Strukturiert programmieren mit Java

(Grundkonzepte)

`www.programmieraufgaben.ch`

Philipp Gressly Freimann 24. Nov. 2016 (Version 1.2.12 (TeX))

Impressum

24. Nov. 2016

1.2.12 (TeX)

http://www.programmieraufgaben.ch

Herausgeber: Philipp Gressly Freimann

Grafik: Philipp Gressly Freimann

Editor: GNU Emacs 23.3.1

Publisher: LaTeX: pdfTeX 3.1415926-1.40.10-2.2

Inhaltsverzeichnis

Präambel Eine geschlechtsneutrale Bezeichnung von Personen oder eine Bezeichnung verschiedener Geschlechter wurde weitgehend vermieden, um die Lesbarkeit des vorliegenden Dokuments zu erleichtern. Alle Leser sind selbstverständlich gleichermaßen angesprochen.

Vorwort

Das Buch «Programmieren lernen» [GFG11] (S. Literaturliste auf Seite 197) ist weitgehend unabhängig von einer Programmiersprache. Glauben Sie jedoch nicht, dass Sie ohne eine Programmiersprache des Programmierens mächtig werden; Sie lernen auch nicht Auto fahren, indem Sie Fahrstunden ohne Wagen absolvieren; welche Automarke bzw. welche Programmiersprache Sie dazu auswählen ist jedoch sekundär.

Damit die Übungen aus [GFG11] fürs JAVA-Selbststudium durchführbar sind, ist eine minimale Einführung in JAVA nötig. Dies wird hiermit zur Verfügung gestellt.

Diese JAVA-Einführung zeigt diejenigen Syntaxelemente, welche es braucht, um einfache strukturierte Programme zu schreiben. Damit sollten die meisten Aufgaben aus
`http://www.programmieraufgaben.ch` lösbar sein.

Alles wird neu Die Grundkonzepte der Programmierung sind für Lehrlinge wie auch für alte Hasen gleichzeitig unerlässlich. Da ich sowohl Anfänger wie auch Profis unterrichte, behaupte ich folgendes: Je besser die Grundlagen sitzen, umso eher ist man in der Lage, sich auf Neues einzustellen. Wer vorwiegend mit «copy & paste» programmiert[1] und sich dann zufrieden gibt, wenn «es läuft» und dabei nicht versucht, zu verstehen was wirklich passiert, hat wenig Chancen weiterzukommen. Schlimmer: Solche Programmierer werden früher oder später von einer neuen Technik überholt und ausrangiert. Dieses Buch richtet sich an Anfänger, die das Kunsthandwerk der Computerprogrammierung von Grund auf lernen wollen, ebenso wie an langjährige Quereinsteiger, die nie die Gelegenheit hatten, sich mit den Grundlagen moderner Programmiersprachen auseinander zu setzen und gleichzeitig interessiert sind, eine für sie neue Sprache (hier JAVA) kennen zu lernen.

Ich kann mit meinen gut 30 Jahren Programmiererfahrung[2] wohl noch nicht behaupten, dass ich zu den wirklich alten Hasen gehöre. Etwas habe ich dennoch erfahren: Programmiersprachen kommen und gehen. Von Assembler bis hin zu Python und Grails habe ich gesehen, dass alles im Wandel ist. Doch etwas Wichtiges habe ich auch gelernt: **Die Grundkonzepte von Programmiersprachen haben sich nicht verändert!** Auch heute denke ich oft noch in Anweisungen, Variablen (Attributen und Entitäten), Selektionen, Iterationen, Methoden und Schnittstellen. Die Sprachen der 3. Generationen (BASIC, C, JAVA) wurden immer mächtiger[3], die Fehlerquellen minimiert, die Entwicklerwerkzeuge immer hilfreicher, die Fehlersuche immer ausgeklügelter; doch die Grundlagen blieben dieselben. Ich versuche mit diesem Buch dem Leser diejenigen Konzepte beizubringen, von denen ich überzeugt bin, dass sie noch einmal 40 Jahre überdauern werden.

[1] Mit «copy & paste» zu arbeiten, ist im Übrigen nicht programmieren.

[2] Die ersten Programme schrieb ich 1982.

[3] Einige Sprachen sind hier ausgenommen. Sprachen höherer Generationen wie SQL und PROLOG haben ihre eigenen speziellen Konzepte. Dennoch bin ich überzeugt, dass die Grundlagen von 3.-Generationsprachen unerlässlich sind, um das Handwerk der Computerprogrammierung wirklich zu verstehen.

Über die Wichtigkeit von Grundlagen Es gibt in meinen Augen nichts Wichtigeres in einem Ingenieurberuf, als die Grundlagen wirklich zu beherrschen. Goethe sagte schon in seinem Gedicht «Katzenpastete»:

Bewährt den Forscher der Natur
Ein frei und ruhig Schauen
So folge Messkunst seiner Spur
Mit Vorsicht und Vertrauen...

So kann ich meinen Vorsprecher nur unterstützen in der Aussage, indem ich hier den ersten Insider Tipp abgebe.

Geek Tipp 1

Wenn Du nicht wirklich verstanden hast, warum ein Programm läuft, dann läuft es nicht[a]!

[a]Warum sollte es dann nicht laufen? Natürlich läuft es; für einen Moment. Oft aber nur solange wir als Programmierer gerade dabei sind, dieses Programm zu testen. Sobald unser Programm in eine andere Umgebung kommt, so ist ein Versagen gut möglich. Wenn wir jetzt versuchen, das Programm zu reparieren, oder noch schlimmer, jemand anders versucht es zu verstehen und danach zum Laufen zu bringen, treten die wirklichen Probleme auf: Das Programm ist nicht mehr wartbar!

In obigem Sinne möchte ich den Leser anweisen, mindestens zu versuchen, den ersten Teil dieses Skripts (bis und mit Kapitel 7 «Zeichenketten» auf Seite 91) wirklich zu verstehen. Natürlich sind wir nach dem Erlernen der Grundlagen noch lange keine guten Programmierer. Wir haben jedoch das beste Werkzeug in der Hand, alle darauf aufbauenden Konzepte schneller zu erfassen und mit sicherem Gefühl anwenden zu können. Sollte später etwas nicht so funktionieren, wie wir das erhofft haben, so haben wir immer noch die Grundlagenwerkzeuge, die uns (meist) ermöglichen, das entstandene Problem selbst zu lösen, oder aber gekonnt zu umgehen.

Und was heißt bitte schön «objektorientiert»?

Strukturiert programmieren mit JAVA? Geht das überhaupt? JAVA sei doch eine objektorientierte Sprache? Ja, auch für Dich, der schon viel mehr weiß, habe ich einen Antwortversuch bereit:

Nun, formulieren wir es einmal so. Mit einer objektorientierten Sprache wie JAVA kann man unten anderem:

- test-orientiert programmieren
- service-orientiert programmieren
- objektorientiert programmieren
- strukturiert programmieren
- unstrukturiert programmieren
- fehleranfällig programmieren
- überhaupt nicht programmieren

Die Art des Vorgehens hat nichts mit der Sprache zu tun. Auch wenn es in Assembler oder `C` viel schwieriger ist, objektorientiert vorzugehen, so ist dies sehr wohl möglich. Jede objektorientierte Sprache ist aber auch eine prozedurale Sprache und somit sind alle Konzepte, welche für die strukturierte Programmierung nötig sind in JAVA auch vorhanden. Genau genommen sind mittels `return`, `break` und `continue` sogar unstrukturierte Vorgehensweisen in JAVA zugelassen[4].

Ob es nicht sinnvoller wäre, in JAVA direkt objektorientiert mit der Kunst des Programmierens einzusteigen, ist eine didaktische Grundsatzdiskussion. Es geht beides. Begonnen beim «Bit», wie in diesem Buch, oder direkt begonnen bei «Fenstern», «Knöpfen» und der «Ereignisverarbeitung».

Wer schnell ein Ergebnis auf dem Schirm sehen will, ist mit diesem Lehrmittel wohl nicht gut bedient. Wer jedoch möglichst alle Grundlagen kennen lernen will, fährt besser mit der in diesem Buch vorgeschlagenen «bottom up»-Methode; also derjenigen Vorgehensweise, bei der man vom innersten Kern einer Programmiersprache (Datenstrukturen, Variable, Sequenzen, Ein- und Ausgabe, ...) sich langsam in die große Welt der Computerprogrammierung hineinarbeitet.

Ein Gerüst, um in JAVA weitgehend strukturiert zu programmieren findet sich im Anhang (s. Kap. A.4 auf Seite 135).

Aufruf Hier noch ein Aufruf an Programmierer anderer Sprachen: Bitte stellt uns solche Einführungen auch für andere Sprachen zur Verfügung. Python ist der nächste Kandidat. `C`, Ruby und VisualBASIC werden auch oft gewünscht. Wer sich an einer «Übersetzung» beteiligen will, soll sich doch bitte mit mir in Verbindung setzen: Philipp Gressly Freimann (`phi@gressly.ch`).

[4] Es sind sogar mittels `else-if` innerhalb einer Endlosschleife `while` mit einer «jump»-Variable beliebige Sprünge denkbar. Somit braucht es gar keine Spezialbehandlungen, um mit einer «strukturierten»-Sprache eben «nicht strukturiert» zu programmieren.

Dank

Ein spezieller Dank an Rémy Gressly, der sich die Mühe gemacht hat, sehr viele Tippfehler aufzuspüren.

Ein weiteres Dankeschön gilt Bruno Keller, welcher die Vorabversion dieses Dokumentes mit einer Schulklasse durchgearbeitet hatte. (Entschuldige Bruno, dass Du in einer weiteren Durchführung die Seitennummern und einige Kapitel neu verweisen musst.)

Ein erstes Java-Programm

Bevor wir mit den Themen (also den Kapiteln aus [GFG11]) beginnen, möchte ich alle Leser ermuntern, ein erstes JAVA-Programm zu schreiben[5]:

```
public class MeinErstes {
  public static void main(String[] args) {
    new MeinErstes().top();
  }

  void top() {
    System.out.println("Hallo␣Welt");
  }
}
```

Dieses Programm wird

1. In einem Texteditor geschrieben und als `MeinErstes.java` gespeichert;
2. mit dem JAVA-Compiler kompiliert `javac MeinErstes.java`;
   es entsteht die Datei `MeinErstes.class`;
3. und zuletzt mit der JAVA-virtuellen Maschine (JVM) ausgeführt:
   `java MeinErstes`

Eine detaillierte Installationsanleitung und Tipps zur Fehlersuche finden Sie im Anhang (s. Kap. A.1 auf Seite 127). Ein generisches strukturiertes JAVA-Programm ist im Kapitel über Abfolgen zu finden (s. Kap. A.4 auf Seite 136).

Bemerkung

Die Anweisung `System.out.println(...)` bezeichnet in JAVA die Ausgabe einer Textzeile auf der Konsole.

[5] «Didaktischen Hinweis für Lehrer» Es ist natürlich genauso gut möglich, auf die schlecht erklärbare Instanzierung `new MeinErstes().top();` zu verzichten und danach bei jeder Subroutine und bei jeder globalen Variable das JAVA-Schlüsselwort `static` anzufügen. Dies kann im Unterricht jedoch zu Fragen führen, die noch nicht beantwortet werden können, denn das Gegenteil (nämlich der *dynamische* Code) wird – aus Zeitgründen – ja allenfalls gar nie erklärt werden können.

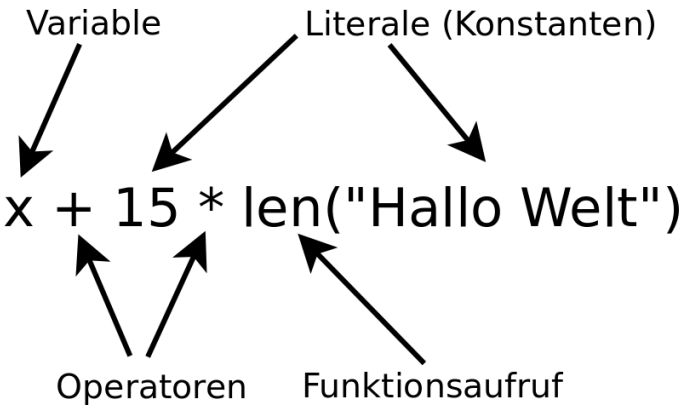
Egal, wie wir es in JAVA drehen und wenden; in JAVA müssen wir uns immer auf die eine oder andere Weise mit Objekten beschäftigen.

Wenn wir nun den Lernenden einfach sagen, dass obiger Starter (oder aber das Schlüsselwort `static` vor jeder Funktion und vor jeder globalen Variable) einfach immer dazu gehört, so haben wir kritische Fragen nur von denjenigen Kursteilnehmern, welche sich sowieso schon vertieft um die Materie gekümmert hatten, und für diese kleine Gruppe ist eine Erklärung dann rasch gegeben.

1 | Ausdrücke und Datentypen

1.1 Ausdrücke und Operatoren

Beginnen wir doch gleich mit demselben **Ausdruck** wie in [GFG11]. Nur, dass wir diesmal den JAVA-Code angeben[6]:




```
x + 15 * len("Hallo␣Welt")
```

Ein zusammengesetzter Ausdruck besteht stets aus einfacheren Ausdrücken. Die Grundbausteine (Literal, Variable, Funktionsaufruf und Operator) werden ab der nächsten Seite erklärt.

[6]Im Beispiel ist die Funktion `len()` keine Standardfunktion von JAVA! Dieses Beispiel zeigt lediglich den Aufbau eines Ausdrucks.

1.1.1 Ausdrücke

Ausdrücke werden in vier Kategorien eingeteilt. Im Wesentlichen kann man es sich folgendermaßen merken. Ein «Ausdruck» ist etwas, was einen Wert hat. Werte werden fast ausschließlich einer Variable zugewiesen: Entweder geschieht dies mit einer Zuweisung («=» in JAVA) oder als Argument in einem Subroutinen-Aufruf[7].

Ausdrücke sind

- **Literale**[8] Zahlen, Buchstaben und Zeichenketten werden als Konstante (bzw. konstante Werte) direkt in den Code geschrieben.
  Beispiele: `5`, `"Hallo"`, `'c'`, `-4.85`
- **Variable** bezeichnen Speicherstellen. An ihrer Stelle können (binär kodierte) Werte stehen. Jede Variable hat einen Namen (Bezeichner), einen Datentypen und allenfalls einen Wert.
  Beispiele: `x`, `chfPer31Dez`
- **Funktionsresultat** Jedes Unterprogramm, das einen Wert zurück liefert, kann als Ausdruck angesehen werden.
  Beispiele: `random()`, `Math.abs(-3.75)`, `Math.sin(x)`
- **Zusammengesetzte Ausdrücke** Ebenso können Ausdrücke mit speziellen **Operatoren** zu neuen Ausdrücken zusammengesetzt werden.
  So ist beispielsweise `5 + x + random()` ein Ausdruck bestehend aus einem Literal (`5`), einer Variable (`x`) und einem Funktionsresultat (`random()`). Solche zusammengesetzten Ausdrücke werden wie ein Literal, eine Variable oder ein Funktionsresultat verwendet.
  Beispiele: `9 * b`, `3*(sin(45) + 44.0) - 8*(r/5.6)`, ...

