennamen *ohne* die Endung .class angeben!

Auch beim Starten eines Java-Programms aus NetBeans heraus muss man das nicht alles wissen, NetBeans startet die main-Methode des aktuellen Projekts automatisch (F6 drücken).

3.3 Entwicklungsumgebungen

Da das Übersetzen von Java-Quellcode und das Aufrufen der virtuellen Maschine von Hand doch etwas mühsam ist, haben eine Menge fleissiger Programmierer sogenannte Entwicklungsumgebungen geschrieben, bei denen man per Mausklick oder Funktionstasten (F6, F9) compilieren und die VM starten kann.

Ausserdem bringen diese IDEs einen komfortablen Editor mit, verwalten auch grosse Projekte mit vielen Klassen und erleichtern duch viele Funktionen (z.b. Dot-Complete) das Programmieren erheblich.

Die bekanntesten sind Eclipse und NetBeans.

An der wara wird NetBeans verwendet.

4 Variablen

4.1 Datentypen

Ein Programm ist ein Werkzeug, mit dem Daten bearbeitet werden. Diese Daten können z.B. Grafikdaten eines Spiels, Textdaten (z.B. Name, Vorname, Adresse in einer Datenbank) oder Zahlenwerte (z.B. Kontostand) sein.

Alle Daten lassen sich in zwei Hauptgruppen aufteilen:

- numerische Daten = Zahlen
- alfa-numerische Daten = Zahlen und Buchstaben

Um innerhalb eines Rechners mit diesen unterschiedlichen *Datentypen* arbeiten zu können, muss man jedes *Datum* (Datum ist die Einzahl von Daten) im Speicher ablegen können.

Dazu muss man aber dem Rechner genau mitteilen, welcher Datentyp an welcher Stelle im Speicher steht. Diese Mitteilung eines Datentyps nennt man *Deklaration*. Sie ist nötig, da z.B. die Bitfolge 0100 0001 als numerisches Datum gelesen die Zahl 65 ergibt, als alfa-numerisches Datum aber das Zeichen 'A'.

4.2 Modell eines Speichers

Wie ein Speichermodul aussieht weiss inzwischen fast jeder. Da sein innerer Aufbau und seine Funktionsweise aber äusserst kompliziert sind, stellt man sich den Speicher in seiner Funktion so vor:

Speichermodell:
Einen Speicher kann man sich als Schubladenstapel vorstellen. Dabei erhält jede Schublade einen *eindeutigen Namen*, mit dem man jederzeit wieder auf den Inhalt der Schublade zugreifen kann. Der Schubladeninhalt entspricht den gepeicherten Daten. Er kann jederzeit verändert werden.
Statt des Begriffs Schublade verwendet man in der Programmierung den Begriff ***Variable***.
In jeder Variablen kann man einen bestimmten Datentyp ablegen, der zuvor *deklariert* werden muss.

4.3 Datentypen in Java

Java kennt eine Reihe von Datentypen, von denen wir der Einfachheit halber nur drei verwenden werden:

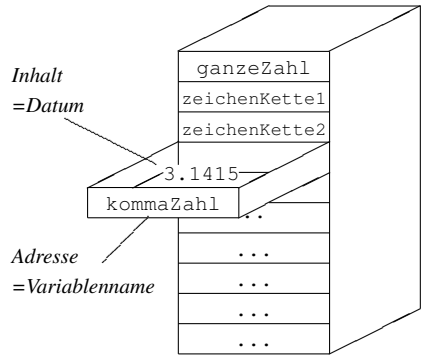

Abbildung 9: Speichermodell

Die drei allerwichtigsten Datentypen in Java:

`String`: das ist der Datentyp für alfa-numerische Zeichen*ketten* (Kette=String).

`int`: int steht für Integer. Integer-Datentypen sind alle **ganzen** Zahlen zwischen -2 147 483 648 und +2 147 483 647. Eine Integer-Variable ist in Java immer 32 bit gross.

`double`: eine double-Variable kann eine 64-Bit Gleit**kommazahl** aufnehmen.

Achtung: Das Schlüsselwort `String` wird grossgeschrieben, `double` und `int` dagegen klein!

Abbildung 10 zeigt ein Beispiel mit ein paar Deklarationen.

Neben diesen drei Datentypen benötigt man von Fall zu Fall noch die folgenden Beiden:

`long`: long nimmt ebenfalls ganze Zahlen auf. Eine `long`-Variable ist aber 64 bit gross. Daraus ergibt sich ein Wertebereich von $-2^{63} \cdots + 2^{63}$.

`boolean`: Variablen dieses Typs können nur zwei Werte annehmen: `true` oder `false`.
Beispiel: `boolean binaer = true;`

Die eigentlichen Deklarationen stehen ganz am Anfang in den Zeilen 3 bis 7.

Anschliessend werden die Variablen mit Werten gefüllt und auf dem Bildschirm ausgegeben. Allgemein sieht eine Deklaration so aus:

```
class Beispiel{

     int   ganzeZahl;
     String zeichenKette1;
     String zeichenKette2;
     double kommaZahl;
     boolean binaer;

Beispiel(){

     ganzeZahl = 100;
     binaer = true;
     zeichenKette1 = "eine Zeichenkette";
     zeichenKette2 =
        "steht zwischen Gaensefuesschen";
     kommaZahl = 3.1415;

     System.out.println(
                 "Typ int:" + ganzeZahl);
     System.out.println("Typ String:" +
                         zeichenKette1 +
                         zeichenKette2);
     System.out.println(
                 "Typ double:" + kommaZahl);
      System.out.println(
                 "Typ boolean:" + binaer);
  }
  public static void main(String[] kzp){
     new Beispiel();
  }
}
```

Abbildung 10: Die `main`-Methode

Deklaration einer Variablen:

Variablentyp Variablenname;

- Typ kann sein: `int`, `double` oder `String`.
- Der Name *muss* mit einem Kleinbuchstaben beginnen.
- Die Variable muss deklariert werden, bevor man sie das erste Mal verwendet!

Die Deklaration einer Variablen kann also an beliebiger Stelle im Programm erfolgen, man muss nur sicherstellen, dass man eine Variable deklariert, bevor man sie verwendet.

5 Methoden

Methoden sind nichts anderes wie Unterprogramme, also Programmteile, die man mehrfach verwendet und nicht jedesmal neu in den Quelltext schreiben möchte.

Jede Methode hat einen eindeutigen Namen. Dieser *muss* mit einem Kleinbuchstaben beginnen.

Jedesmal wenn man im Programm diesen Namen aufruft, werden alle Befehle der Methode ausgeführt.

5.1 Eine Beispielmethode

Damit wir gleich zur Sache kommen, schreiben wir eine Methode, die das Maximum zweier Zahlen ermittelt und zurückliefert (Abb. 11). Was die einzelnen Programmzeilen bedeuten, folgt in den nächsten beiden Unterkapiteln.