Aufgabe: Finden Sie alle zehn (Teil)ausdrücke im folgenden Ausdruck:

```
w + 3.2 * (x + 2 * sin(y))
```

Lösung (s. Kap. B auf Seite 189).

[7] Dieses Argument wird dabei dem entsprechenden Subroutinen-Parameter (also auch wieder einer Variable) übergeben.

[8] Literale werden auch als Konstanten bezeichnet. Ebenfalls als Konstanten bezeichnet werden aber auch Variable, deren Werte nicht verändert werden dürfen; daher verwende ich lieber den Namen «Literal», auch wenn der Name nicht besonders üblich ist.

1.1.2 Operatoren

Einige Operatoren werden in JAVA mit den üblichen mathematischen Symbolen dargestellt. Multiplikation verwendet hingegen einen Stern (`*`) und die Division eine Schrägstrich (`/`).

Die wichtigsten Operatoren in JAVA sind:

- Addition (`+`)
- Subtraktion (`-`)
- Vorzeichen (`-`)
- Multiplikation (`*`)
- Division (`/`)
- Restbildung (`%`)(s. Kap. 1.1.3 auf Seite 19)

1.1.3 Restbildung (Modulo %)

Die Restbildung wird in JAVA mit dem Zeichen `%` dargestellt. Wie wir alle wissen, geht nicht jede Division «auf». Manchmal interessieren wir uns aber nicht für das Divisionsresultat, sondern lediglich für den Divisionsrest, eben das, was beim ganzzahligen Dividieren übrig bleibt. Die Modulo-Operation liefert diesen «Rest».

In den C-Sprachen (C, C++, JAVA, ...) wird hierzu der %-Operator verwendet:

18 % 7 ergibt 4 (denn 18 durch 7 gibt 2 Rest 4)

27 % 4 ergibt 3 (denn 27 durch 4 gibt 6 Rest 3)

1.2 Unär, binär, ternär

Operatoren wirken nicht immer auf zwei Operanden. Einige (auch unäre oder einargumentige Operatoren genannt) wirken auf einen einzigen Operanden. Der klassische Vertreter ist das Vorzeichen. In JAVA gibt es auch noch einen ternären (dreiargumentigen) Operator, der jedoch selten Verwendung findet.

Typ	Beschreibung	Beispiel
unär	Einargumentig	Vorzeichen (`-`), aber auch das NOT (`!`) und die Bitweise Umkehrung(`~`)
binär	Zweiargumentig, wie die meisten	Summe `+` Produkt `*`
ternär	Dreiargumentig	`a < b ? "kleiner ": "groesser "`

1.3 Vorrangregeln

JAVA-Ausdrücke werden in der Regel von links nach rechts geklammert. So wird folgender Ausdruck ...

```
a + b - c - 4.5 + a
```

... wie folgt geklammert:

```
((((a + b) - c) - 4.5) + a)
```

Beim Auswerten von Ausdrücken gilt in auch in JAVA die «Punkt-Vor-Strich»-Regel. Operatoren 2. Stufe (Level) werden zuerst geklammert:

```
a * b - c - 4.5 * a
```

Obiger Ausdruck wird wie folgt geklammert:

```
((a * b) - c) - (4.5 * a)
```

Eine vollständige Liste der Operatoren und deren Klammerung in JAVA finden wir hier:
`http://www.santis-training.ch/java/javasyntax/operatoren.php`

1.4 Datentypen und ihre Literale

Die wichtigsten fünf Datentypen in JAVA sind die folgenden:

```
boolean : Wahrheitswert : true / false
int     : integer = ganz; ganze Zahl (positiv, negativ)
double  : Dezimalbruch mit doppelter Genauigkeit (64-Bit)
char    : Einzelzeichen (Character)
String  : Zeichenkette
```

Bemerkung 1.1. Geroge Boole (1815-1864) begründete die nach ihm benannte Boole'sche Algebra, die nur mit zwei Zuständen (0 = false, 1 = true) auskommt, daher der Name `boolean`.

Eine Zusammenstellung aller Datentypen in JAVA findet sich im Anhang (s. Kap. A.2 auf Seite 132).

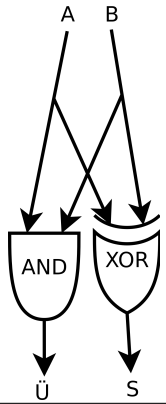
1.4.1 Ganze Zahlen

Zahlen werden in JAVA im Binärsystem dargestellt. JAVA kennt die Ganzzahltypen `byte`, `short`, `int` und `long`, die sich lediglich in der Anzahl ihrer Bits[9] unterscheiden. Ganzzahltypen können in JAVA positive, wie auch negative ganze Zahlen darstellen. Wegen den negativen Zahlen (Vorzeichen `-`) nennen wir diese Datentypen auch «*(mit)* Vorzeichen behaftet».

1.4.2 Der Halbaddierer

Nun können wir uns fragen, wie denn ein Computer (Rechner) mit den Zahlen rechnen kann. Am Beispiel des Halbaddierers soll gezeigt werden, wie dies funktioniert.

Ein Halbaddierer ist eine grundlegende elektronische Schaltung, die zwei Bit zusammenzählen kann. Dabei werden Stromspannungen als Bit repräsentiert (z. B. 0 Volt = 0-Bit und 5 Volt = 1-Bit) und mit einem elektronischen AND (und) und einem elektronischen XOR (exklusives oder) miteinander verknüpft:



A	B	Übertrag	Summe
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Ein Halbaddierer kann einfach mit Relais oder Transistoren gebaut werden. Eine Nachbildung befindet sich im SANTIS Museum of Computing History (SMOCH):
`http://smoch.santis-basis.ch/index.php/halbaddierer`

[9] Bit = Binary Digit = Ziffer im Zweiersystem

1.4.3 Gebrochene Zahlen (metrische Daten)

Gebrochene Zahlen (Zahlen mit Nachkommastellen oder Brüche) werden in der Praxis an verschiedensten Orten angetroffen. Meist handelt es sich dabei um **Messgrößen** (z. B. Volumen [Liter], Längen [Meter, mm], Temperatur [Celsius], ...). In JAVA ist der Datentyp einer solchen Variable mit `double` (oder `float`) festgelegt. Beachten Sie insbesondere, dass es sich in JAVA bei nicht abbrechenden Dezimalzahlen immer um eine Annäherung handelt. So ist es also nicht möglich die Zahl Pi, oder 1/7 damit exakt darzustellen. Mehr noch: nicht einmal die Zahl 0.1 ist damit genau darstellbar, da die Zahlen im Binärsystem abgebildet werden. Dabei ist eben 0.1 (also ein Zehntel) ein nicht abbrechender «Binärbruch».

Wissenschaftliche Notation Erschrecken Sie nicht, wenn Zahlen wie folgt daher kommen sollten:

- 5.7e6
- 4.35353535354e-2
- -3.66666666667e-20

Keine Angst: das Zeichen «e» bedeutet hier nichts esoterisches. Diese Schreibweise wird auch als **Exponentialschreibweise** bezeichnet (daher wohl das «e»). Dies sagt einfach aus, um wie viele Stellen der Dezimalpunkt nach rechts verschoben werden soll (bzw. links, falls die Zahl nach dem «e» negativ ist). Somit sind die obigen Zahlen wie folgt zu lesen:

- $5.7e6 = 5\,700\,000 = 5.7 \cdot 10^{6}$
- $4.35353535354e\text{-}2 = 0.0435353535354 = 4.353... \cdot 10^{-2}$
- $-3.66666666667e\text{-}20 = -0.00000000000000000003666666666667 = -3.66... \cdot 10^{-20}$

Typumwandlung (Casting) In typisierten Sprachen wie JAVA muss oft zwischen gebrochenen und ganzen Zahlen *umgeschaltet* werden. Aus einer ganzen Zahl eine gebrochene zu erhalten, ist in allen mir bekannten Sprachen trivial: Dies geschieht automatisch. Die Umwandlung **in die andere Richtung** bedarf meist einer Zustimmung durch den Programmierer (Casting = explizite Typumwandlung), da durch die Umwandlung in der Regle Information verloren geht. Was geht verloren? Allfällige Nachkommastellen.

Beispiel:

```
int    i;
double d;

// Diese Umwandlung ist trivial:
i = 4;
d = i;

// Dies ist nur mit Casting erlaubt:
d = 3.1104 ;
i = (int) d; // i hat nun der Wert 3
```

1.4.4 Beliebig große, bzw. beliebig genaue Zahlen

In JAVA existieren zwei weitere Datentypen, welche mit beliebiger Genauigkeit rechnen können:

`BigInteger`	Ein `BigInteger` kann beliebig große ganze Zahlen darstellen. Dabei sind 300stellige Zahlen keine Seltenheit, aber auch kein Problem für JAVA.
`BigDecimal`	Dagegen kann ein `BigDecimal` mit beliebig vielen Nachkommastellen umgehen. Die Zahl π auf tausend Nachkommastellen zu berechnen ist mit JAVA also gut möglich.

Im folgenden Beispiel wird die Zahl 10 durch 1.234 geteilt und 30 Nachkommastellen werden ausgegeben:

```
BigDecimal bd = BigDecimal.TEN;
bd = bd.setScale(30);
bd = bd.divide(BigDecimal.valueOf(1.234), BigDecimal.ROUND_UP);
System.out.println(bd.toPlainString());
```

1.4.5 Funktionsresultate

Funktionsresultate können in JAVA auch als Ausdrücke angesehen werden und werden wie Variable und Konstanten (Literale) verwendet:

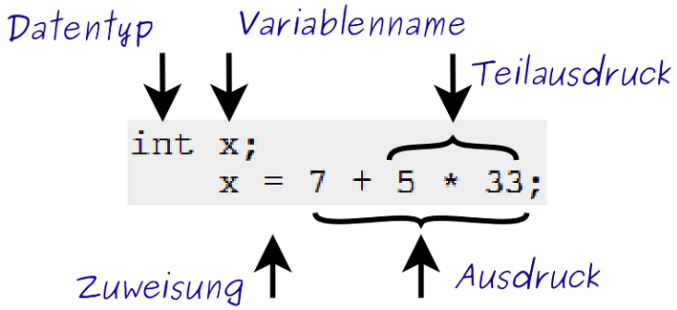
```
double y;
y = Math.sin(x) + 2 * Math.cos(Math.ln(x) + phi * Math.min(a, b));
```

1.5 Variable

Es gibt im Umgang mit Variablen grundsätzlich zwei Arten von Programmiersprachen. Bei typisierten Sprachen (C, JAVA, ...) können einer Variable nur Werte zugewiesen werden, die in einem vorgegebenen Wertebereich (z. B. «ganze Zahl») liegen. Bei untypisierten Sprachen (PHP, JavaScript, ...) kann einer Variable jeder mögliche Wert der Programmiersprache zugewiesen werden.

Beispiel in JAVA:

```
int x;
x = 7 + 5 * 33;
```



JAVA ist eine typisierte Sprache. Die Variablen müssen daher vor dem ersten Gebrauch deklariert werden. Deklarieren heißt hier: Wir geben dem Programm an, um welchen Datentyp es sich bei der Variable handelt. Jeder Variable liegt also ein Datentyp zugrunde. Dies ist vorteilhaft damit die Programmierer nicht versehentlich falsche Werte in die Variable einfüllen.

In JAVA wird eine Variable deklariert, indem zunächst der Datentyp angegeben wird, und danach folgt der Name der Variable.

Beispiele:

```
boolean eingabeKorrekt    ;
int     anzahlTreffer     ;
double  blutMengeInLitern;
char    steuerzeichen     ;
String  name              ;
```

In JAVA können pro Deklaration mehrere Variable angegeben werden, und Variable können in der Deklarationsanweisung auch einen Initialwert erhalten.

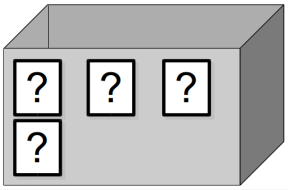
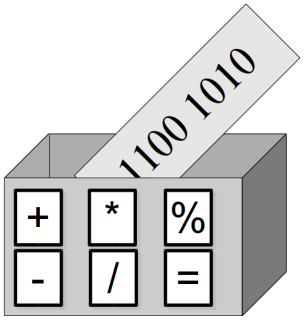
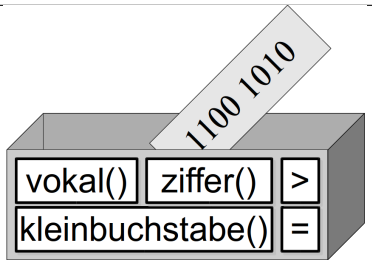
Beispiele:

```
int    x, y, z;

double abstandInMetern = 4.88   ;
String name            = "Meier";
```

1.5.1 Beispiel `short` vs. `char`

Die folgende Tabelle veranschaulicht den Zusammenhang von Variablen und ihren Datentypen:

	Eine Variable kann man sich als Kiste vorstellen. In diese Kiste werden die Werte der Variable gelegt. Die Variable hat einen Datentypen. Wir können uns eine Variable als Fernsteuerung für die Inhalte vorstellen.
 `s: short` (202)	In unserem Beispiel wird das Bitmuster «1100 1010» in die Variable `s` (vom Datentypen `short`) gespeichert. Die Variable können wir nun addieren, subtrahieren, vergleichen etc. Als ganze Zahl (`short`) hat diese Variable den Wert 202. Ich habe den Datentypen `short` gewählt, weil er aus 16 Bit besteht. Dies sind genau so viele Bit, wie auch in eine Variable vom Datentypen `char` passen. Beachten Sie, dass unser Bitmuster nur 8 Bit benötigt. Die vorangehenden 8 Bit werden von JAVA automatisch mit Nullen «0000 0000» aufgefüllt.
 `ch: char` (202 = Ê)	Betrachten wir nun die Variable `ch` (vom Datentypen `char`), so kann dasselbe Bitmuster eine ganz andere Bedeutung haben. Wir können nun die Variable prüfen (`istZiffer()`, `istVokal()`), aber auch verändern (z. B. `kleinBuchstabe()`). Als Unicode Zeichen hat das Bitmuster «1100 1010» den Wert Ê.

1.5.2 Bezeichner (identifier)

Variable werden mit sogenannten Bezeichnern referenziert. Bezeichner (en. identifier) sind Namen. Diese Namen sind beinahe beliebig, müssen sich in JAVA jedoch an folgende Regeln halten:

- Bezeichner dürfen keine reservierten Wörter sein, z. B. `double`, `class` oder `while` (s. Kap. A.5 auf Seite 137).
- Bezeichner dürfen Buchstaben, Ziffern, Währungssymbole und den Unterstrich (`_`) enthalten.
- Bezeichner dürfen nicht mit einer Ziffer beginnen.

Gut sind Bezeichner wie:	`x`, `aktuelleZeit`, `width`, `TAUSEND` ...
Verwendbar, wenn auch unglücklich:	`$_1`, `länge`, ...
Ungültig sind:	`1x`, `const`, `s#`, ...