5.1.1 Wie man eine Methode schreibt

Hier zunächst eine schrittweise Anleitung:

1. Wir legen fest, was unsere Funktion *macht*: Sie soll das Maximum zweier Zahlen bestimmen. Die beiden Zahlen werden der Methode *übergeben*.
2. Wir legen einen *Bezeichner* für die Methode fest. Der Bezeichner ist also nichts anderes als der Methodenname. Für die Maximum-Methode wählen wir den Bezeichner: `maxi`.
3. Wir legen als nächstes fest, welche Daten unsere Methode *übergeben* bekommt und welche Typen diese Daten haben: die max-Methode bekommt *zwei* Daten vom Typ *int* übergeben. Im Beispiel unten sind dies `int aPar` und `int bPar`.
4. Und schon wieder wird was festgelegt: was gibt uns die Funktion als *Ergebnis* zurück: Die max-Funktion wird *einen* `int`-Wert zurückliefern.
5. Zum Schluss wird festgelegt, *wer* die Funktion aufrufen darf: wir erlauben es jedem, dafür gibt es das Schlüsselwort `public`.

5.1.2 Methodendeklaration

Bevor man eine Methode verwenden kann, muss sie -ebenso wie die Variablen- deklariert werden. Dies geschieht im Beispiel Abb. 11 ganz am Anfang. Die Methodendeklaration besteht immer aus einem Methodenkopf und einem Methodenrumpf.

Der Kopf ist hier die Zeile:

```
public int maxi(int aPar, int bPar)
```

Dabei bedeuten die Schlüsselwörter und Namen Folgendes:

- `int`: int ist der Datentyp des Rückgabewerts. Unsere Methode liefert uns das Maximum zweier Zahlen zurück und das ist ein Integer-Wert.
- `maxi`: das ist einfach unser *Methodenname*. Dieser ist frei wählbar, nur muss er mit einem Kleinbuchstaben beginnen.
- `(int aPar, int bPar)`: Das ist die sog. *Parameterliste*. Sie steht in runden Klammern. Innerhalb dieser Klammern stehen zwei Deklarationen für die Variablen `aPar` und `bPar`.

```
class MethodenBeispiel{

    public MethodenBeispiel(){
        int a = 10;
        int b = 20;
        System.out.println("Das Maximum ist: "
                           + maxi(a,b) );
    };

    //hier beginnt die methode maxi(int,int)
    public int maxi(int aPar, int bPar){
        if (aPar > bPar){
            return aPar;
        }else{
            return bPar;
        }
    }


    public static void main(String[] kzp){
        new MethodenBeispiel();
    }
}
```

Abbildung 11: Ein Beispielprogramm mit einer Methode

Wenn man irgendwo in einem Java-Programmtext einen Namen gefolgt von einer runden Klammer findet, handelt es sich um eine Methode.

Der Methodenrumpf ist der Block, der direkt auf den Kopf folgt. Der Rumpf enthält zweimal den Befehl `return`. Dadurch wird entweder der Wert von `aPar` oder `bPar` an denjenigen zurückgegeben, der die Methode aufruft. Und da `aPar` und `bPar` vom Typ `int` sind, muss auch im Kopf der Rückgabewert als `int` deklariert werden.

5.1.3 Methodenaufruf mit Parameterübergabe

Wie kann nun diese Methode aufgerufen werden und was passiert dann?

Im Methodenkopf haben wir in der Parameterliste zwei *neue* Variablen vom Typ `int` deklariert. In diese Variablen werden beim Methodenaufruf zwei Werte *kopiert*. Diese beiden Werte muss man beim Methodenaufruf angeben. Hierzu gibt es drei Möglichkeiten:

1. Man nimmt direkt Zahlenwerte: maxi(10,6);
2. Man kann wiederum andere Variablen angeben: maxi(a,b);
3. Natürlich kann man auch Zahlenwerte und Variablen kombinieren: maxi(a,77);

Im Beispiel wird die zweite Methode verwendet. In der `main`-Methode werden zunächst die beiden Variablen a und b deklariert und gleichzeitig mit Werten gefüllt. Dieses Deklarieren und gleichzeitige Initialisieren findet man sehr häufig:

```
int a = 10;
int b = 20:
```

Dann wird unsere maxi-Methode mit den Variablen a und b als aktuelle Parameter aufgerufen:

```
maxi(a,b);
```

Beim Aufruf von `maxi(a,b)` werden die **Inhalte** von a und b in die entsprechenden Variablen der Methode `maxi` **kopiert**.

Parameterübergabe:

- Beim Methodenaufruf werden die aktuellen Werte, die sog. *aktuellen Parameter* oder auch *Argumente* in die Variablen die im Methodenkopf stehen, die sog. *formalen Parameter*, kopiert.
- Beim Kopieren spielen die Variablennamen keine Rolle. Kopiert wird ausschliesslich entsprechend der Reihenfolge der Parameter.
- Die Anzahl der formalen Parameter und der aktuellen Parameter *müssen exakt* übereinstimmen.
- Die Datentypen der formalen und aktuellen Parameter müssen an den entsprechenden Stellen übereinstimmen.

Die Übergabe verdeutlicht nochmals Abb. 12.
Im Beispiel ist dieser Methodenaufruf sogar wieder selbst ein Argument, er steht nämlich innerhalb der runden Klammern der `println`-Methode: den Wert, den `maxi(a,b)` zurückliefert, wird gleich an die Methode println() weitergereicht.

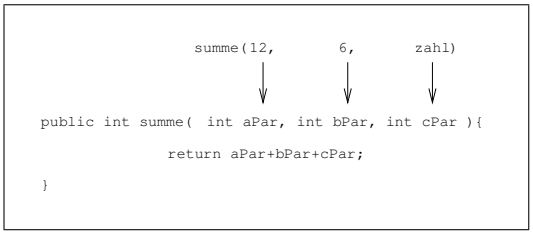

Abbildung 12: Aktuelle und formelle Parameter

5.2 Die `println`-Methode

In den letzen Unterkapiteln haben wir fleissig Gebrauch von einer speziellen Methode zur Bildschirmausgabe gemacht:

```
System.out.println("zahl");
System.out.println(5);
System.out.println("zahl ="+5);
```

Die Methode heisst `println`. Das Wörtchen `out` davor zeigt an, dass `println` seinen Datenstrom an die *Standardausgabe* schicken soll und `System` ist die Klasse, wo der Kompiler `println` und `out` findet.

Was aber an dieser Stelle sehr wichtig ist, ist der Parameter von `println`: man darf `println` beliebige Datentypen übergeben und `println` wird sie brav alle ausdrucken.

Möchte man allerdings Strings und numerische Werte zusammen ausgeben, muss man diese mit dem **+**-Zeichen *verketten*. Das **+**-Zeichen steht in einem solchen Fall nicht mehr für die Addition sondern für das Aneinanderhängen von Strings.

Bedingung dafür ist aber, dass mindestens einer der beiden Ausdrücke rechts oder links vom **+**-Zeichen ein String ist.

Mit dem **+**-Zeichen kann man natürlich auch zwei Strings *verketten*. Das ist oft vorteilhaft, wenn der Ausgabetext so lang ist, dass er im Programmtext auf der nächsten Zeile weitergeschrieben werden soll:

```
System.out.println("am ende dieser zeile" +
                   "steht ein plus.");
```

Die Ausgabe auf dem Bildschirm sieht dann so aus:

```
am ende dieser zeile steht ein plus.
```

Da ich beim zweispaltigen Satz dieses Dokuments wenig Platz für lange Zeilen habe, habe ich von dieser Möglichkeit bei der Darstellung von Quellcode oft Gebrauch gemacht.

5.2.1 Wozu die main-Methode da ist

Eine Sonderstellung nimmt die main-Methode ein. Wir wissen ja schon, dass es in einem Java-Programm mindestens eine main-Methode geben muss.

Der Trick bei der main-Methode ist das Schlüsselwort **static**

> **static** heist nichts anderes wie: lade sofort in den Speicher

Man kann aber in jeder Klasse eine main-Methode schreiben. Beim Start des Programm muss man dann allerdings angeben, von welcher Klasse die main-Methode gestartet werden soll. Bei NetBeans kann man das über die Projekteigenschaften (Properties) festlegen.

Da die main-Methode nun statisch ist (static) wird sie als allererstes in den Speicher geladen und kann dann ausgeführt werden (wenn sie nicht im Speicher steht, kann sie ja nicht ausgefuehrt werden).

Sobald die main-Methode läuft, kann diese wiederum ein Ojekt aus einer Klasse erzeugen, das dann auch im Speicher steht und dessen Methoden damit ebenfalls ausgeführt werden können. Dieses Objekt kann dann weitere Objekte anlegen und so fort.

Man kann also folgende Parallele ziehen:

Die main-Methode ist so eine Art boot-loader für das Java-Programm.

Die Form des Kopfs der main-Methode ist festgeschrieben, nur der Name des String-Arrays, das beim Aufruf vom Betriebssystem übergeben wird, ist frei wählbar. Hier habe ich das `kzp` (kommando zeilen parameter) genannt, meistens steht aber `args` dort:

```
public static void main(String[] kzp){ ... }
```

6 Kontrollstrukturen

Unter Kontrollstukturen versteht man Befehle, mit denen der geradlinige Ablauf eines Programms verändert werden kann. An dieser Stelle beschränken wir uns auf zwei Arten von Kontrollstukturen:

1. Bedingte Ausführung von Blöcken
2. Wiederholte Ausführung von Blöcken in Schleifen

6.1 Bedingte Ausführung

Eine bedingte Ausführung haben wir bereits in Abb. 11 kennengelernt. In allgemeiner Form sieht die bedingte Ausführung so aus wie in Abb. 13

```
if (bedingung){
   anweisung;
   anweisung;
      ...
}
```

Abbildung 13: Die if-Anweisung

Der Block wird eben nur dann ausgeführt, wenn die Bedingung eintrifft. Wenn sie nicht eintrifft, wird der Block ausgelassen.

Dann gibt es noch die Variante aus Abb. 14.

Hier wird wahlweise einer der beiden Blöcke ausgeführt. Trifft die Bedingung zu, der erste Block, andernfalls der Block der nach `else` folgt.

Eigentlich ist die erste Variante (ohne `else`) ein Spezialfall der zweiten, bei der der Block hinter `else` leer ist. In diesem Fall wird das `else` weggelassen.

6.2 Struktogramm der bedingten Ausführung

Um sich den Programmablauf der bedingten Ausführung besser zu verdeutlichen, zeichnet man sich am besten ein **Struktogramm**. Abb. 15 zeigt, was damit gemeint ist. Im Beispielstruktogramm sind beide Blöcke mit Anweisungen gefüllt. Soll im else-Fall

```
if (bedingung){
   anweisung;
   anweisung;
      ...
}else{
   anweisung;
   anweisung;
      ...
}
```

Abbildung 14: Die if-Anweisung

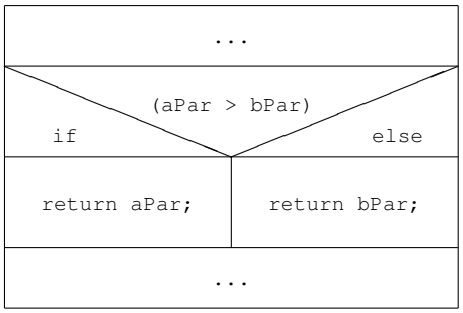

Abbildung 15: Struktogramm der If-Else-Anweisung

nichts ausgeführt werden, bleibt der else-Zweig leer. In Abb. 13 sind die beiden bedingt auszuführenden Blöcke nebeneinander angeordnet. Der Programmablauf verzweigt also an dieser Stelle entweder in den if-Block oder den else-Block.

Im Quellcode kann man aber Blöcke nicht nebeneinander anordnen, sondern muss sie hintereinander schreiben. Den Ablauf des Programms stellt man sich aber am besten wie im Stuktogramm gezeigt vor.

6.2.1 Bedingungen

Die ganze Zeit über wird hier von Bedingungen gesprochen, aber bis jetzt wurde dieser Begriff noch gar nicht erklärt.

An dieser Stelle soll Folgendes genügen:

- Eine Bedingung steht stets in runden Klammern
- Die Bedingung kann nur **wahr** oder **falsch** sein
- Es gibt die folgenden Vergleiche:
  1. `(a<b)` : wahr, wenn a kleiner b
  2. `(a<=b)` : wahr, wenn a kleiner oder gleich b
  3. `(a>b)` : wahr, wenn a grösser b
  4. `(a>=b)` : wahr, wenn a grösser oder gleich b

5. `(a==b)`: wahr, wenn a gleich b
6. `(a!=b)`: wahr, wenn a ungleich b

Achtung:
Die Gleichheit wird mit einem **doppelten** Gleichheitszeichen getestet!

6.3 Schleifen

Neben der bedingten Ausführung eines Blocks gibt es noch die wiederholte Ausführung eines Blocks. Soetwas nennt man eine **Schleife**. Hier gibt es drei Arten:

1. Kopfgesteuerte Schleife
2. Fussgesteuerte Schleife
3. Zählschleife

Bei allen drei Arten wird ein Block ständig wiederholt, solange eine Bedingung zutrifft. Lediglich der Zeitpunkt, zu dem die Bedingung geprüft wird, ist bei den drei Arten verschieden.

6.3.1 While-Schleife

Abb. 16 zeigt das Struktogramm der kopfgesteuerten Schleife. Anstelle des Java-Schlüsselworts `while`, habe ich im Struktogramm einen deutschen Ausdruck verwendet. Auf diese Weise erklärt das Struktogramm sich selber. Abbildung 17 zeigt ein Java-Beispiel

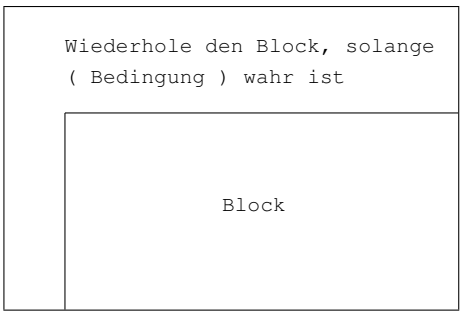

Abbildung 16: Kopfgesteuerte- oder While-Schleife

mit einer While-Schleife:

6.3.2 Do-While-Schleife

Das Struktogramm der fussgesteuerten Schleife sieht so aus (Abb. fussSchleife): Abbildung 19 zeigt das zugehörige Java-Beispiel.

```
int i = 0;
while (i < 10){
   System.out.println("Zaehlerstand: " + i);
   i = i + 1;
}
```

Abbildung 17: Ein Beispiel für eine While-Schleife

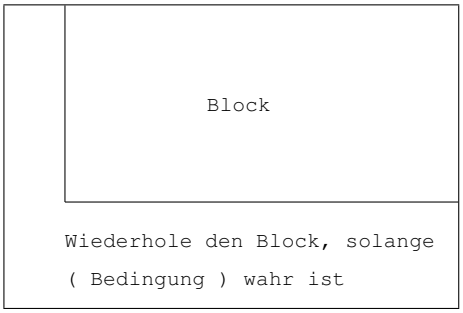

Abbildung 18: Fussgesteuerte- oder Do-While-Schleife

Achtung:

- Bei der Do-While-Schleife wird der Schleifenblock immer mindestens einmal durchlaufen, auch wenn die Bedingung nicht erfüllt ist!
- Bei der While-Schleife dagegen wird der Schleifenblock überhaupt nicht ausgeführt, wenn die Bedingung schon von Anfang an falsch ist.

6.3.3 Die Zähl- oder For-Schleife

Diese dritte Schleife ist der While-Schleife sehr ähnlich. In der Tat kann man jeder Art von Zählschleife mit der While-Schleife realisieren und braucht die For-Schleife eigentlich gar nicht. Da der Quellcode der For-Schleife aber sehr bequem zu formulieren ist, wird sie sehr häufig verwendet.

Hier also gleich mal wieder ein Beispiel (Abb. 20:

Die For-Schleife beginnt mit dem Schlüsselwort `for`. Dann folgt eine Klammer, die drei Ausdrücke enthält:

1. `int i=0;` ← **Initialisierung**
2. `i<10;` ← **Bedingung**
3. `i=i+1;` ← **Reinitialisierung**

Beim Vergleich der For-Schleife mit der While-Schleife aus Abb. 17 wird die Gemeinsamkeit deutlich: die For-Schleife ist eine While-Schleife, die um die Initialisierung erweitert wurde.

```
int i = 10;
do{
   System.out.println("Zaehlerstand: " + i);
   i = i + 1;
}while (i < 10);
```

Abbildung 19: Ein Beispiel für eine Do-While-Schleife

```
for(int i=0; i<10; i=i+1){
   System.out.println("Zaehlerstand: " + i);
}
```

Abbildung 20: Die For-Schleife

Entsprechend kann man ein Struktogramm wie in Abb. 21 verwenden.

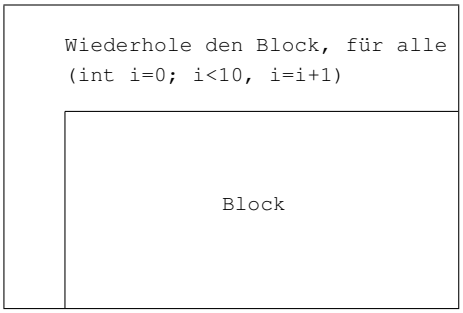

Abbildung 21: Struktogramm der For-Schleife

Zum Abschluss der Kapitel über bedingte Ausführung und Schleifen gibt es ein Java-Programm, das alle Primzahlen bis zu einer Obergrenze ausgibt.

Damit das Programm lesbarer wird, sind eine Menge Kommentare eingefügt. Eine Kommentarzeile wird mit // eingeleitet, ein Kommentarblock steht zwischen /* und */.

Der Quellcode zu diesem Programm steht am Ende im Anhang (Abb. 35).

7 Objektorientierte Programmierung

Alles was wir bis hier programmiert haben, war noch nicht objektorientiert. Was es nun mit der Objektorientierung auf sich hat und warum sie so beliebt geworden ist, steht ab hier.

Eigentlich kann man mit den oben besprochenen Schleifen und der bedingten Ausführung alle Probleme in der Softwareentwicklung lösen. Allerdings passen die meisten Anforderungen der Datenverarbeitung nur sehr schlecht zu diesen einfachen Werkzeugen, so

dass Softwareentwicklung damit sehr mühsam wird.

Gerade die immer bunteren, grafischen Benutzeroberflächen lassen sich sinnvoll nur noch objektorientiert programmieren.

7.1 Das unvermeidliche Auto-Beispiel

Ein Auto ist unbestritten ein Objekt mit bestimmten *Eigenschaften* (`farbe, bauJahr, hubRaum`). Und ein Auto hat auch bestimmte Funktionen. Diese lassen sich dadurch beschreiben, was mit dem Auto geschehen kann: anlassen, beschleunigen, die Hupe drücken, usw. Lasst uns mal diese Funktionen als *Methoden* bezeichnen.

Mit den Methoden wird beschrieben, was mit dem Objekt *geschehen* kann

Im Beispiel unseres Autoobjekts könnte es also folgende Methoden geben:

- *betanken*
- *starten*
- *beschleunigen*
- *hupen*
- ...

Und das Autoobjekt hat vielleicht folgende Eigenschaften:

- farbe
- hubraum
- baujahr
- ...

Die objektorientierte Programmierung erleichtert es nun, die reale Welt im Computer abzubilden:

- reale Eigenschaften → Variablen = Speicherplätze
- aus den realen Methoden werden dementsprechend Java-Methoden:
  - *betanken* → betanken()
  - *starten* → starten()
  - *beschleunigen* → beschleunigen()
  - *hupen* → hupen()
  - ...

Eigenschaften und Methodennamen:

- **Die Namen von Eigenschaften und Methoden müssen mit einem Kleinbuchstaben beginnen!**
- **Immer wenn auf einen Namen eine runde Klammer folgt, handelt es sich um eine Methode.**

7.2 Wie sich Objekte im Speichermodell darstellen lassen

Für jemanden der jahrelang strukturiert mit Pascal oder reinem C programmiert hat, gibt es noch eine andere Sichtweise auf den Begriff Objekt:

In C und Pascal gibt es Variablen. Eine Variable ist eine recht einfache Datenstruktur, mit der man z.B. nur schlecht Datensätze erfassen kann. Eine Variable speichert nunmal nur einen Wert.

Von der einfachen Variablen gelangt man zum Variablenfeld, dem Array. Schon besser für das Bearbeiten von Daten, aber ein Array enthält nur Daten gleichen Typs. Und da haben sich Brian und Dennis bzw. Nikolaus die Datentypen `struct` bzw. `record` ausgedacht.

In einem Record können z.B. sehr einfach die Daten einer Person (Name, Vorname, Tel.Nr., ...) gespeichert werden.

Um jetzt z.B. die Personen einer Kartei nach Namen sortiert auszugeben, schreibt man sich noch schnell eine Sortier-Prozedur.

Und dann schreibt man noch viele weitere Prozeduren, um Daten zu erfassen, zu löschen, etc. . Und je mehr Prozeduren man schreibt, umso schwieriger wird deren Verwaltung: Datentypen, Parameteranzahl, Namenskonflikte, etc. : all dies muss man berücksichtigen und im Auge behalten.