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 1. 2 Berechnung von Ausdrücken (1)
- 1. 3 Berechnung von Ausdrücken (2)

2 | Sequenzen (Anweisungen und Abfolgen)

Ein einfaches Computerprogramm besteht aus einer Abfolge (= Sequenz oder Aneinanderreihung) von Anweisungen (Befehlen). Nach dem Starten des Programms wird die Sequenz in Leserichtung (also der Reihe nach von oben nach unten) verarbeitet.

Selbst in objektorientierten Sprachen muss der Prozessor stets wissen, welches die nächste auszuführende Aktion ist. Somit ist auch in JAVA der Kontrollfluss ein elementares Konzept. Der Kontrollfluss wird in JAVA mittels Sequenzen, Selektionen, Schleifen, Methoden, Events, Exceptions und Multithreading gesteuert.

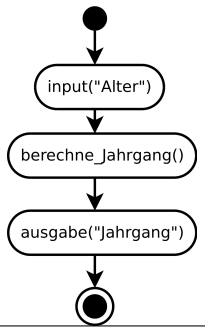
So viel aufs Mal? Beginnen wir mit einfachen Abfolgen, auch Abläufe oder Sequenzen genannt.

2.1 Abfolgen (Sequenzen)

JAVA-Anweisungen werden in Leserichtung aufgeschrieben. Erst wenn ein Befehl ganz fertig ausgeführt ist, wird mit der nächsten Anweisung begonnen[10]. Mit anderen Worten: Mit einer Anweisung in einer Sequenz wird erst begonnen, wenn die vorangehende Anweisung **komplett abgeschlossen** ist.

Gegenüberstellung JAVA zu UML[11]:

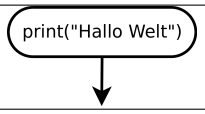
```
{
  eingabe("Alter");
  berechne_Jahrgang();
  ausgabe("Jahrgang");
}
```



Semikolon

Die UML-Notation wurde bereits teilweise in [GFG11] verwendet und eine Zusammenfassung der wichtigsten UML-Elemente findet sich im Anhang des PDF-Buches [GF14]. Hier ein Beispiel einer einzelnen Anweisung in JAVA. Jede Anweisung in JAVA wird mit einem Strichpunkt (Semikolon `;`) beendet:

```
System.out.println("Hallo Welt");
```



Bemerkung 2.1. Die Subroutine `println()` gibt bekanntlich einen Text auf der Konsole aus.

[10] Typischerweise schreibt man pro Zeile eine Anweisung (= Befehl).
[11] UML = Unified Modelling Language

2.2 Anweisungen

Die wichtigsten **vier** Arten von Anweisungen[12] sind:

- **Deklarieren von Variablen**:

```
int    x, y       ;
char   zeichen    ;
String begruessung;
```

- **Zuweisen von Werten an Variable** (s. Kap. 2.5 auf Seite 33):
  - Zuweisen von Literalen (Konstanten):
    `x = 4;`
  - Zuweisen von zusammengesetzten Ausdrücken und Funktionsresultaten (z. B. Aufruf mathematischer Funktionen):
    `begruessung = "Hallo" + "Welt";`
    `y = x + 3 * Math.max(5, x);`
  - Einlesen von Text und Zahlen von der Tastatur:

```
Scanner sc = new Scanner(System.in);
int     x  = sc.nextInt();
```

  - Hilfsfunktionen (Kalender, Ausgabeformat, ...):

```
Calendar cal  = new GregorianCalendar() ;
int      jahr = cal.get(Calendar.YEAR)  ;
```

- **Prozeduraufruf**:
  Ausgabe von Text und Zahlen auf die Konsole:
  `System.out.println("Hallo Welt");`
  Warten auf das Beenden eines anderen Prozesses:
  `threadXY.join();`
  Setzen von (System)eigenschaften:
  `setDate("2015-02-05");`

- **Kontrollflusssteuerung**:
  Selektion (`if` / `else`) (s. Kap. 3 auf Seite 37) und Iteration (`while` / `for`) (s. Kap. 4 auf Seite 53).

[12] Synonyme: Anweisung, Befehl, Statement

2.2.1 Weitere Anweisungen in Java

Neben den soeben erwähnten vier häufigsten Anweisungen der prozeduralen Programmiersprachen (Deklaration, Zuweisung, Prozeduraufruf, Kontrollflusssteuerung) gibt es in JAVA weitere Anweisungen:

- Variableninkrement bzw. -dekrement (z. B. `++i;` , `x--;`)
- Return Anweisung (`return;`)
- Objekterzeugung (z. B. `new JFrame();`)
- `try` - `catch` -Block
- `throw` -Anweisung

2.3 Deklarationen

Die standardmäßige Variablendeklaration in JAVA ist ein Datentyp gefolgt von einem Bezeichner, welcher als Variablennamen dient:

```
int    plz      ;
String eingabe  ;
double masse    ;
```

Bemerkung 2.2. Durch Komma abgetrennt können in JAVA pro Deklaration mehrere Variable angegeben werden:

```
int    plz,      jahrgang      ;
String vorname, familienname  ;
```

Bemerkung 2.3. Es ist gestattet, wie bereits im Kapitel über Variable erwähnt, Variable gleich bei ihrer Deklaration zu definieren, also mit einem Initialwert zu versehen. Die beiden folgenden Code-Blöcke sind identisch:

```
String begruessung;      // <- Deklaration
begruessung = "Hallo";   // <- Definition
```

Abkürzend kann die Deklaration und die Definition in einem Schritt angegeben werden:

```
String begruessung = "Hallo";
```

2.4 Prozeduraufrufe

Bei einem Prozeduraufruf (z. B. `println()`)[13] wird eine andere Stelle im Programm ausgeführt, von wo nach beenden ebendieser Prozedur wieder an dieselbe Stelle im Ablauf zurückgekehrt wird.

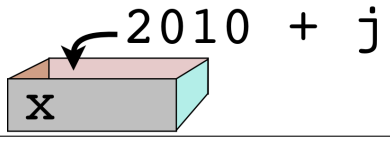
Neben bereits besprochenen vorgegebenen Subroutinen gibt es in JDK[14] 1.7 mehr als 7000 Module mit beinahe 40 000 Hilfsfunktionen. Zum Beispiel existieren dutzende von Kalenderfunktionen, Funktionen für Zufallszahlen und trigonometrische Funktionen, Funktionen für die Computersicherheit und fürs Verschlüsseln und Entpacken von Daten, Funktionen fürs Erstellen von graphischen Benutzerschnittstellen, für den Zugriff aufs Internet und Datenbanken, für die Bearbeitung und Formatierung von Text, Erstellen und Lesen von XML-Dateien, Behandeln von regulären Ausdrücken, und und und.

[13] Subroutinen sind: Prozeduren, Unterprogramme, Funktionen, Methoden (s. Kap. 5 auf Seite 63).
[14] JDK = JAVA Developers Kit

2.5 Zuweisung

Die wohl am häufigsten eingesetzte **Anweisung** ist die **Zuweisung**. Variable erhalten ihre Werte durch Zuweisung via Ausdruck.

JAVA-Beispiel:

`x = 2010 + j;`	

Die Zuweisung wird mathematisch mit einem ':=' oder mit dem Symbol. '←' geschrieben. In JAVA hingegen wird die Zuweisung durch ein einfaches Gleichheitszeichen «=» dargestellt.

Bemerkung: Das einfache Gleichheitszeichen ist keine Gleichheit im mathematischen Sinne. Dabei bedeutet `x = 9*x + 3;` keinesfalls, dass eine Gleichung in der Variable `x` zu lösen ist (Einige Sprachen tun dies und `x` erhält den Wert -0.375)! In JAVA wird hier als erstes der Wert rechts des Gleichheitszeichens berechnet und dieser Wert wird danach in die Variable gespeichert.

Beachten Sie den folgenden Code:

```
a = 4;
b = a;
a = 5;
```

Zuerst erhält `a` den Wert 4. Danach erhält `b` den Wert 4, also den Wert der Variable `a`. Zuletzt wird `a` auf 5 gesetzt. Beachten Sie, dass `b` immer noch den Wert 4 hat, auch wenn die Variable `a` und `b` vermeintlich gleich gesetzt wurden. **Merke**: Die JAVA Zuweisung bedeutet **«Variable wird zu Wert»** und nicht **«Variable wird zu Ausdruck»**.

Zur Ein- und Ausgabe von Werten über die Tastatur siehe Anhang A.6 über die Ein- und Ausgabe auf Seite 138.

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 2. 1 Hello World
- 2. 3 Variablenwert nach einer Sequenz
- 2. 4 Ohm'sche Gesetze
- Satz von Pick (Web-Code: hpaj-dw3x)
- 2. 5 Abbildungsgleichung
- Eisenwarenhändler (Web-Code: ranc-fuh0)
- 2. 6 Runden (1)
- Noten Runden (Web-Code: u3kp-4ffh)
- 2. 7 Runden (2) «schwierig»
- 2. 8 Stunden, Minuten, Sekunden
- 2. 9 Ganzzahlarithmetik (Kommutativgesetz)
- 2.10 Gleitkommaarithmetik (Assoziativgesetz)
- Aufgabe «Quadratische Gleichung» (Web-Code: dk4s-3aeg)

Aufgabe 2.1 « `int` » Hier noch eine kleine Übung zu den JAVA-Datentypen. Programmieren Sie die folgende Sequenz und erklären Sie das Ergebnis:

```
int x;
x = 2_147_483_647;
x = x + 1         ;
System.out.println("x:␣" + x);
```

3 | Selektion (Verzweigung)

3.1 Einführungsbeispiel

Ein Stück Code (hier Schneeketten) soll nur unter gewissen Umständen ausgeführt werden:

```
int temperatur;
temperatur = new Scanner(System.in).nextInt();
System.out.println("Einsteigen und anschnallen!");
if(temperatur < -10)
{
  System.out.println("Schneeketten obligatorisch!");
}
System.out.println("Losfahren!");
```

3.1.1 Die allgemeine `if`-Anweisung

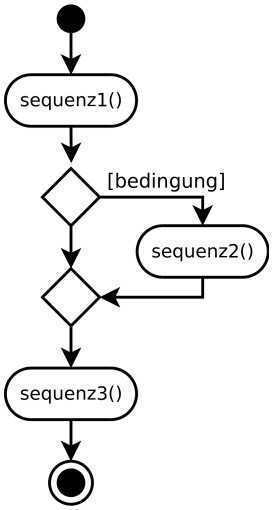
Neben der sequenziellen Ausführung von Anweisungen ist die **Selektion**[15] ein elementares Konzept der Programmierung. Eine Selektion steuert den Verlauf eines Programms. Bestimmte Teilsequenzen werden nur unter einer gegebenen Bedingung ausgeführt.

Die Selektion wird in JAVA durch das Schlüsselwort `if` eingeleitet. So wird in folgendem Beispiel die 2. Sequenz (`sequenz2()`) **nur** ausgeführt, wenn die Bedingung (`bedingung`) wahr ist. Die `sequenz1()` am Anfang und die `sequenz3()` am Schluss werden jedoch immer ausgeführt.

```
sequenz1();


if(bedingung)
{
  sequenz2();
}


sequenz3();
```



Der Bedingungsblock (hier `sequenz2()`) wird in JAVA mit geschweiften Klammern (`{`, `}`) eingeschlossen.

[15] Selektion wird auch Verzweigung, Entscheidung oder Auswahl genannt.

3.1.2 Syntax

Die generelle Syntax der Selektion in JAVA lautet:

```
if(<BEDINGUNG>) <BLOCK>
[ else <BLOCK> ]
```

Dabei bezeichnet der `<BLOCK>` eine Sequenz mit beliebig vielen Anweisungen. Ein Block sollte in JAVA immer in geschweifte Klammern («`{`»,«`}`») gestellt werden.

Der `else`-Teil ist optional (daher habe ich ihn in eckige Klammern gestellt; diese eckigen Klammern sind in JAVA natürlich nicht einzugeben).

3.2 Boole'sche Ausdrücke

Die Bedingungen werden durch Wahrheitsausdrücke (Boole'sche Ausdrücke) angegeben. Solche Ausdrücke erhalten beim Auswerten entweder `true` oder `false`.

Boole'sche Ausdrücke kennen ebenfalls die vier bereits behandelten Typen (s. Kap. 1.1.1 auf Seite 18):

- Literale: `true` und `false`
- Variable: `boolean ok = true; if(ok) ...`
- Funktionsresultat: `if(eingabeKorrekt()) ...`
- Zusammensetzungen: `(4 < x) && !(x >= 6)`

3.3 Das Bit

Die wohl einfachste aber in meinen Augen auch überwältigendste Datenstruktur ist das Bit (binary digit). Mit einem Bit kann einer von zwei binären Zuständen dargestellt werden.

Andere Wörter bzw. Vorsilben für binär sind: Zwei, Duo, dual, Boole'sch, stereo, sekund-, doppel-, zwie-, bi-, di-, ...

Physikalisch wird dieser Zustand verschieden dargestellt:

Zustand 1	*Zustand 2*	*Datenträger*
Finger gebogen	Finger gestreckt	Hand beim digitalen Zählen (lat. digitus=Finger; digitalis = zum Finger gehörig)
Magnetischer Nordpol	Magnetischer Südpol	Bei Disketten, Harddisks, Magnetbändern, ...
Kein Strom (z. B. < 0.5 Volt)	Strom (z. B. >3 Volt)	Datenübertragung zwischen Komponenten (Bus) des PCs
Papier	Loch	Lochstreifen
Keine Ladung	Ladung	Flipflops (Transistoren) im Speicher (RAM), MOSFET, Flash, Relais, ...
Reflexion des Lasers	Streuung des Lasers	CD-ROM, DVD, Blu-ray
tiefer Ton	hoher Ton	Akustikkoppler / Modem
Diode ist dunkel	Diode leuchtet	Statusanzeige für Harddisk Aktivität / Feststelltaste, Batteriestatus, ...
...	...	...

Ein Bit kann in der realen Welt nun die verschiedensten Bedeutungen haben:

Zustand 1	*Zustand 2*	*Bedeutung*
Falsch (false)	Wahr (true)	In verschiedenstem Kontext
Nein	Ja	Wahrheitswert auf eine Aussage
0	1	als Zahlen
nicht archiviert	archiviert	als sog. Flag (Markierung) einer Datenstruktur
Rot	Grün	bei einer Signalsteuerung
Geschlossen	Offen	Bei Barrieren / Türen / ...
Schwarz	Weiß	Als Figurenfarbe im Go-Spiel
Feststelltaste nicht aktiv	Feststelltaste eingeschaltet	Diodenanzeige «CAPS-LOCK»
...	...	...

3.3.1 Beispiele zu Boolean

```
boolean offen       ; // false = geschlossen
boolean archiviert  ; // false = nicht archiviert
boolean weiss_am_Zug; // false = schwarz_am_Zug
boolean one         ; // false = zero (Eins und Null)
```

Das Bit kann aber nicht nur Gegenteile darstellen. Nehmen wir zur Verdeutlichung ein Trinkglas. Sein Zustand kann auf zwei Arten modelliert werden, und es ist wichtig, dass wir die beiden Fälle Unterscheiden:

Variante 1:

```
boolean voll; // false = leer
```

Variante 2:

```
boolean voll; // false = nicht voll
```

Im ersten Fall kann ich mich auf den Standpunkt stellen: Ein Trinkglas, das noch etwas darin hat, ist voll. Ein solches Glas darf z. B. vom Kellner noch nicht abgeräumt werden.

Im zweiten Fall interessiert mich wohl eher, ob ein Glas ganz voll ist. ¾-voll wäre in diesem Umfeld eben nicht mehr **voll**. Mit anderen Worten: Das Gegenteil von «voll» ist eben nicht immer ganz «leer»; sicher aber immer «nicht voll»!

Es ist auf dieser Ebene die Aufgabe von uns Programmierern die reale Welt sinnvoll abzubilden. Wir müssen abklären, ob «nicht voll» gleich «leer» ist, ob «nicht rot» gleich «grün» ist etc.

3.3.2 Vergleichsoperatoren

Vergleichsoperatoren vergleichen numerische Werte miteinander und liefern als Resultat entweder `true` (bei Zutreffen) oder `false` (bei Nicht-Übereinstimmung).

Operator	in Java	Bedeutung
$<$	`<`	ist kleiner als
$\leq$	`<=`	kleiner oder gleich
$>$	`>`	ist größer als
$\geq$	`>=`	größer oder gleich
$=$	`==` 	identisch, gleich
$\neq$	`!=`	ungleich

3.3.3 Logische Operatoren

Logische Operatoren werden benötigt, um logische Ausdrücke (z. B. Vergleiche) miteinander zu kombinieren.

Die wichtigsten logischen Operatoren sind:

Operator	C/Java - Symbol	Beschreibung
$=$	`==`	Gleichheit
$\neq$	`!=`	Verschiedenheit
AND	`&`, `&&`	Und
OR	`|`, `||`	inklusives, mathematisches Oder
NOT	`!`	Verneinung, Umkehrung, Negation
XOR	`^`	exklusives Oder

Die Operatoren `&&` und `||` sind im mathematischen Kontext zu verstehen. `&&` heißt, dass beide Operanden links und rechts wahr sein müssen, damit der Gesamtausdruck wahr wird; wohingegen bei `||` verstanden wird, dass mindestens einer der beiden Operatoren wahr sein muss. Sprachlich ist dies je nach Formulierung aber genau umgekehrt und manchmal wollen wir nur ein **exklusives Oder**, was in JAVA mit `^` bewerkstelligt wird.

Hier ein Beispiel, bei dem in der Umgangssprache ein `&&` oder ein `||` steht, je nach der Satzstellung:

Ein «Kind» ist ein Mädchen **oder** ein Junge.

Mädchen **und** Jungen sind somit beides Kinder.[16]

Bemerkung 3.1. «Abkürzungsoperatoren»

Im vorigen Kapitel habe ich erwähnt, dass für **AND** `&&` oder `&` geschrieben werden kann. Analog gilt dies für **OR**.

Verwenden wir den Doppeloperator (`&&`, `||`) anstelle der einfachen Operatoren (`&`, `|`), so verwenden die C-Sprachen (und dazu gehört auch JAVA) eine **Abkürzung**. Falls nämlich beim Betrachten des linken Operanden das Resultat schon klar ist, wird der rechte Operand gar nicht mehr ausgewertet, was oft auch einen Zeitgewinn (Performance) bringt. Ist nämlich beim `&&` der linke Operand `false`, so ist der Gesamtausdruck auch `false` und zwar unabhängig vom rechten Operanden (Analog bei `||`).

Nichts verstanden? Da es in den allermeisten Fällen keine Rolle spielt, ob sie `&` oder `&&` als logischen Operator verwenden, gewöhnen Sie sich lieber gleich den Doppeloperator `&&` an.

[16] Das Beispiel ist aus [Knu97] und dort in englisch formuliert; es zeigt aber, dass es in deutsch genau dieselben sprachlichen Unklarheiten gibt.

3.3.4 Zusammensetzen von Vergleichen mit logischen Operatoren

In der Praxis werden Vergleichsoperatoren (`<=` , `>` , ...) oft mit logischen Operatoren (`&&` , `^` , ...) verknüpft. Ein klassisches Beispiel ist die Eingrenzung eines Zeichens (z. B. Buchstabe oder Ziffer) in untere und obere Schranken:

```
if(('a' <= ch) && (ch <= 'z'))
{
  System.out.println("Kleinbuchstabe");
}
```

Weitere Beispiele

- `if(isOK) {...}`
- `if((! eingabeKorrekt) || eingabeLeer) {...}`
- `if((alter < 19) && inAusbildung) {...}`

In obigem Beispiel sind `isOK`, `eingabeKorrekt`, `eingabeLeer` und `inAusbildung` Variable vom Datentyp `boolean`. Die Variable `alter` hingegen ist eine ganze Zahl (`byte`, `short`, `int` oder `long`).
Weitere Beispiele in den Aufgaben am Ende des Kapitels.

3.3.5 De Morgan'sche Regeln

Oftmals ist es einfacher in einem AND, statt in einem OR zu denken. Gerade, wenn eine Bedingung verneint werden soll.
So gelten die beiden folgenden De Morganschen Regeln[17]:

```
! (a & b) == (! a) | (! b)
! (a | b) == (! a) & (! b)
```

Beachten Sie die Vertauschung von `&` und `|` bei der Verneinung.

[17] Die beiden Regeln werden nach Augustus De Morgan (1806-1871) benannt, obschon diese bereits vor dessen Lebzeiten bekannt waren.

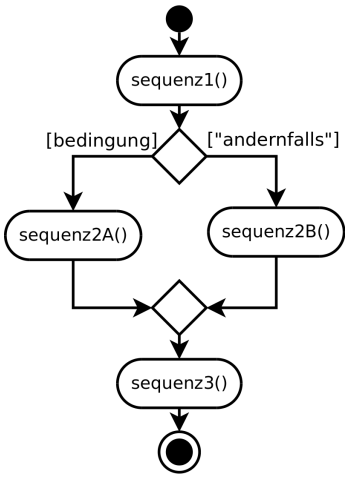
3.4 `else`

Alternativ auszuführender Code kann meist mit der folgenden Erweiterung der `if`-Struktur erreicht werden. Ein `else` bezeichnet einen Block, der genau dann ausgeführt werden soll, wenn die `if`-Bedingung falsch ist. Wir können somit zwei alternative Code-Blöcke schreiben, ohne die Bedingung explizit zu verneinen.

```
sequenz1();

if(bedingung)
{
  sequenz2A();
}
else
{
  sequenz2B();
}

sequenz3();
```



Bemerkung: In JAVA darf die geschweifte Klammerung (im `if`- sowie im `else`-Teil) unter gewissen Umständen weggelassen werden. Mehr dazu im Anhang (s. Kap. A.7 auf Seite 144).

3.5 Mehrfachselektion

Führen bei einer Auswahl mehrere Entscheidungen in verschiedene Wege, so kann mit der folgenden Mehrfachselektion entschieden werden, welcher Code denn nun ausgeführt werden soll:

```
     if (bedingung1)
{
  <BLOCK 1>
}
else if (bedingung2)
{
  <BLOCK 2>
}
else if (bedingung3)
{
  <BLOCK 3>
}
else
{
  <STANDARD-BLOCK>
}
```

Dabei wird `<BLOCK 1>` genau dann ausgeführt, wenn die `bedingung1` wahr ist, `<BLOCK 2>` genau dann, wenn `bedingung1` falsch war, jedoch `bedingung2` stimmt etc. Der letzte Block (`<STANDARD-BLOCK>`) wird dann ausgeführt, wenn keine der Bedingungen zutrifft.

3.5.1 switch

Eine Code-Auswahl aufgrund eines Variablen-Wertes kann mit der `switch()`-Anweisung vereinfacht werden.
Beispiel:

```
int note; /* Leistungsbeurteilung */
...
switch(note)
{
  case 1:
    print("sehr␣");     /* falls through */
  case 2:
    print("schwach");
    break;
  case 3:
    print("un");        /* falls through */
  case 4:
    print("genuegend");
    break;
  case 6:
    print("sehr␣");     /* falls through */
  case 5:
    print("gut");
    break;
  default:
    print("unbekannte␣Note" + note);
}
```

Beachten Sie die `break`-Anweisung. Diese verlässt den `switch`-Block augenblicklich. Eine gängige Fehlerquelle ist es, diese `break`-Anweisung zu vergessen. Wird die `break`-Anweisung nämlich weggelassen, (wie hier im Fall 1, 3 und 6), so wird einfach beim nächsten `case`-Block weitergefahren.
Die letzte `case`-Marke[18] wird hier von einer `default`-Marke gefolgt. Hier werden alle Fälle behandelt, zu denen es kein explizites `case` gibt.

[18] Sprungmarken werden oft auch Labels genannt.

3.6 Gebrochene Zahlen

3.6.1 Vergleiche mit Dezimalbrüchen

Vergleichen Sie gebrochene Zahlen nie auf ihre Identität (in JAVA also nie mit `"=="`). Denn zwei Zahlen können unter Umständen dasselbe Resultat bedeuten, sich durch Rundungsfehler aber auf den letzten Binärstellen unterscheiden. Fragen Sie sich beim Vergleich immer, wie genau sollen die beiden Zahlen beieinander liegen? Normalerweise sucht man sich eine sehr kleine Zahl (z. B. `epsilon = 0.00001`)[19] und schaut, ob sich die beiden ursprünglichen Zahlen um maximal diese kleine Zahl unterscheiden. Ist der Unterschied kleiner als dieses Epsilon, so kann davon ausgegangen werden, dass die Zahlen dieselben Resultate bedeuten:

Wie werden die beiden `double` Zahlen `zahl_1` und `zahl_2` nun miteinander verglichen? Etwa so?

```
if(zahl_1 == zahl_2)
{
  // beide Zahlen sind gleichwertig ???
  ...
```

Nein, bitte nicht. Streichen Sie obigen Block rot durch. Ein Vergleich von reellen Zahlen sollte Rundungsfehler auf den letzten Stellen berücksichtigen. So ist's

BRAV[20]:

```
double epsilon   = Math.max(zahl_1,  zahl_2) * 0.00001;
double differenz = Math.abs(zahl_1 - zahl_2)           ;
if(differenz < epsilon)
{
  // beide Zahlen sind gleichwertig
  ...
```

3.6.2 Prüfe auf Ganzzahligkeit

Wie auch bei Vergleichen mit gebrochenen Zahlen kann in der Regel von einer Variable nicht einfach gesagt werden, ob diese eine ganze Zahl darstellen soll. Rundungsfehler auf den letzten Binärziffern machen uns das Leben schwer. Wir können jedoch wieder eine Genauigkeit angeben, um damit auf Rundungsfehler zu prüfen. Verwenden Sie die folgende Funktion, um eine gebrochene Variable mit großer Wahrscheinlichkeit auf Ganzzahligkeit zu prüfen:

```
double bruch   = ...    ;
double epsilon = 0.00001;
if(Math.abs(bruch - Math.round(bruch)) < epsilon)
{
  ... // Der bruch ist wohl ganzzahlig
```

[19] Der Variablenname `epsilon` geht hier auf die sog. Epsilontik zurück, welche mit beliebig kleinen positiven Zahlen arbeitet.

[20] BRAV = **B**rüche **r**ücksichtsvoll mit **A**bstandsfunktion **v**ergleichen.

3.7 Funktionale Gebundenheit Boole'scher Ausdrücke

Boole'sche Ausdrücke sind Ausdrücke, wie andere auch. Sie liefern jedoch als Resultat immer nur entweder wahr (`true`) oder falsch (`false`). Mit diesem Wissen können wir sofort verstehen, warum die beiden nachfolgenden Codestücke äquivalent sind. Der erste Code ist a) schlechter zu lesen, b) fehleranfälliger und somit c) schlechter wartbar. Immer Köpfchen einschalten und mit LOGIK[21] arbeiten:

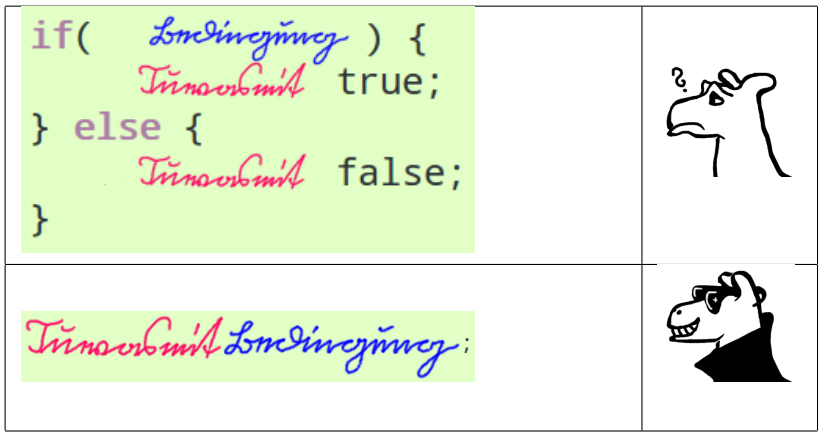
Erst ohne LOGIK:

```
boolean bedingung = ...;
if(bedingung)
{
  doSomething(true);
}
else
{
  doSomething(false);
}
```

Und nun das selbe mit LOGIK:

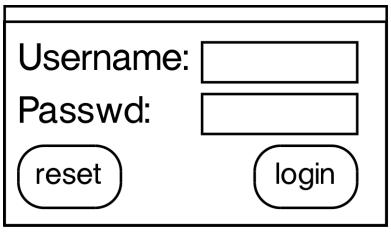
```
boolean bedingung = ...;
doSomething(bedingung);
```

Betrachten Sie auch die folgende Grafik: Die obere `if/else`-Variante ist absolut identisch mit dem unteren direkten Aufruf (Dabei handelt es sich beim blauen Ausdruck um irgendeinen Boole'schen Ausdruck und bei der roten Teilanweisung um einen beliebigen Code-Ausschnitt):



[21] LOGIK = **L**öse **o**ffensichtliche **G**ebundenheit **i**m **K**ontext.

Beispiel Dazu gleich noch ein klassisches, immer wieder auftauchendes Beispiel aus der Praxis. Dabei geht es um eine Maske, bei der sich der Benutzer anmelden kann. Es hat zwei Textfelder.