Und da wird es endlich Zeit, zur nächsthöheren Datenstruktur überzugehen: eben dem Objekt.

Ein Objekt ist nichts anderes als ein Record (=eine Struktur), der gleichzeitig alle Methoden zur Manipulation und Verwaltung seiner Datenfelder enthält.

So stellt sich ein Objekt zumindest aus der Sicht des Entwicklers dar. In Wirklichkeit werden die Methoden nicht für jedes Objekt extra gespeichert, sondern stehen nur einmal im Speicher. Die ganze Arheit mit dem Verwalten der Methoden/Prozeduren nimmt uns aber der Compiler ab.

7.3 Was Variablen und Objekte gemeinsam haben

Im vorhergehenden Kapitel (Kap. 7.2) wurde gezeigt, dass ein Objekt eine *Datenstruktur* ist, die im Speicher steht.

Mit Hilfe eines *Schubladenmodells* kann man sich den Übergang von einfachen Datentypen zu Objekten veranschaulichen. Zuersteinmal stellt man sich den Hauptspeicher als *Stapel* von *Schubladen* vor. Jede Schublade entspricht hierbei einer Variablen. Die Schublade ist vorne mit dem Variablennamen beschriftet, während der Variableninhalt dem Schubladeninhalt entspricht. Abb. 22 zeigt ein solches Speichermodell.

Einfache Datentypen kann man nun als Schubladen ohne weitere interne Unterteilung darstellen. Abb. 23 zeigt das Modell der Variablen `pi` mit dem Wert `3.1415`.

Fasst man mehrere *gleiche* Datentypen zusammen, erhält man eine Schublade, die im Inneren regelmäßig, wie ein Setzkasten, unterteilt ist. Dieser Datentyp wird *Array* oder *Liste* genannt. Abb. 24 zeigt das zugehörige Modell am Beispiel eines Arrays von 3 Zeichenketten (`String[]`). Das ganze Array hat den Namen `args`, während die einzelnen Elemente `args[0]`, `args[1]` und `args[2]` heissen [1].

Lässt man nun innerhalb der Schublade *verschiedene* Datentypen zu, kommt man zu

[1] Auch in Java gibt es Arrays. Im Unterschied zu den hier dargestellten Arrays haben die Schubladen von Java-Arrays noch ein Fach mit dem Namen `length`, in dem immer die Anzahl der Array-Elemente steht.

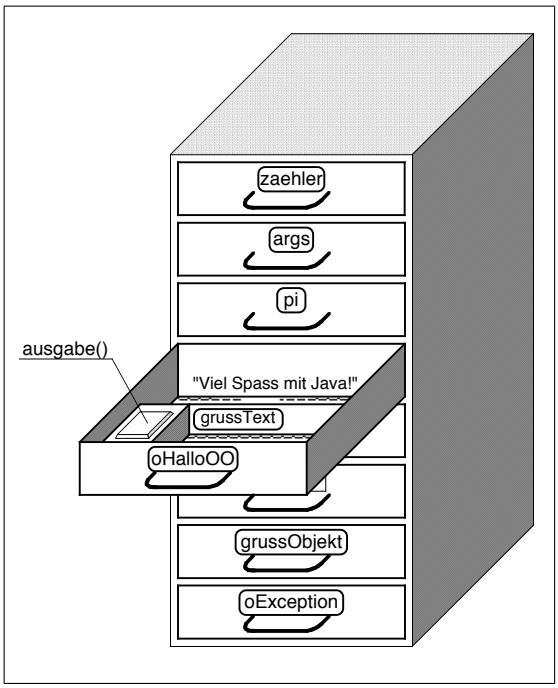

Abbildung 22: Der Speicher als Schubladenstapel

einer Datenstruktur, die man `record` (in Pascal) oder `struct` (in C) oder einfach *Datensatz* nennt. Abb. 25 zeigt dieses Modell. Der Schubladenname entspricht hier dem Namen des Datensatzes, während die Namen der internen Fächer den Zugriff auf deren Inhalt ermöglichen.

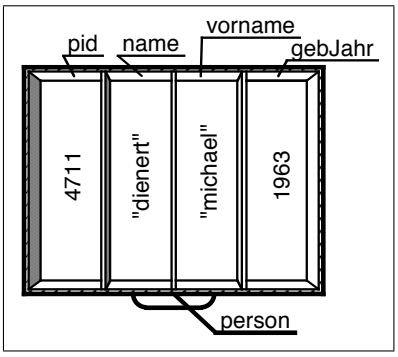

Abbildung 25: Modell eines Datensatzes

Jetzt wird die Schublade nochmals erweitert: zusätzlich zu den Unterfächern wird ein Feld mit *Funktionstasten* eingebaut. Mit diesen Tasten kann man Funktionen starten, die mit den Daten in den Unterfächern arbeiten. In Abb. 22 ist die Schublade `oHalloOO` als ein Objekt der Klasse `hallooo` (vergl. Abb. 26) dargestellt. Dies ist ein recht einfaches Objekt, es besitzt ein *Datenfeld* (Schubladenfach) für einen Grusstext und eine Methode (Funktionstaste), die diesen Grusstext auf dem Bildschirm ausgibt.

Bei einer Methode mit Rückgabewert springt einem beim Betätigen der Funktinstaste dieser Wert sozusagen entgegen. Damit man den Wert richtig auffangen kann, wird bei der Funktionsdeklaration der Typ des Rückgabewerts angegeben.

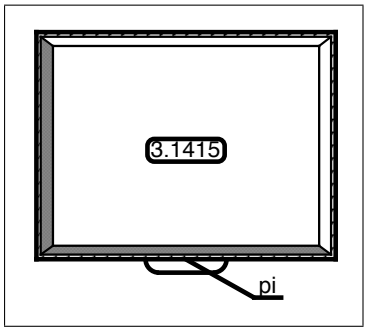

Abbildung 23: Modell einer einfachen Variablen

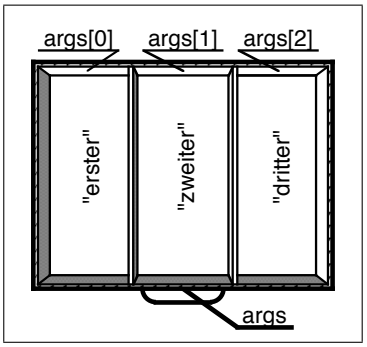

Abbildung 24: Modell eines Arrays

Die Sprache Java kennt nur Objekte aber keine Records oder Structures. Dies ist auch nicht nötig, denn ein Record ist ja nichts anderes als ein Objekt ohne Funktionen.

Anhand des Schubladenmodells kann man nun auch leicht erklären, wie man auf die Datenfelder (Schubladenfächer) und Funktionen eines Objekts zugreifen muss: zuerst muss man die entsprechende Schublade aufziehen, d.h. ihren Namen angeben und dann erst kann man eine darin enthaltene Funktionstaste betätigen oder auf ein Datenfeld zugreifen.

In Java-Syntax wird das mit der Punkt-Notation geschrieben:

Zugriff auf Eigenschaft: `oHalloOO.grusstext = "Viel Spass mit Java"`

Funktionsaufruf: `oHalloOO.ausgabe()`

7.4 Wir bauen ein Auto(objekt)

Jetzt wird es Zeit, ein Auto in Java zu bauen. D.h. wir wollen nun ein Autoobjekt programmieren.

Genau wie man in der realen Welt vor dem Bau eines Autos erst mal einen Plan zeichnet, müssen wir auch zuerst einen Plan zeichnen oder besser gesagt, programmieren.

Eine Klasse ist der Bauplan für ein Java-Objekt.

Also müssen wir eine Klasse schreiben. Bevor man aber gleich den Java-Code in den Rechner hämmert, kann man ein sog. *Klassendiagramm* zeichnen. Das zeigt übersichtlich den Klassennamen, die Eigenschaften und Methoden einer Klasse auf.

Das Klassendiagramm unseres Autos zeigt Abb. 27. Es besteht aus drei Teilen:

- ganz oben steht der Klassenname
- in der Mitte die Eigenschaften
- unten die Methoden

Eigentlich wollte ich als Methode `hupen` nehmen, aber ein Computer hupt nicht und wie man die Sound-Karte mit Java ansteuert, ist noch nichts für diesen Einsteigerkurs.

```
class HelloOO{

   //eigenschaften
   private String grussText;

   //konstruktor
   HelloOO(String grussTextVar){
      grussText = grussTextVar;
   }

   //methoden
   public void ausgabe(){
      System.out.println(grussText);
   }

   public static void main(String[] args){
      HelloOO text =
         new HelloOO("Gruetzi wohl!");
      text.ausgabe();
    }
}
```

Abbildung 26: Objektorientiertes Hello-Programm

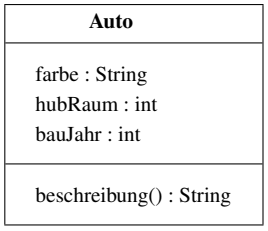

Abbildung 27: Klassendiagramm der Auto-Klasse

Also habe ich stattdessen die Methode `String beschreibung()` verwendet. Diese Methode kann man sich als Knopf am Armaturenbrett vorstellen, der, wenn man ihn drückt einen String (Zeichenkette) “ausspuckt”. Z.B. den String

`Dies ist ein blaues Auto, gebaut 2001 mit 3000 ccm Hubraum.`

Schön, jetzt hat man also eine Auto-Klasse und gleich ein paar Probleme:

- Wie soll aus der Klasse ein Objekt werden?
- Weiter oben im Text stand mal was von einer `main`-Methode, wo bleibt denn die?
- Wer ruft die Methode `beschreibung()` auf?
- Wer fängt den String auf, den beschreibung() zurückgibt?

```
class Auto {

    //eigenschaften
    String farbe;
    int bauJahr;
    int hubRaum;

    //methode(n)
    String beschreibung() {
        return "\n\nDies ist ein " +
            farbe +
            "es Auto, gebaut " +
            baujahr +
            " mit " +
            hubraum +
            " ccm Hubraum.\n\n";
    }
}
```

Abbildung 28: Eine Auto-Klasse mit einer Methode

Abb. 28 zeigt zunächst den Java-Code der Auto-Klasse.

Wenn man wieder den Vergleich heranzieht, eine Klasse ist ein Bauplan, dann hat man bis jetzt nur ein Stück Papier (eben den Plan) und noch kein Auto.

Um das Auto zu bauen, also im Speicher ein Auto-Objekt zu erzeugen, muss man folgende Anweisung verwenden:

```
    ferrari = new Auto();
```

`ferrari` ist der Name des Objekts. `new` ist das Java-Schlüsselwort, mit dem das Objekt gebildet wird und `Auto()` ist ein ganz spezielle Methode (runde Klammern!), die den Bauvorgang steuert. Sie hat den gleichen Namen wie die zugehörige Klasse.

Weiter oben haben wir Variablen deklariert. Variablen sind Speicherplätze für einfache Datentypen (int, double). Nun muss aber auch unser Objekt im Speicher einen Lebensraum bekommen, den man genau wie bei den Variablen deklarieren muss. Bevor man das `ferrari`-Objekt erzeugen kann, muss man zuvor eine *Deklaration* für den Speicherplatz des Objekts machen. Vollständig sieht die Objekterzeugung dann so aus:

```
    Auto ferrari;\\
    ferrari = new Auto();
```

Die erste Zeile sieht exakt aus wie die bekannte Variablendeklaration. Und man kann ja auch wirklich sagen:

`ferrari` ist ein Objekt vom Typ Auto.

Ein Objekt ist also die *erweiterte Form* einer Variablen. In einem Objekt wird nicht nur ein einzelnes Datum gespeichert, sondern:

- evtl. viele Daten: **die Eigenschaften**

- und **Methoden**, mit denen man die Eigenschaften manipulieren kann.

Die zweiteilige Form zur Objekterzeugung kann man auch abkürzen:

```
    \fbox{\parbox{7cm}{ Auto ferrari = new Auto(); }}
```

Nun könnte man natürlich fragen, wenn ein Objekt die erweiterte Form einer Variablen ist, wozu brauche ich dann noch Variablen? Statt einer Variablen könnte man ja ein Objekt einer Klasse nehmen, die nur eine Eigenschaft und keine Methoden hat.

Und da waren die Java-Erfinder (James Gosling) nicht konsequent: Java ist nicht 100%-tig objektorientiert, es gibt noch einfache Variablen, die eigentlich unnötig sind. Wahrscheinlich wollte man durch das Beibehalten von Variablen viele Leute dazu bringen, von C++ nach Java zu wechseln.

Jetzt zurück zu den Problemen mit der Auto-Klasse. Wir brauchen für ein vollständiges Java-Programm auf jeden Fall eine main-Methode. Und innerhalb dieser main-Methode können wir dann Autos bauen:

```
class AutoTest{
   public static void main(String[] kzp){
      Auto ferrari;
      ferrari = new Auto();
      ferrari.farbe = "ferrarirot";
      ferrari.hubRaum = 3500;
      ferrari.bauJahr = 2001;
      System.out.println(
                 ferrari.beschreibung());
   }
}
```

Abbildung 29: Ein Beispiel für eine Do-While-Schleife

7.5 Instanzmethoden

Vielleicht erinnert sich noch jemand an das Schlüsselwort `static` bei Methoden. Bei der Methode `beschreibung()` der Klasse `Auto` wird es nicht mehr verwendet. Methoden, die *ohne* das Schlüsselwort `static` deklariert werden, bezeichnet man als *Instanzmethoden*. Solche Methoden kann man erst dann verwenden, wenn man ein Objekt der Klasse erzeugt hat.

Um zum Autobeispiel zurückzukehren: den Hupenknopf kann ich eben erst drücken, wenn ein Auto gebaut worden ist. Den Knopf auf dem Bauplan zu drücken bringt nicht viel. Ebenso kann ich die Methode beschreibung() nur in Verbindung mit einem Objekt aufrufen, solange kein Objekt gebildet worden ist, ist die Methode `beschreibung()` nicht zugänglich. Die Verbindung von Objekt und Methode ist der Punkt:

```
    ferrari.beschreibung();
```

Instanz ist ein anderer Begriff für Objekt. Durch den Begriff Instanzmethode will man eben die Verbindung von Methode und Objekt verdeutlichen.