Im ersten Feld *username* kann der Benutzer seinen Namen eingeben und im zweiten Feld *passwd* kann er sein Passwort eintippen. Darunter sind zwei Knöpfe (Buttons). Der erste (*reset*) ist dazu da, die Felder zurückzusetzen; mit anderen Worten, den Inhalt der Textfelder wieder zu löschen. Der zweite Knopf, der *login*-Button, wird gedrückt, um sich am System anzumelden. Dabei ist der *reset*-Button immer aktiv, wenn mindestens eines der beiden Felder einen Text enthält; der *login*-Button hingegen ist nur dann aktiv, wenn beide Felder Text enthalten. Betrachten Sie nun die folgenden beiden Codestücke (zunächst wieder ohne LOGIK):

```
// username UND passwort eingetragen:
if( filled(username) && filled(passwd))
{
    reset.setEnabled(true);
    login.setEnabled(true);
}
// nur passwort leer gelassen:
if( filled(username) && !filled(passwd))
{
    reset.setEnabled(true);
    login.setEnabled(false);
}
// nur username leer:
if( !filled(username) && filled(passwd))
{
    reset.setEnabled(true);
    login.setEnabled(false);
}
// beide Felder leer:
if( !filled(username) && !filled(passwd))
{
    reset.setEnabled(false);
    login.setEnabled(false);
}
```

Doch nun besser (Streichen Sie obigen Code bitte rot durch!), diesmal mit LOGIK. Ist dieser Code nicht DUFTE[22]?

```
// mind. ein Feld ist eingetragen (ODER):
reset.setEnabled( filled(username) || filled(passwd) );
// beide Felder sind eingetragen (UND)
login.setEnabled( filled(username) && filled(passwd) );
```

[22] DUFTE = **D**irekt **U**nverändert **F**alse/**T**rue-Werte **e**ingesetzt.

3.8 Selektionsinvariante

Betrachten wir folgenden Code, so fällt auf, dass einige Aufrufe doppelt vorkommen:

```
if(A)
{
  f();
  x();
  g();
}
else
{
  f();
  y();
  g();
}
```

Der Code kann kürzer und effektiver wie folgt geschrieben werden, denn `f()` und `g()` sind unabhängig (also invariant) im Bezug auf die die Selektion (`if`); dies nenne ich «selektionsinvariant».

Betrachten und diskutieren Sie nun den korrigierten Code:

```
f(); // pre-Code
if(A)
{
  x();
}
else
{
  y();
}
g(); // post-Code
```

3.9 `int`, diesmal Bitweise

Eine spannende Möglichkeit, jedes Bit innerhalb eines `int`-Wertes als Wahrheitswert anzusehen ist die Bit-Maskierung (s. Kap. A.16 auf Seite 160).

3.10 Aufgaben

Aufgabe: Die richtige Wahl

Welches ist die richtige Wahl, wenn es darum geht, einen Wahrheitswert abzufragen. Im Beispiel wollen wir nur tränken, wenn das Kamel durstig ist. Dazu gebe es eine Variable durstig vom Datentyp boolean (Wahrheitswert). Betrachten Sie den folgenden Code:

```
boolean durstig;
durstig = ...;
/* A */  if(durstig && true ) { traenken(); }
/* B */  if(true && durstig ) { traenken(); }
/* C */  if(durstig = true  ) { traenken(); }
/* D */  if(true = durstig  ) { traenken(); }
/* E */  if(durstig == true ) { traenken(); }
/* F */  if(true == durstig ) { traenken(); }
/* G */  if(durstig         ) { traenken(); }
/* H */  if(false != durstig) { traenken(); }
```

Beantworten Sie die drei folgenden Fragen:

- Welche der obigen Abfragen können kompiliert werden?
- Welche der obigen Abfragen (die kompilieren) erfüllen die geforderte Bedingung?
- Welche Abfrage ist die beste? (Nur eine Wahl!)

Lösung im Anhang (s. Kap. B auf Seite 189).

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 3. 1 If-Formulierungen
- 3. 2 Boolesche Ausdrücke
- 3. 4 Selektionsinvariante
- 3. 5 Bedingte Zuweisung
- 3. 6 AHV
- 3. 7 Body-Mass-Index
- 3. 8 Hundert Binärzahlen
- 3. 9 Kilobyte
- 3.10 Elektrische Spannung
- Ausflugsplanung (Web-Code: 54d5-o0ns)

4 | Schleifen

Eines der wesentlichen Konzepte der Programmierung überhaupt ist die Möglichkeit ein Stück Code beliebig oft ausführen zu lassen. In der Maschinensprache wird dies mit einem sog. Sprung (Jump) an eine **vorangehende** Programmzeile gelöst. In JAVA wird dies generell durch eine `while`-Schleife (manchmal auch durch eine `for`-Schleife[23]) vollzogen.

Wir hatten mit der Selektion (s. Kap. 3 auf Seite 37) ein Verfahren kennengelernt, bei dem ein Stück Programmcode nur bei Eintreffen einer gegebenen Bedingung ausgeführt wird. Man könnte dies als bedingten **Sprung nach vorn** auffassen, bei dem ein Stück Code eventuell ausgelassen wird. Analog gibt es den «Sprung zurück», mit dem sich abbrechende oder nicht abbrechende Schleifen erzeugen lassen.

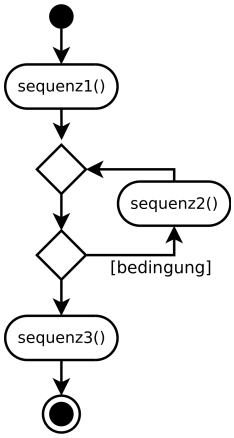
Für eine Wiederholung steht in JAVA i. d. R. das Schlüsselwort `while`. In folgendem Beispiel wird die 2. Sequenz (`sequenz2()`) **so lange** ausgeführt, wie die Bedingung (`bedingung`) wahr ist.

Wieder gilt: Die `sequenz1()` wird immer ausgeführt. Die `sequenz3()` hingegen erst, wenn die Bedingung nicht mehr erfüllt ist.

```
sequenz1();

while(bedingung)
{
  sequenz2();
}

sequenz3();
```



Die Bedingung steuert also Häufigkeit des Durchlaufs bzw. den Abbruch. Die `bedingung` ist ein boolescher Ausdruck (der als Resultat `true` oder `false` aufweist). Einige Sprachen erlauben auch Zahlen und werten diese verschieden aus. Die Programmiersprache «C» z. B. kann als Bedingung eine ganze Zahl entgegennehmen und führt die Anweisungen nur dann aus, wenn die Zahl nicht Null (0) ist; Null bedeutet also «falsch», alles andere bedeutet «wahr». Nur wenn die Bedingung wahr (`true` in JAVA) ist, so werden die Anweisungen ausgeführt. Beim Teil, der mit `sequenz2()` gekennzeichnet ist, kann es sich um beliebig viele Anweisungen handeln (also auch keine oder eine).

[23] Schleifen werden in der Schweizer Mundart oft auch *Schlaufen* genannt, was den Vorteil hat, es nicht mit einer Schleifmaschine oder einer Rutschbahn zu verwechseln ;-)

Die Schleife ist in allen Programmiersprachen der 3. Generation[24] vorhanden. Die `while()`-Schleife ist die grundlegendste Ablaufsteuerung in JAVA. Alle anderen (`if`, `do`, `for`; ja sogar `switch` und `break`, ...) ließen sich damit simulieren.

[24] JAVA ist eine objektorientierte Programmiersprache der 3. Generation. Höhere Generationen (wie z. B. Prolog oder SQL) verwenden Schleifen zur Lösungsfindung implizit, also ohne, dass sich der Programmierer darum kümmern müsste.

4.1 Anwendungen

Der wahre Geek versucht, die stumpfsinnigste Arbeit am PC zu minimieren[Him01]. Dazu gehört vor allem die Tipparbeit. Wenn wir ein Stück Code wieder benötigen, so sollten wir es unter keinen Umständen abtippen. Jede Tipparbeit ist fehleranfällig. Jedoch ist natürlich auch das Kopieren bösartig. Mit hintereinander kopieren kann nicht erreicht werden, dass eine Sequenz beliebig oft abgespielt wird. Wenn ich als Programmierer von vornherein nicht weiß, wie oft ein Code durchlaufen werden wird, so bin ich mit dem Kopieren von Code sowieso auf dem Holzweg.

Wozu werden nun Schleifen in der Praxis eingesetzt?

- Zählen und Summen bilden
- Suchen
- Anwenden einer Operation auf alle Elemente einer Datensammlung
- mehrere Eingabezeilen (ab User-Input, Files oder Netzwerk) lesen und verarbeiten
- Endlosschleifen (z. B. Simulatoren oder das Betriebssystem selbst.)

Geek Tipp 2

Wir tippen nichts zweimal. Wir kopieren niemals Code innerhalb eines Projektes!
Kopierter Code ist **böse**: Er verbreitet **Mühsal**, **Verderbnis** und **Tod**!

4.2 Syntax

Die generelle Syntax der JAVA-`while()`-Schleife lautet:

```
while (<BEDINGUNG>)
 <BLOCK>
```

Dabei ist die <BEDINGUNG> ein boolescher Ausdruck. Der <BLOCK> beinhaltet wie bei der Selektion (`if`) beliebig viele in geschweiften Klammern eingeschlossene Anweisungen.

Ebenso existiert in JAVA eine Fuß-gesteuerte Schleife, die in der Praxis aber selten Anwendung findet:

```
do
  <BLOCK>
while (<BEDINGUNG>);
```

Diese Bedingung muss genau gleich wie bei der kopfgesteuerten `while()`-Schleife den Wert `false` aufweisen um den Zyklus zu durchbrechen. Es handelt sich also **nicht um eine Abbruchbedingung** sondern auch um eine Wiedereinstiegsbedingung.

4.2.1 Beispiel

Hier das Zinseszinsbeispiel aus dem Buch [GFG11] Seite 45; diesmal in JAVA:

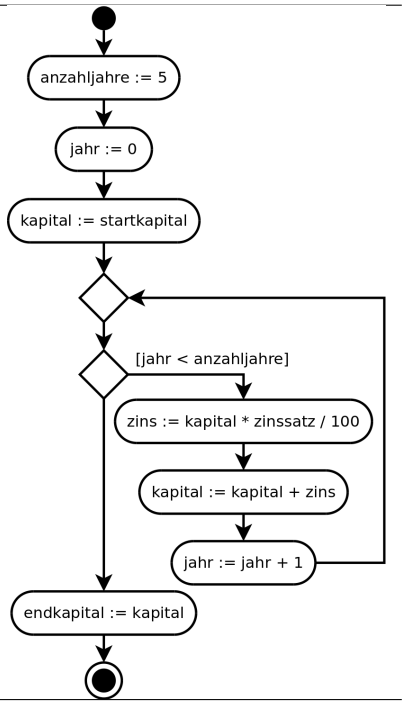
```
int anzahljahre = 5;

int      jahr     = 0;

double   kapital = startkapital;


while(jahr < anzahljahre)
{
   zins     = kapital * zinssatz / 100;
   kapital = kapital + zins           ;
   jahr     = jahr    + 1             ;
}

endkapital = kapital;
```



Beachten Sie, dass eine Division ein Risiko mit sich bringt. Sind beide Zahlen ganzzahlig (wie z. B. `7/3`), so wird in JAVA eine ganzzahlige Division durchgeführt: `7 / 3 = 2`. Soll ein Dezimalbruch errechnet werden, so ist der Dividend oder der Divisor auch als Dezimalbruch anzugeben:

`7.0 / 3 = 7 / 3.0 = 2.333...`

4.3 Zählervariable

Sehr oft wird in einer Schleife ein Zähler benötigt, um den Abbruch steuern zu können (wie `jahr` im Beispiel «Zinseszins» auf Seite 56).[25]
Meist wird dazu eine **ganze** Zahl verwendet; welche in einer sog. **Zählervariable** (im Folgenden `int i`) gespeichert wird:

```
int i = 0;
while(i < 20)
{
 ...
 i = i + 1;
}
```

Weil solche Zähler-Schleifen so häufig sind, benutzt JAVA hier die Konstruktion aus der Programmiersprache C:

```
for(int i = 0; i < 20; i = i + 1)
{
  ...
}
```

Bemerkung: Anstelle von `i = i + 1;` wird abkürzend meist `i++;` verwendet (s. Kap. 2.2.1 auf Seite 31).

Aber Achtung: Verwenden Sie den `++`-Operator entweder nur in diesem Zusammenhang, oder aber erst dann, wenn Sie ihn vollständig verstanden haben, denn `i++` ist eine Anweisung und gleichzeitig ein Ausdruck mit dem Nebeneffekt, den Wert der Variable zu erhöhen. Probieren Sie einmal folgendes aus, staunen Sie und lernen Sie daraus:

```
int i = 5;
i = i++;
System.out.println("Neues␣i:␣" + i);
```

Lösung: Als Anweisungen sind `i++;` und `i = i+1;` zwar identisch. Genau genommen bezeichnet `i++` als **Ausdruck** jedoch `((i = i + 1) - 1)`, was in obigem Beispiel dem `i` wieder den alten Wert – nämlich 5 – zuweist.

[25] Natürlich kann die Abbruchbedingung auch ganz anders definiert sein: Ziel erreicht, Maximum überschritten, Benutzereingabe, Spielende, ...

Geek Tipp 3

Abkürzungen in Programmiersprachen sind für Fortgeschrittene. Verwenden Sie solche erst, wenn Sie sie vollständig verstanden haben und sicher gehen, dass auch niemals Anfänger Ihren Code jemals warten (also verstehen) müssen. Mit anderen Worten: Verzichten Sie so oft als möglich auf Abkürzungen.

4.3.1 Die `for`-Schleife

Die allgemeine Syntax der `for`-Schleife ist links gegeben und entspricht 1:1 der `while`-Schleife rechts:

`for`

```
for(<INIT>; <BEDINGUNG>; <NEXT>)

  <BLOCK>
```

`while`

```
<INIT>
while(<BEDINGUNG>)
{
  <BLOCK>
  <NEXT>;
}
```

Gleich ein Beispiel (hier werden die ersten zehn Quadratzahlen berechnet und ausgegeben):

Mit `for`

```
for(int i = 1; i <= 10; i = i + 1)
{
  System.out.println(i +
    ":␣" + (i*i));

}
```

Mit `while`

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

4.4 Sichtbarkeit (Scope)

In früheren Programmiersprachen waren Variable meist global. Das hat dazu geführt, dass die Programmierer ständig aufpassen mussten, dass Sie den Variablen Namen gaben, die im ganzen Projekt eindeutig waren. Moderne Sprachen verwenden eine Variable nur gerade in ihrem unmittelbaren Kontext (Modul oder Funktion). Die Sichtbarkeit von Variablen geht aber meist noch etwas weiter. So sind in JAVA Variable nur innerhalb des Blocks (normalerweise zwischen auf- und zugehender geschweifter Klammer) sichtbar.

Beispiel:

```
 1. int a = 2; // ueberall im folgenden sichtbar
 2. while(a < 4)
 3. {
 4.   int b = 6; // sichtbar bis zur Zeile 12
 5.   b = b - a;
 6.   a = a + 1;
 7.   if(5 == a) {
 8.     int x = 6; // a, b und x sichtbar
 9.     x = x - b;
10.   } // ab hier ist x nicht mehr sichtbar
11.   a = a + 1;
12. } // nur noch a sichtbar ab hier.
13. a = 10;
```

Bemerkung 4.1. Weitere Spezialitäten von JAVA-Schleifen finden sich im Anhang (s. Kap. A.8 auf Seite 145).