7.6 Punktnotation und Eigenschaften

Genauso wie auf die Methoden eines Objekts zugegriffen wird, muss man auch dessen Eigenschaften ansprechen:

```
ferrari.farbe = "ferrarirot";
ferrari.hubRaum = 3500;
ferrari.bauJahr = 2001;
```

Diese Schreibweise hat ihren Namen vom Punkt, der auch hier die Verbindung zwischen Objekt und seiner Eigenschaft herstellt.

7.7 Der Konstruktor baut das Auto

Zum Abschluss dieser Einführung muss unbedingt noch der Begriff Konstruktor erkärt werden. Er ist zu wichtig, um ihn wegzulassen. Und ausserdem ist er nicht schwierig zu verstehen.

Unser Autoobjekt hat die Eigenschaften `farbe, bauJahr, hubRaum`. Direkt nach der Erzeugung eines Objekts mit z.B.

```
Auto porsche;
porsche = new Auto();
```

haben aber alle diese Eigenschaften noch gar keine Werte zugewiesen bekommen. Da es aber keinen Sinn macht, ein Auto ohne Farbe zu bauen, müsste man doch irgendwie sicherstellen können, dass immer wenn ein Autoobjekt gebaut wird, auch die entsprechenden Eigenschaften mit Werten initialisiert werden.

Und natürlich gibt es in Java für diese Aufgabe eine Lösung: zum Initialisieren der Eigenschaften eines Objekts dient eine ganz spezielle Methode, die man *Konstruktor* nennt und die die folgenden, besonderen Eigenschaften besitzt:

- der Konstruktor wird automatisch immer dann aufgerufen, wenn ein neues Objekt mit **new** erzeugt wird
- der Konstruktor hat *immer* denselben Namen wie die Klasse, in der er definiert ist; sein Name beginnt deshalb auch mit einem Grossbuchstaben
- der Konstruktor liefert *keinen* Wert zurück; es darf daher auch kein Rückgabetyp angegeben werden, auch nicht **void**
- man kann dem Konstruktor allerdings Parameter mitgeben, mit denen das Objekt initialisiert wird
- wird der Konstruktor nicht definiert, verwendet Java einen Default-Konstruktor ohne Parameter.

Am besten macht man sich den Begriff Konstruktor mit der Übung im nächsten Kapitel klar:

7.8 Übung zu Methoden und Konstruktoren und dem Schlüsselwort `public`

In dieser Übung wird Schritt für Schritt ein Konstruktor geschrieben. Dabei wird auch gleich noch die Steuerung der Sichtbarkeit von Methoden und Eigenschaften erklärt.

1. direkter Zugriff auf die Eigenschaften
2. Zugriff mit sog. set-Methoden
3. Zugriff mit einer einzelnen set-Methode
4. Objekterzeugung mit Konstruktor

7.8.1 Direkter Zugriff auf die Eigenschaften

Im ersten Beispiel sollt Ihr die Klasse *Auto* schreiben, die vier Eigenschaften und eine Methode enthält. Bild 30 zeigt das entsprechende Klassendiagramm.

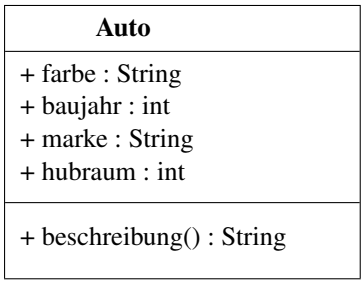

Abbildung 30: Klassendiagramm

Um diese Klasse zu testen, muss man wie oben bereits erklärt, noch eine weitere Klasse, die Klasse *AutoTest* schreiben.
AutoTest soll nur aus der Methode `main` bestehen, weshalb ein Klassen-Diagramm hier wenig aussagt. Als Hilfe gibt es stattdessen ein Code-Gerüst (Abb. 31).

```
class AutoTest {
   public static void main(
               String[] kzp){
                    ...
   }
}
```

Abbildung 31: Code-Gerüst der *AutoTest*-Klasse

Da in der Klasse *Auto* zwar Eigenschaften definiert werden, diese Eigenschaften zunächst aber noch keine Werte haben, muss man den Eigenschaften in der Klasse *AutoTest* konkrete Werte zuweisen. Das soll an dieser Stelle noch mit der Punkt-Notation, also dem direkten Zugriff auf die Eigenschaften geschehen.

D.h. Ihr müsst ein Auto-Objekt erzeugen und es dann färben, mit Hubraum versehen usw.

Hier ein Beispiel: Angenommen man hat das Auto-Objekt `fiat` gebildet, dann kann man den Fiat so lackieren:

```
fiat.farbe = "rot";
```

Das funktioniert aber nur, wenn die Eigenschaften der Autos von aussen überhaupt sichtbar sind!

Aus diesem Grund werden alle Eigenschaften der Klasse *Auto* im Klassen-Diagramm mit einem vorangestellten **Pluszeichen** geschrieben. Dieses Pluszeichen wird im Java-Quelltext durch das Schlüsselwort `public` ersetzt. Eine mit `public` deklarierte Eigenschaft ist auch von ausserhalb ihrer Klasse sichtbar. Und genau das wollten wir ja erreichen.

Das gleiche gilt auch für die Methode `beschreibung()`, die wir auch von einer anderen Klasse aus aufrufen.

Nun ist es aber leider so, dass das direkte Setzen von Eigenschaften ein *sehr schlechter* Java-Programmierstil ist! Im nächsten Beispiel werden wir es besser machen.

Aufgaben zum ersten Beispiel:

1. Schreibt die Klasse *Auto* und übersetzt sie. Die Methode `beschreibung()` soll einen beliebigen String zurückgeben, der die Werte der Eigenschaften enthält.
2. Vervollständigt die Klasse *AutoTest*:
   - Erzeugt ein *Auto*-Objekt!
   - Weist den Eigenschaften dieses Objekts Werte zu! Um auf die Eigenschaften des Objekts zugreifen zu können, muss man die Punkt-Notation verwenden.
   - Gebt den `String`, den die Methode `beschreibung()` zurückliefert mit der Methode `println` auf dem Bildschirm aus!
   - Startet die Klasse *AutoTest*!

7.8.2 Klasse Auto mit Set-Methoden

Im zweiten Beispiel haben alle Eigenschaften im Klassen-Diagramm nun ein vorangestelltes **Minus**-Zeichen. Das Minus steht für `private`. Eigenschaften, die als `private` deklariert sind, sind von ausserhalb der Klasse unsichtbar und man kann deshalb auch nicht auf sie zugreifen.

Um sie dennoch verändern zu können, gibt es zu jeder Eigenschaft eine **set**-Methode. Die set-Methoden sind alle `public`, können also von ausserhalb der Klasse aufgerufen werden. Jede dieser set-Methoden enthält einen Parameter, dem dann beim Aufruf der Methode die entsprechende Eigenschaft zugewiesen wird. Als Name für diesen Parameter kann man z.B. den entsprechenden Eigenschaftsnamen verwenden, dem man die Endung `Par` anhängt: `farbePar`, `baujahrPar`, ...

Aufgaben zum zweiten Beispiel:

1. Schreibt die vier benötigten set-Methoden der Klasse *Auto*!

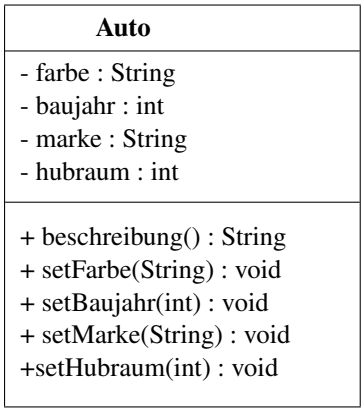

Abbildung 32: Klassendiagramm

2. Passt die Klasse *AutoTest* an! Statt die member-Variablen direkt zu verändern, muss man jetzt die set-Methoden aufrufen.

3. Testet die Klassen!

7.8.3 Klasse Auto mit einer kombinierten Set-Methode

Abbildung 33 zeigt das Klassen-Diagramm der Auto-Klasse, bei dem alle vier Eigenschaften mit *einer* set-Methode gesetzt werden. Diese set-Methode `setAlleEigenschaften(...)` ist noch nicht vollständig dargestellt: es fehlen noch die Parameter, die Ihr selbst festlegen sollt.

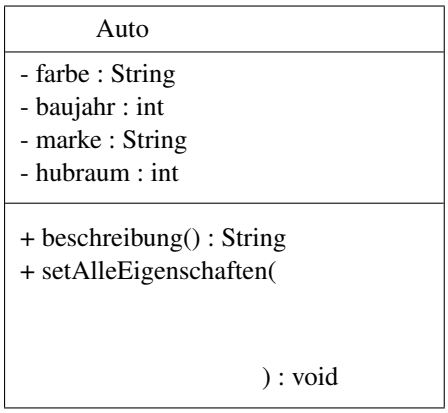

Abbildung 33: Klassendiagramm mit einer set-Methode

Aufgaben zum dritten Beispiel:

1. Wieviele Parameter muss man `setAlleEigenschaften` übergeben und welche Datentypen haben diese jeweils?

2. Tragt die Parameter in das Klassen-Diagramm ein!

3. Ändert die Klasse *Auto*: Ersetzt die vier einzelnen set-Methoden durch `setAlleEigenschaften`!

4. Nach dem Anpassen von *AutoTest*, sollen wieder alle Klassen getestet werden.

7.8.4 Klasse *Auto* mit Konstruktor

Der krönende Abschluss dieser Übung ist schliesslich die *Auto*-Klasse mit Konstruktor.

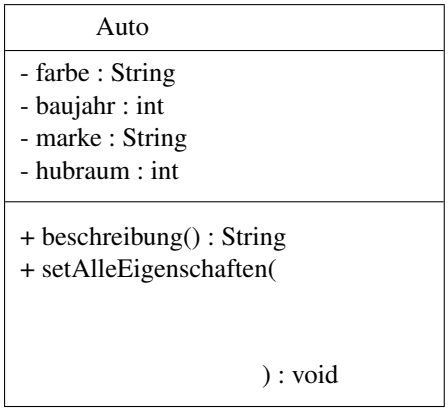

Abbildung 34: Klassendiagramm mit Konstruktor

Aufgaben zum vierten Beispiel:

1. Vervollständigt das Klassendiagramm!

2. Schreibt den Konstruktor in Java, führt evtl. nötige Anpassungen an *AutoTest* duch und testet alles!

3. Wann wird der Konstruktor aufgerufen? Wann kann ich eine gewöhnliche Methode aufrufen?

Literatur

[1] Helmut Balzert. *Lehrbuch Grundlagen der Informatik*. Spektrum Akademischer Verlag GmbH, Heidelberg . Berlin, 1999.

[2] Bruce Eckels. *Thinking in Java*. Prentice Hall 1998. Diverse Postscript und html-Versionen sind kostenlos im Internet erhältlich.

[3] David Flanagan. *JAVA in a Nutshell, deutsche Ausgabe*. O´Reilly Verlag, Köln 1998.

[4] Kalle Dallheimer. *Jetzt mach ich´s selber. Ein Programmierkurs für Einsteiger*. Zeitschrift c´t, 1999 Heft 11 bis 16. Heise Verlag, Hannover, 1999.

[5] Hubert Partl. *JAVA eine Einführung*. Postscriptversion ftp://ftp.boku.ac.at/www/javakurs.ps, html-Version http://www.boku.ac.at/javaeinf/ .

[6] R. Rössler, G. Hartmann. *Grundlagen der Objektorientierten Programmierung*. URL: //parallel.fh-bielefeld.de/pv/vorlesung /sp/begleitmaterial/Java.html .

A Online Dokumentation zu Java

Auf der Java-Seite von SUN befindet sich eine Riesenmenge an Online-Dokumentation im html-Format. Diese ist in zwei Pakete aufgeteilt. Leider sind die Pakete riesig (entpackt 107MByte/38MByte), so dass man schon eine sehr schnelle Internetanbindung braucht, um vernünftige Übertragungszeiten zu erhalten.

SUN erlaubt es meines Wissens nach nicht, dass die Dateien auf CDROMs veröffentlicht werden, aber gelegentlich verirren sie sich doch auf eine CD...

A.1 Dokumentation zur Java 2 Standard Edition

Der wichtigste Teil dieses Pakets ist die Beschreibung der Java API (Application Programming Interface). Sie enthält einen alfabetischen Index aller Java Standard Klassen, Methoden und Konstanten und eine ausführliche Beschreibung *aller* Klassen.

Gerade diese Klassendokumentation ist beim Entwickeln von Java-Programmen eine grosse Hilfe. Beim Schreiben eines Programms kann man die API-Dokumentation in einem eigenen Fenster geöffnet halten und hat dann ein bequemes Nachschlagewerk aller Java-Standard Klassen.

A.2 Java Tutorial

Dieses Paket enthält, ebenfalls im html-Format, eine Menge an Einführungen, die fast alles abdecken, was mit Java zu tun hat. Sehr interessant ist zum Beispiel die Lektion über Objekte und Klassen: Eigentlich hätte ich mir diesen Text hier sparen können...

B Java-Beispiele

```
class Primzahl{
    public static void main(String[] kzp){
        int obergrenze;
        boolean marke = true;
        // obergrenze = Integer.parseInt(kzp[0]);
        obergrenze = 100;

        for(int i=3; i<obergrenze; i=i+2){ //aeussere schleife, alle
                                           //ungeraden zahlen

            for(int j=3; j<=Math.sqrt(i);j=j+2){  //innere schleife, nur
                                                  //ungerade zahlen, laeuft
                                                  //bis zur wurzel von i

                if ((i%j)==0){       // i durch j teilbar?
                    marke = false;   //falls teilbar, marke = false setzen
                    break;           //schleife vorzeitig abbrechen
                }

            }
            if(marke){  //i ausgeben, wenn marke=true
                System.out.println(i + " ist eine Primzahl");
            }
            marke=true;
        }
    }
}
```

Abbildung 35: Ein Programm zur Suche von Primzahlen