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 4. 1 Iterationen
- 4. 2 Hundert Quadratzahlen
- 4. 3 Kleines Einmaleins
- 4. 4 Fahrenheit
- 4. 5 Domino
- 4. 6 Wertetabelle
- 4. 8 Selektion vermeiden
- 4.10 Zahlensumme (1)
- 4.11 Multiplikation
- 4.12 Fakultäten
- 4.13 Zahlensumme (2)
- 4.14 Sinusfunktion
- 4.16 Mondlandung
- 4.17 Bienenflug
- 4.18 Das Buch der Weissagungen
- 4.19 Parallelschaltung Ohm'scher Widerstände
- 4.20 Anzahl Ziffern
- 4.21 Ziffernfolge umdrehen
- 4.22 FizzBuzz (1)
- 4.23 FizzBuzz (2)
- 4.24 Waage

5 | Unterprogramme (Subroutinen)

Neben der Selektion und der Iteration bietet sich mit Unterprogrammen (Subroutinen) eine dritte Möglichkeit an, den Kontrollfluss zu steuern.

Bereits im Kapitel über Anweisungen (s. Kap. 2.2 auf Seite 30) sind uns Unterprogramme begegnet. Wir hatten dort vorgegebene Systemfunktionen aufgerufen:

- `System.out.println("...")`)
- `Math.sqrt(...)`

Neu ist in diesem Kapitel, dass wir solche Teilprogramme auch selbst definieren können.

Ein Unterprogramm[26] ist eine mit einem Namen versehene Sequenz. Eine Unterprogramm kann später aus beliebigen Stellen des Programmes aufgerufen werden.

Unterprogramme sind der Kern der «Strukturierten Programmierung». Damit werden komplexe Abläufe in fassbare kleine Problemstellungen aufgeteilt. Dies macht den Code überblickbar und besser wartbar.

Der Begriff **Divide et Impera** ist lateinisch und steht für Teile und Herrsche[27]. Wir verwenden das Prinzip von kleinen Teilprogrammen, um ein größeres Problem in den Griff zu bekommen[28].

Das Konzept ist auch bekannt unter dem Namen *funktionale Dekomposition*. In der Praxis spricht man von schrittweiser Verfeinerung oder einfach vom Verwenden von Unterprogrammen. Diese kommen in verschiedenen Sprachen mit diversen «Möglichkeiten» vor:

- Subroutinen ohne Rückgabewerte nennt man auch **Prozeduren**.
- Subroutinen mit Rückgabewerten nennt man meistens **Funktionen**.
- Subroutinen, die auf Objekten wirken nennen wir auch **Methoden**.

Ist die Ausführung der Unterprogramm-Sequenz abgeschlossen, so wird das Programm dort weitergefahren, woher der Unterprogramm Aufruf stattgefunden hat. Unterprogramme brauchen nur einmal definiert zu werden, auch wenn sie an verschiedenen Stellen aufgerufen werden, oder wenn sie innerhalb einer Schleife mehrmals aufgerufen werden.

[26] Ich werde im folgenden den Begriff Unterprogramm synonym zu Subroutine verwenden.

[27] Der Begriff wird Ludwig XI (1423-1483) zugeschrieben. Streue Zwietracht unter die Feinde und herrsche ungestört. Doch bereits 2000 Jahre zuvor ergänzt Jia Lin das Buch von Sun Tzu [Tzu00] um Methoden, die dem Feind schaden; insbesondere: «Säe Zwiespalt zwischen dem (gegnerischen) Herrscher und seinen Ministern».

[28] Schon Sun Tzu rät in seinem Buch «Die Kunst des Krieges», die Armeen aufzuteilen, um eine große Streitmacht zu führen. In dieser Bedeutung war der Begriff «Divide et Impera» definitiv schon im römischen Reich - wenn vielleicht auch nicht unter diesem Namen bekannt.

5.1 Vor- und Nachteile von Unterprogrammen

Vorteile:

- Das Programm ist einfacher zu lesen und dadurch auch besser wartbar und veränderbar.
- Einzelne Subroutinen können auch an anderen Stellen und sogar in anderen Programmen unverändert eingesetzt werden.
- Programme, die in Subroutinen aufgeteilt werden, sind weniger fehleranfällig und somit robuster.
- Gewisse Berechnungen stehen nun an einer wohl definierten Stelle im Code.
- Fehlerprüfungen können einfacher eingebaut werden.
- Kürzere Entwicklungszeiten bei häufiger Nutzung der selben Subroutine. Ebenso kann die Zeit zur Fehlersuche reduziert werden.

Nachteile:

- Mehr Tipparbeit
- Programme werden (wenn auch meist kaum messbar) langsamer in der Ausführung (Performanz).

Geek Tipp 4

Wer komplizierten Code schreibt, wird oft zu Unrecht bewundert: Es ist weitaus schwieriger verständlichen Code zu schreiben.
Die Krux der Applikationsentwickler ist die Entscheidung, ob komplizierter, unverständlicher Code geschrieben wird, damit man im Projekt unentbehrlich wird, oder ob man besser – und dies ist weitaus schwieriger – sauberen verständlichen Code schreibt, mit der Tatsache im Auge, dass man nun jederzeit ersetzbar ist.
Tun Sie sich selbst den Gefallen und schreiben Sie verständlichen Code!

5.2 Arten von Unterprogrammen

Wir werden in den folgenden Kapiteln vier Kategorien von Unterprogrammen kennenlernen:

Die vier Arten von Unterprogrammen	
Prozeduren	Funktionen
Prozeduren haben keinen Rückgabewert.	*Funktionen haben wie Variable einen (Ausdrucks-)Wert.*

Unterprogramme **ohne Argumente**	
Einfaches Unterprogramm (s. Kap. 5.2.1 auf Seite 66)	**Abfragefunktion** (s. Kap. 5.2.4 auf Seite 69)
Beispiele	
• `beep();` • `exit();` • `logTimestamp();` • `pause();`	• `currentTimeMillis()` • `getUserName()`

Unterprogramme **mit Argumenten**	
Prozedur mit Argumenten (s. Kap. 5.2.2 auf Seite 67)	**Funktion mit Argumenten** (s. Kap. 5.2.5 auf Seite 70)
Beispiele	
• `println("Hallo");` • `wait(millisecods);` • `printFibonacciNumbers(25);`	• `sin(360)` • `dreiecksflaeche(3, 4, 5)`

Aufruf ist jeweils...	
... als Anweisung	... als Ausdruck

Beachten Sie, dass die Prozeduren mit einem Strichpunkt (`;`) beendet wurden. Dabei handelt es sich in aller Regel um eine Anweisung, wohingegen die Funktionen als Ausdrücke verwendet werden (sollten) und somit nicht als Anweisungen stehen werden.

5.2.1 Primitive Unterprogramme

Unterprogramme ohne Argumente und ohne Rückgabewerte werden eingesetzt, um Programmteile zu kennzeichnen (einen Namen zu geben) und um das Programmstück aus verschiedenen Stellen im Hauptprogramm auszurufen.

Beispiele

- Beep: Einen Warnton ausgeben.
- Exit: Programm verlassen.
- Pause: Die Programmausführung um eine Sekunde blockieren.
- Log: Den aktuellen Zeitpunkt (Timestamp) in Erfahrung bringen und diesen z. B. in einer Log-Datei ausgeben (z. B. zur Geschwindigkeitsprüfung).
- ...

Geek Tipp 5

Teile den Code in überblickbare Teilprogramme. Ein Maximum von sieben Codezeilen ist ein guter Richtwert.

5.2.2 Parameter und Argumente

In JAVA können bei einer Subroutine beliebig viele[29] Parameter deklariert werden. Jeder Parameter hat einen Datentypen und einen formalen Parameter**namen**.

Beispiel 5.1. Hier einige Funktionsköpfe (Prototypen) mit keinem, einem oder mehreren Parametern:

```
void mainLoop()
void println(String text)
void wait(long milliseconds)
void wait(long milliseconds, int nanoseconds)
```

`void`: Prozeduren, also Subroutinen, welche **keinen** Wert zurückgeben, werden in JAVA mit dem Schlüsselwort `void` gekennzeichnet.

Beim Aufruf der Subroutinen geben wir den Parametern Werte mit. Diese aktuellen Werte der Parameter nennt man auch «Argumente».

Hier einige Prozeduraufrufe:

```
mainLoop()            ; // Ohne Argumente
println("Hallo Welt"); // Argument ist der String "Hallo Welt"
wait(2000)            ; // Argument ist "2000 (Millisekunden)"
wait(2000, 500 + x)   ; // Argumente sind hier zwei Zahlen

x = 5200 ;
wait(x)   ; // Argument ist die Zahl 5200 (nicht die Variable x!)
```

Beispiel 5.2. Das folgende Beispiel soll einfach die ersten Quadratzahlen auf der Konsole ausgeben. Dazu ist lediglich die Zahl `max` wichtig, welche angibt, bei welcher Quadratzahl dann das Ende des Programmes erreicht sein soll:

```
void zeigeQuadratzahlen(int max)
{
  for(int i = 0; i <= max; i++)
  {
    System.out.println("i : " + (i*i));
  }
}
```

Bemerkung 5.1. Subroutinen können denselben Namen, aber eine andere Anzahl formaler Parameter aufweisen. Diesen Sachverhalt nennen wir entweder «statischen Polymorphismus» oder das überladen von Subroutinen (s. Kap. A.11 auf Seite 149).

[29] Beliebig viele heißt hier aber auch: Eine Subroutine kann ohne Parameter auskommen.

5.2.3 Unveränderbarkeit (keine variable Parameter)

Wie erwähnt, nennen wir die übergebenen Werte[30] **Argumente**. Argumente sind in JAVA-Unterprogrammen **nicht veränderbar**. Ob ein Parameterwert mittels Literal, Variable, Funktionsresultat oder als zusammengesetzter Ausdruck mitgegeben wird, kann die Funktion nicht unterscheiden, auch dann nicht, wenn die aufrufende Variable denselben Namen hat:

Beispiel 5.3. Gegeben ist die folgende Funktion:

```
void change(int x)
{
  System.out.println("start␣x:␣" + x);
  x = x + 1;
  System.out.println("end␣␣␣x:␣" + x):
}
```

Aufrufe:

```
int i = 6;
int x = 4;
                    //      Ausgabe:
change(6);          // ->  6,  7
change(i + x * 2); // -> 14, 15
change(x);          // ->  4,  5
change(x);          // ->  4,  5 !! x wird NICHT veraendert!
```

Beachten Sie: Die Variable `x` aus dem Aufruf (zweiter Programmblock), wird innerhalb der Methode `change()` niemals verändert! Es handelt sich dort um eine andere Variable `x`, welche zufällig denselben Namen trägt.

Bemerkung: Die Parameterwerte (Attribute) und Werte der lokalen Variablen[31] liegen in der Maschine auf dem sog. Stack (Stapel)[32] und haben nur während der Ausführung der Subroutine ihre Gültigkeit.

Bemerkung 5.2. Zur Syntax von JAVA-Subroutinen siehe hier (s. Kap. A.10 auf Seite 148).

Aufgabe 5.1 «Name und Alter» Schreiben Sie eine Subroutine `nameUndAlterAusgeben(String name, int jahrgang)`, welche eine persönliche Begrüßung und das Alter ausgibt. Achtung: Das Alter wird der Subroutine nicht mitgegeben; nur der Jahrgang!

[30] JAVA übergibt **immer die Werte** der Ausdrücke, niemals die Variablen, auch dann nicht, wenn ein Ausdruck genau aus einer Variable besteht. Ein sog. Variablenparameter ist in JAVA also nicht möglich.
[31] Für lokale Variabel Siehe Seite 74
[32] Stack Siehe Anhang auf Seite 150

5.2.4 Rückgabewerte (Parameterlose Funktionen)

Subroutinen können nicht nur Attributwerte als Parameter entgegen nehmen; sie können auch Werte zurückgeben.

Unterprogramme, welche einen Wert zurückgeben nennen wir auch **Funktionen**.

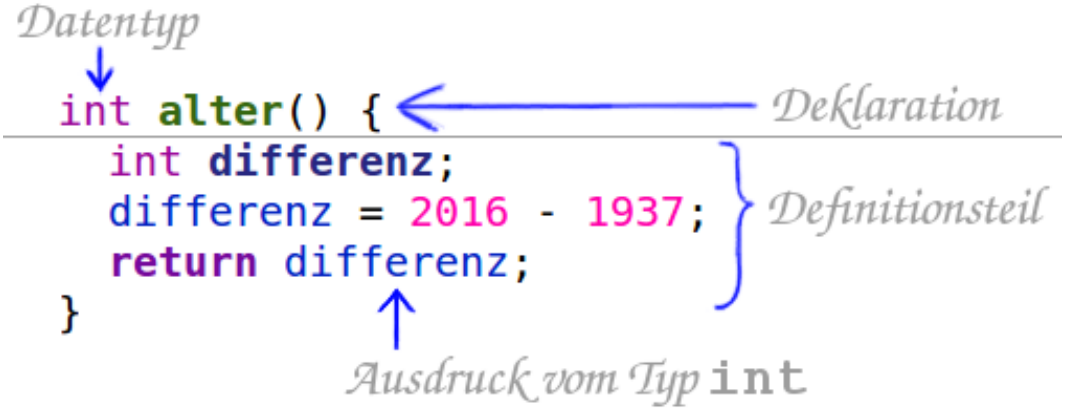
Beispiel:

`currentTimeMillis()` liefert die Systemzeit in Millisekunden (gezählt ab dem 1. Januar 1970).

`getUserName()` liefert den Namen des aktuell eingeloggten Benutzers[33].

Rückgabewerte werden verwendet, wie jeder andere Ausdruck auch. Funktionen werden **nie** als `void` deklariert. Es ist bei der Deklaration der Funktion immer ein Datentyp anzugeben. Mittels `return` wird bei der Funktionsdefinition stets ein Wert von diesem Datentyp zurückgegeben.

Beispiel:



Obige Funktion berechnet die Differenz zwischen zwei Jahreszahlen. Diese Differenz wird als `alter()` zurückgegeben.

Gleich ein zweites Beispiel, welches den Jahrgang via Konsole erfragt und als `int` zurückgibt:

```
int jahrgang() {
  System.out.println("Bitte␣Jahrgang␣eingeben:␣");
  String jahrString = new Scanner(System.in).nextLine();
  int    jahr       = Integer.parseInt(jahrString);
  return jahr;
}
```

[33]Leider unterscheidet sich hier der JAVA-Funktionsname je nach Betriebssystem. Unter MS-Windows lautet der Befehl `getName()`.

5.2.5 Klassische Funktionen

Eine klassische Funktion berechnet einen Wert abhängig von einer Eingabe nach dem EVA-Prinzip. **EVA** bezeichnet hier *Eingabe-Verarbeitung-Ausgabe*. Bei JAVA-Funktionen ist die **Eingabe** die Werteliste der Argumente, die **Verarbeitung** wird durch den Funktionsrumpf (Body) erreicht und die **Ausgabe** ist der Wert, welcher dem aufrufenden Programm zurückgegeben wird.

Beispiel 5.4.

Das folgende Unterprogramm berechnet das Volumen eines Quaders:

```
double volumen(double breite, double hoehe, double tiefe)
{
  return breite * hoehe * tiefe;
}
```

Aufgerufen wird dies dann ganz natürlich:

```
double vol1 = volumen(2.0, 3.8, 1.5);
double b    = 6  ;
double h    = 5.9;
double vol2 = volumen(  b,   h, 3.0 + 4.8);
```

Für die Rückgabe steht der JAVA virtuellen Maschine nur ein Register (4 Byte) zur Verfügung[34]. Dies reicht für Zahlen und Zeichen. Die Rückgabe in JAVA erfolgt mit dem Schlüsselwort `return`. Da JAVA eine typisierte Sprache ist, muss der zurückgegebene Wert denjenigen Datentypen aufweisen, der im Prototypen der Funktion (also in der Kopfzeile) angegeben wurde:

Beispiel 5.5. Die folgende Funktion halbiert Zahlen und muss somit auch gebrochene Werte (Dezimalbrüche) zurückgeben können:

```
double halbieren(int zahl) {
  double haelfte;
  haelfte = zahl / 2.0;
  return haelfte;
}
```

Der Aufruf eines selbst geschriebenen Unterprogrammes ist identisch mit den vorgegebenen Systemfunktionen (s. Kap. 1.4.5 auf Seite 23). Auch hier kann die Subroutine wie ein Ausdruck verwendet werden:

```
double d;
d = halbieren(4) + halbieren(3) +
    halbieren(3*8 + ((int) halbieren(6)));
```

[34] Ausnahmen: für `double` und `long` werden 8 Byte benötigt, also zwei Register. Andere Programmiersprachen (z. B. Python oder Ruby) können via `return` gleich mehrere Werte zurückgeben.

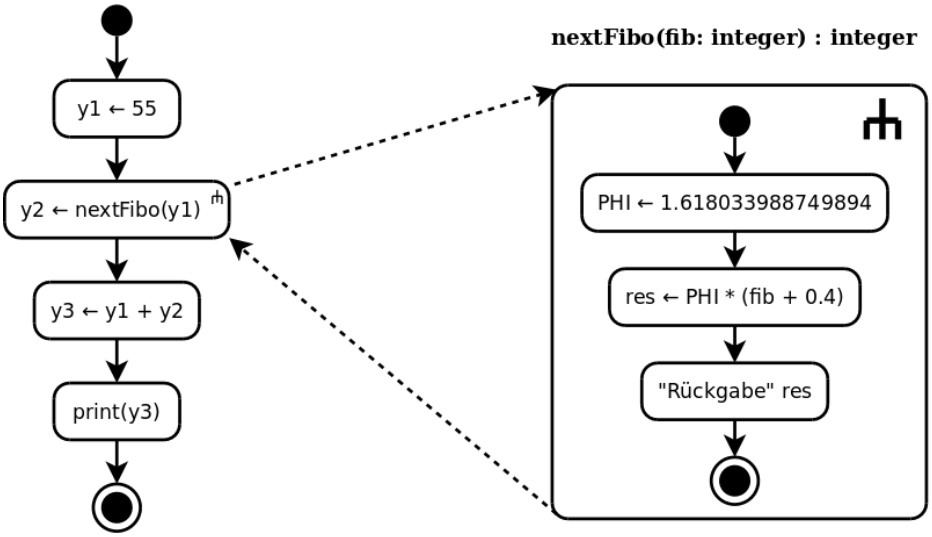
Für kompliziertere oder zusammengesetzte Funktionsresultate verwendet man Referenzen auf Objekte. Es sind Objekte, die man bereits mitgibt und deren Werte verändert werden, oder aber man erzeugt innerhalb der Subroutine ein neues Objekt. Dazu mehr im Band «Objekte und Klassen» [GF14].

Das folgende Beispiel zeigt, dass jeder Wert eines Ausdrucks direkt mit `return` zurückgegeben werden kann. Einzig der Datentyp muss übereinstimmen.

Beispiel 5.6. In der folgenden Subroutine `istGerade()` wird im Parameter `p` eine ganze Zahl entgegengenommen. Es wird geprüft, ob der Zweierrest (MOD 2) den Wert Null (0) ergibt. Ist der Zweierrest 0, so ist `p` gerade und es wird dem aufrufenden Programm `true` zurückgegeben (return), ansonsten `false`.

```
boolean istGerade(int p) {
  return 0 == p % 2;
}
```

Beispiel 5.7. Das Unterprogramm `nextFibo()` (rechter Teil der Grafik) berechnet aus einer Fibonacci-Zahl seinen Nachfolger in der Fibonacci-Reihe. [35]



Aufrufendes Programm:

```
int y1, y2, y3;

y1 = 55;
y2 = nextFibo(y1);
y3 = y1 + y2;

print(y3);
```

Unterprogramm:

```
int nextFibo(int fib) {
  double PHI = 1.618033988749894;
  double res;

  res = PHI * (fib + 0.4);
  return (int) res;
}
```

[35] Die Fibonacci-Reihe beginnt mit 1, 1, 2, 3, ... Jeder Nachfolger wird aus der Summe der beiden letzten Zahlen der Reihe berechnet. Beachten Sie auch die rekursive Berechnung aus [GFG11] dort: Aufgabe 11.2 auf Seite 143 und beachten Sie weiter die Bemerkungen in diesem Skript (s. Kap. 11.1 auf Seite 121). Die direkte Berechnung (also ohne Iteration oder Rekursion) ist sicher die rascheste Berechnung für große Werte.

5.3 Ausdrücke 2. Teil

Neben den bereits gelernten Ausdrücken (s. Kap. 1.1.1 auf Seite 18) können wir nun auch selbst Funktionen schreiben, die als Ausdrücke verwendet werden können. Dabei ist einzig darauf zu achten, dass es sich um Funktionen mit Rückgabewert (also keine `void`-Methoden) handelt. Gleich ein Beispiel: Die linke Methode kann nun verwendet werden, als wäre es ein «Built-In», also eine vorgegebene, Funktion.

```
long kubik(int s)
{
  int a, b, c, d;
  a =  6;
  b = 12;
  c =  7;
  d =  1;
  while(s > 1)
  {
    d = d + c;
    c = c + b;
    b = b + a;
    s = s - 1;
  }
  return d;
}
```

```
void top() {
  System.out.println("Kubik von 8 : "
    + kubik(8));
  System.out.println("Kubik von 6 : "
    + kubik(6));
}
```

5.3.1 Zusammenfassung der wichtigsten Anweisungen und der Ausdrücke

wichtigste ***Anweisungen***	***Ausdrücke***
Deklaration `int breite;`, `char niveau;`, ...	Literal `5`, `'c'`, `"Hallo Welt"`, `3.7e5f`, ...
Zuweisung `breite = 4;`, ...	Variable `breite`
Prozeduraufruf `beep();`, `println("Hallo");`, ...	Funktionsresultat `getDate()`, `sin(x + 45)`, ...
Ablaufsteuerung `if()...`, `while()...`, ...	Zusammengesetzte Ausdrücke `x + sin(45 + p)`, ...

Beachten Sie nochmals: Anweisungen werden in JAVA immer mit einem Semikolon (Strichpunkt) beendet – Ausdrücke hingegen nicht!

5.4 Lokale und globale Variable

Innerhalb von Unterprogrammen existiert wie bereits erwähnt ein eigenständiger Speicherbereich (Stapel bzw. *stack*[36]). Damit ist es möglich, Variable zu deklarieren, die nur während der Ausführung des Unterprogramms Gültigkeit haben. Diese Variable werden **lokale Variable** genannt. Sie können dieselben Namen tragen, wie Variable im aufrufenden Programm, ohne mit diesen in Konflikt zu geraten. Sobald zum aufrufenden Programm zurückgekehrt wird, verlieren lokale Variable ihre Werte, d. h. die lokalen Variable können nur innerhalb der entsprechenden Unterprogramme verwendet werden. Damit ein Unterprogramm auch Werte von Variablen im Hauptprogramm verändern kann, verwendet man entweder **globale Variable**, die aus allen Programmteilen sichtbar sind, oder man übergibt den Unterprogrammen Referenzen (Speicheradressen) auf Variable, die im aufrufenden (Haupt-)Programm deklariert sind (Referenzvariable oder «variable Parameter» genannt).[37]

Lokale Variable liegen auf dem Stack und werden vernichtet, sobald die Routine (das Unterprogramm) seinen Dienst erfüllt hat. Globale Variable liegen auf dem Heap und werden erst vernichtet, sobald kein Unterprogramm mehr darauf zugreifen kann.

Code:

```
class Test {
  int a = 3;          // global: in allen Unterprogrammen sichtbar

  void top() {
      int b = 4;      // lokal
      a = fct1(b);
      System.out.println("A=" + a + "␣B=" + b);
  }

  int fct1(int x) { // Parameter sind auch nur lokal
      return (3 + x) * x + a;
  }
}
```

Geek Tipp 6

Lokale Variable sind globalen vorzuziehen!

[36] S. S. 150

[37] Natürlich können wir **Werte** im Hauptprogramm auch mittels Funktionsrückgabe verändern, indem wir diese als Ausdrücke verwenden; doch hier geht es im Moment ausschließlich um das Verändern von **Variablen**.

5.5 Geringschätzung...

Jedes Funktionsresultat ist ein Ausdruck und **keine Anweisung**. Ausdrücke sollten wenn immer irgendwo verwendet werden. So erstaunt dieser ignorierende Code jeden erfahrenen Programmierer:

```
int x;
x = 4;
x = x * 6;
doppelDecker(x); // *** ??? ***
x = x + 2;
```

bei der folgenden gegebenen Methode:

```
int doppelDecker(int a)
{
  int resultat;
  resultat = 2 * a + 2;
  return resultat;
}
```

Die Methode `doppelDecker()` hat eine Aufgabe zu erfüllen. Sie berechnet eine Zahl (hier `resultat` genannt). Dieses Resultat wird der aufrufenden Programmsequenz übergeben, aber ... oh Wunder ... dort (an der mit `*** ??? ***` bezeichneten Stelle) nicht verwendet. Der Programmierer, der die Funktion `doppelDecker()` aufruft, vernichtet geringschätzig (oder mangels Verständnis des Funktionsresultates) die Arbeit derjenigen Person, die die Methode `doppelDecker()` implementiert hatte. Mit demselben Resultat[38] hätte unser Ignorant folgendes schreiben können:

```
int x;
x = 4;
x = x * 6;
50;
x = x + 2;
```

Programmiersprachen, die das Vernichten von Funktionsresultaten überhaupt nicht erlauben sind hier hoch zu loben. In den wenigen Fällen, wo das Resultat tatsächlich nicht verwendet wird, ist das mit einem Kommentar zu erklären. Ein `call`-Schlüsselwort wie in anderen Programmiersprachen gibt es in JAVA leider nicht!

[38] Natürlich wird an der mit `***???***` markierten Stelle die Funktion als Anweisung ausgeführt. Dadurch wird der Prozessor aufgewärmt. Doch mehr als heiße Luft produziert es nicht, wenn wir eine Funktion nicht als Ausdruck, sondern als Anweisung verwenden!

Geek Tipp 7

Verwenden Sie Prozeduren als Anweisungen!
Verwenden Sie Funktionen stets als Ausdrücke!

5.6 Wächter

Ein Aufruf von `return` beendet die aktuelle Methode sofort. Dieses Wissen kann verwendet werden, um sog. Wächterabfragen in den Code einzubauen, ohne dass der Code bei jeder Abfrage um eine Einrückung weiter geschoben werden muss.

Gehen wir einmal von der folgenden bekannten Subroutine aus, welche den Schalttag zurückgibt. Ist ein Jahr ein Schaltjahr, so wird Eins (`1`) zurückgegeben, in allen anderen Fällen die Null (`0`):

```
int schalttag(int jahr) {
  if(jahr %   4 != 0)                  { return 0; }
  if(jahr % 400 == 0)                  { return 1; }
  if(jahr % 100 == 0 && jahr > 1582) { return 0; }
  return 1;
}
```

Nun wollen wir den letzten Tag in einem Monat berechnen und benötigen dazu sogar noch das Jahr; nämlich für den letzten Tag jeweils im Februar. Um mögliche fehlerhafte Eingaben zu vermeiden, müssen wir sowohl das Jahr, wie auch den Monat auf Gültigkeit prüfen.

```
int letzterTagImMonat(int monat, int jahr) {
  if(jahr != 0)
  {
    if(monat >= 1 && monat <= 12)
    {
      if(4 == monat || 6 == monat || 9 == monat || 11 == monat)
      {
        return 30;
      } else {
        if(monat != 2) {
          return 31;
        }
      }
      return 28 + schalttag(jahr);
    }
  }
  // Fehler:
  return 0;
}
```

Doch mit etwas GRIPS[39] könnten wir Wächterabfragen einbauen, welche die Fehler von vornherein ausschließen und den Code doch einiges besser lesbar machen:

```
int letzterTagImMonat(int monat, int jahr) {
  // Fehlerbehandlung (Waechter)
  if(jahr == 0)                return 0;
  if(monat < 1 || monat > 12) return 0;

  // Hauptteil
  if(4 == monat || 6 == monat || 9 == monat || 11 == monat)
  {
    return 30;
  }
  if(monat != 2)
  {
    return 31;
  }
  return 28 + schalttag(jahr);
}
```

Bemerkung: Streng genommen unterwandern wir mittels `return` aus einer Kontrollstruktur (`if`, `while`) die «Strukturierte Programmierung». Es handelt sich hier lediglich um eine Fehlerbehandlung, welche nach Funktionsende zum selben Punkt zurück gelangt, wie wenn der Fehler nicht aufgetaucht wäre. Somit können wir hier ein Auge zudrücken.

Bemerkung 5.3. Die vorab gegebene Funktion `schalttag()` hatte sich dieses Wissen bereits zu nutze gemacht.

Geek Tipp 8

Schreiben Sie Ihren Code wenn immer möglich mit etwas GRIPS.

[39] GRIPS = Generelles Return Im Prozedur-Start.

5.7 Leseaufgaben

5.7.1 a, b oder c zum Ersten

Was ist die Ausgabe (c = ...) des folgenden Programmes?

```
{ // aufrufender Block (Hauptprogramm)
  int a = 4;
  int b = 5;
  f(a, a, b);
}

// aufgerufene Methode f()
void f(int a, int b, int c)
{
  a = a + b;
  b = b + c;
  c = a + b;
  print("c␣=␣" + c);
}
```

5.7.2 x, y oder z

Was ist die Ausgabe des folgenden Programmes?

```
{ // aufrufender Block (Hauptprogramm)
  int x = 6;
  int y = 8;
  f(x, y, x);
}

// aufgerufene Methode f()
void f(int z, int y, int x)
{
  int r;
  x = x + y;
  y = x + 2*z;
  r = x + y + z;
  print("r␣=␣" + r);
}
```

5.7.3 a, b oder c zum Zweiten

Was ist die Ausgabe des folgenden Programmes?

```
{ // aufrufender Block (Hauptprogramm)
  int a = 7;
  int b = 4;
  f(a, a, b);
}

// aufgerufene Methoden f() und g()
void f(int a, int b, int c)
{
  int x, y, z;
  x = g(b, b, c);
  y = g(a, b, b);
  z = g(c, c, a);
  print("x=" + x);
  print("y=" + y);
  print("z=" + z);
}

int g(int b, int a, int c)
{
  return b + 2*a + 3*c;
}
```

Lösungen im Anhang (s. Kap. B auf Seite 189).

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 5. 1 Ungerade Zahlen
- Liste von Suchresultaten (Web-Code: rqte-f27b)
- 5. 2 RGB (1)
- 5. 3 RGB (2)
- 5. 4 Quersumme
- 5. 5 Abstand
- 5. 6 Dreiecksflächen
- 5. 7 Geometrie
- 5. 8 ggT
- 5. 9 Exponenten
- 5.11 Schaltjahre
- 5.13 Exponenten
- 5.14 Zahlenmuster (bzw. online «Anne lernt lesen» Web-Code: oukh-6gbt)
- 5.15 Frankengewinnspiel
- 5.16 Parameter vs. globale Variable

6 | Felder (Arrays)

Arrays (selten auch Felder, Reihungen oder Vektoren genannt) sind eindimensionale Tabellen von Variablen mit gleichem Typ. Auf die Elemente eines Arrays wird wahlweise mit Hilfe eines ganzzahligen Indexes zugegriffen. Wir sprechen von einem «random access[40]».

Einführungsbeispiel

Eine Methode `printDatum(tag, monat, jahr)`, welche ein Kalenderdatum in lesbarer Form ausgibt, könnte wie folgt aussehen:

```
void printDatum(int tag, int monat, int jahr) {
  print(tag + ".␣");
       if(1  == monat) print ("Januar" );
  else if(2  == monat) print ("Februar");
  else if(3  == monat) print ("Maerz" );
  else if(4  == monat) print ("April" );
  else if(5  == ...
  ...
  else if(12 == monat) print ("Dezember");
  print("␣" + jahr);
}
```

Für solche Fälle sind mit Vorteil Arrays zu benutzen:

```
String [] monatsNamen = {"Januar", "Februar", "Maerz", ...};
void printDatum(int tag, int monat, int jahr) {
  print(tag + ".␣" + monatsNamen[monat - 1] + "␣" + jahr);
}
```

Beachten Sie den um 1 verminderten Monatsindex (monat - 1). Dieser ist in JAVA wichtig, denn in JAVA werden Arrays immer von Null (0) an indexiert. Der Monat mit der Nummer 1 soll in diesem Beispiel (und nach allgemeinem Verständnis) jedoch der Januar sein.

[40] «random» bezeichnet hier nicht «zufällig» sondern eher «nach beliebiger Wahl».

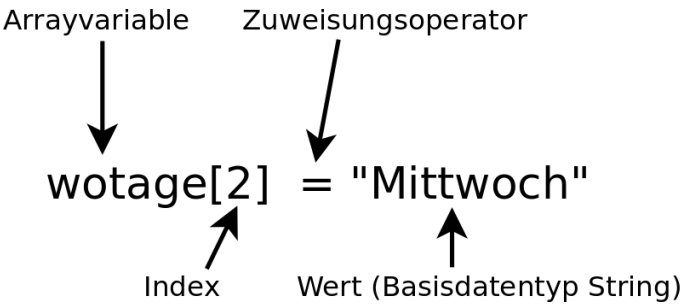
Buch Beispiel

Das Beispiel aus dem Buch ([GFG11] dort Seite 74) wird hier nochmals aufgegriffen. Zum besseren Verständnis nenne ich die Variable hier `wotage` (statt einfach `x`).

Index	wotage[0]	wotage[1]	wotage[2]	wotage[3]	wotage[4]	wotage[5]	wotage[6]
Wert	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	Samstag	Sonntag

Zugriff

Der Zugriff auf die einzelnen Komponenten geschieht mittels Index in eckigen Klammern ('[', ']').



Obiger Array kann in JAVA auf zwei Arten initialisiert werden:

a) durch direkte Array-Zuweisung

```
String[] wotage =  {"Montag"    , "Dienstag", "Mittwoch",
                    "Donnerstag", "Freitag" , "Samstag" ,
                    "Sonntag"};
```

b) schrittweise (einzeln)

```
String[] wotage;

wotage    = new String[7];

wotage[0] = "Montag"     ;
wotage[1] = "Dienstag"   ;
wotage[2] = "Mittwoch"   ;
wotage[3] = ...
...
```

Syntax von Arrays

Deklaration

Deklariert werden Arrays indem eckige Klammern hinter den Datentypen gestellt werden:

```
int[] s;
```

Achtung: In diesem Beispiel wird für die `int`-Elemente noch kein Speicherplatz geschaffen! Es wird erst deklariert, dass die Variable `s` später mit einem `int`-Array gefüllt werden darf.

Definition

Bevor wir die einzelnen `int`-Werte abfüllen können, müssen wir vom Betriebssystem Speicher anfordern. Dies geschieht mit dem `new`-Operator:

```
s = new int [5];
```

Mit dieser Definition wird Speicher für fünf `int`-Werte angelegt.

Defaultwerte

Im Gegensatz zu anderen Sprachen werden Arrays von Zahlen (hier `int`) je mit Null initialisiert. Es gilt unmittelbar nach dem Definieren des Arrays `s` also:

```
0 == s[0]
0 == s[1]
0 == s[2]
...
```

Indexieren

Der kleinste Index ist in JAVA immer 0 (Null). Somit ist beispielsweise der höchste Index des obigen Arrays `s` gleich 4.

Beispiele:

```
s[0] = 17;
s[1] = 9 * s[0];
s[4] = s[1] + s[2]; // s[2] == 0 (Defaultwert)
s[5] = 77;          // FEHLER: s indexiert nur von 0 bis 4!
```

Die einzelnen Komponenten (`s[0]`, `s[1]`, ...) werden je wie eigene Variable behandelt.

Länge

Arrays können ihre Länge (Anzahl der Elemente) mit dem Attribut `length` bekannt geben. In unserem Beispiel ist « `s.length == 5` ». Die Länge wird oft in Schleifen verwendet:

```
for(int i = 0; i < s.length; i++) {
  s[i] = i * i; // z. B. Quadratzahltabelle
}
```

6.1 Arrays und Schleifen (Iteration)

6.1.1 «for-each»

JAVA kennt eine sog. «for-each»-Schleife, mit folgender Syntax:

```
for(<Typ> <variable> : <Sammlung>) {
  <BLOCK> // mache etwas mit "variable"
}
```

Um zu zeigen, wie damit Arrays verwendet werden, können wir sogleich die Daten (Wochentage) aus den beiden obigen Einführungsbeispielen (s. Kap. 6 auf Seite 83) ausgeben:

```
for(String w : wotage) {
  System.out.println(w);
}
```

6.1.2 Index

Manchmal benötigt man den Index, also die Position des Elementes. Dies ist nun bei Arrays (aber auch bei ArrayLists und Lists) dank der bekannten Länge einfach zu bewerkstelligen:

```
int letzterIndex = wotage.length - 1;
for(int i = 0; i <= letzterIndex; i++) {
  System.out.println(i + ":␣" + wotage[i]);
}
```

6.2 Tabellen

Tabellen sind mehrdimensionale Arrays. Meistens handelt es sich um zweidimensionale Felder, wie wir sie von Tabellenkalkulationen kennen.

Am Einfachsten verstehen wir Tabellen mit Hilfe eines Beispiels.

Die Umsätze der drei Abteilungen (X, Y und Z) werden pro Quartal (Q1, Q2, Q3 und Q4) in einer Tabelle abgespeichert.

Umsätze in Tabellenform:

[q][0] = X	**[q][1] = Y**	**[q][2] = Z**	
5000	5600	1800	[0][a] = Q1
2100	3200	7300	[1][a] = Q2
1200	1900	4400	[2][a] = Q3
2600	9100	7700	[3][a] = Q4

Obiges wird in JAVA nun wie folgt implementiert:

```
int[][] umsaetze; // Tabelle = Array aus Arrays

// 4 Zeilen, 3 Spalten anlegen:
// Zeile   = vier Quartale
// Spalten = drei Abteilungen
umsaetze = new int[4][3];

// Indizes beginnen auch hier bei Null (0):
// Betrachten wir nun Q3 von Abteilung Y.
// 3. Zeile  hat Index 2: Q3
// 2. Spalte hat Index 1: Abt. Y
umsaetze[2][1] = 1900;
```

Wie JAVA dies intern technisch umsetzt, finden wir im Anhang (s. Kap. A.18.2 auf Seite 162).

Weitere Spezialitäten und Eigenheiten von Arrays und Tabellen im Anhang (s. Kap. A.18 auf Seite 162);

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 6. 1 Feldelemente aufsummieren
- Zähle vorkommen in Array (Web-Code: 07oi-q7s2)
- Suche Element in Array (Web-Code: t2hd-0qw5)
- 6. 3 Feld filtern (1)
- 6. 4 Feld filtern (2)
- Rotation (Web-Code: r5bd-t6kt)
- 6. 5 Feld filtern (3)
- 6. 7 Transponieren einer Matrix
- 6. 9 Schriftsetzer
- 6.10 n-ter Tag im Jahr
- 6.11 Freitag, der 13.
- 6.12 Waschautomat
- 6.13 Geldbetrag
- 6.14 Maximale Teilsumme
- 6.15 Tic-Tac-Toe
- 6.16 Springer auf dem Schachbrett
- 6.17 Exponententabelle
- 6.18 Gleichungssystem

7 | Zeichenketten

Die Verarbeitung von Zeichenketten kommt in den meisten Applikationen vor. Sei dies, z. B. um Resultate aus einer Datenbank anzuzeigen, eine Datei zu öffnen oder einfach, um eine Schaltfläche genauer zu spezifizieren.

Strings (also Zeichenketten) sind in der Regel nichts anderes als Arrays von Zeichen. In JAVA handelt es sich um Arrays auf dem Datentyp `char`. Da dies so häufig verwendet wird, existiert für Zeichenketten in JAVA ein eigenes Literal – wir schreiben z. B. einfach

```
String nameA = "Ali"  ;
String nameB = "Pedro";
String nameC = "Aisha";
```

7.1 Die Klasse `String`

Zeichenketten werden in JAVA mit der `String`-Klasse umgesetzt. Das String-Literal ist nichts anderes als eine Zeichenkette, die in (doppelten) Apostrophen (`"`) als Begrenzungszeichen eingefasst ist:

```
"Hallo␣Welt"
```

Wie man Zahlen einer `int`-Variable zuweisen kann, so kann man Strings auch `String`-Variablen zuweisen oder ganz generell als Ausdrücke (s. Kap. 1 auf Seite 17) verwenden:

```
String a, b;
a = "Hallo";
b = "Welt" ;
System.out.println(a + "␣" + b);
```

7.2 Verarbeitung von Zeichenketten in Java

Die Beschreibungen zu den folgenden standard-Problemstellungen finden Sie in «Programmieren lernen» [GFG11] in Kap. 7.1 auf Seite 88.

Literal	`"Hallo Welt"`
Variable	`name = "Meier";`
Ausgabe	`System.out.println("Hallo Welt");`

Einlesen	`String eingabe;`
	`Scanner sc = new Scanner(System.in);`
	`eingabe = sc.next();`

Länge	`String txt = "Hallo␣Welt";`
	`int    len;`
	`len = txt.length();`

Zeichen ermitteln	`char  ch = txt.charAt(5);` Strings sind wie Arrays ab 0 (Null) indexiert.

Zusammenfügen	`String a = "Hallo";`
	`String b = "Welt" ;`
	`String c = a + "␣" + b;`

Kopieren	`String a = "Hallo␣Welt" ;` `String b = new String(a);`

Suchen

Zunächst können wir nach einem einzelnen Zeichen suchen:

```
String txt = "Hallo␣Welt";
int    pos = txt.indexOf('W');
```

Eine andere Möglichkeit ist es, nach Teilstrings zu suchen.

```
if(txt.contains("Welt")) { ... }
```

Prüfen

Einfach ist es in JAVA zu prüfen, ob ein String auf eine vorgegebene Endung aufhört:

```
if(txt.endsWith("Welt")) {...}
```

Komplexe reguläre Ausdrücke sind mit JAVA auch möglich: Beginnt der Text mit `"H"`, enthält er (mind.) ein Leerzeichen und endet auf `"t"`?

```
if(txt.matches("H[^␣]*␣.*t")) {...}
```

Extrahieren

```
txt.substring(2, 5); -> "llo"
```

Achtung: Bei `substring()` ist die erste Position **inklusive** jedoch die zweite Position **exklusive**!

Den Code eines Zeichens zu ermitteln ist in JAVA geschenkt...

...einfach einer `int`-Variable zuweisen:

```
char zeichen = 'x';
int  code    = zeichen;
```

Zeichen des Codes: Hier ist ein *Casting* nötig.

```
zeichen = (char) code;
```

Ziffernfolgen in Zahlen verwandeln

```
String zahlStr1 = "2012";
String zahlStr2 = "3.14";

int    zahl1 = Integer.parseInt   (zahlStr1);
double zahl2 = Double .parseDouble(zahlStr2);
```

Zahlen in Ziffernfolgen (Strings) verwandeln

```
int    zahl       = 2012;
String zahlString = "" + zahl;
```

7.3 Zeichenketten und Dateien

Hier die wichtigsten JAVA Befehle, um mit Dateien umzugehen[41]:

- Auffinden von Dateien im Verzeichnisbaum:
  `File halloFile = new File("Hallo.txt");`
  `if(halloFile.exists()) { ... }`

- Öffnen, Schließen von Dateien und Zeilenweise daraus lesen:

```
try(BufferedReader br =
    new BufferedReader(new FileReader(halloFile)))
{
  String line;
  line = br.readLine();    // vorausgehendes Lesen
  while(null != line) {
    System.out.println("Gelesen:␣" + line);
    line = br.readLine(); // wiederholtes Lesen
  }
} catch(IOException iox) {
  System.err.println("Fehler:␣" + iox);
}
```

  Zum doppelten Lesen (vor-Lesen/nach-Lesen) gibt es in JAVA Abkürzungen, welche im Anhang beschrieben sind (s. Kap. A.20.2 auf Seite 170).

- Zeilenweise Schreiben in eine Datei:
  Mit dem Kommando `write();` können Strings, aber auch einzelne Zeichen geschrieben werden. Um eine Zeile zu schreiben, wird am Ende des Strings einfach das Zeilen-Ende-Symbol `\n` angefügt.

```
File out = new File("Ausgabe.txt");
try(FileWriter fw = new FileWriter(out)) {
  fw.write("Eine␣Zeile.\n");
  fw.write("Und␣noch␣eine␣Zeile.\n");
} catch (IOException e) {
  e.printStackTrace();
}
```

- Zeichenweise Lesen und Schreiben in JAVA funktioniert analog mit den Befehlen `read()` und `write()`.

- Das Anfügen an eine bestehende Datei ist im Anhang beschrieben (s. Kap. A.20.3 auf Seite 171).

[41] Weitere Infos im Anhang: (s. Kap. A.20 auf Seite 167)

7.4 Vergleichen von Strings in Java

Ein gängiger Fallstrick ist der Vergleich von Zeichenketten in JAVA. Wenn auch ganze Zahlen mit dem `==`-Operator verglichen werden, so funktioniert dies mit Zeichenketten nur bedingt. Der `==`-Operator für Zeichenketten (und andere Objekte) besagt in JAVA lediglich, ob es sich um die identischen Objekte handelt. Es sagt aber nichts aus, wenn zwei verschiedene Strings dieselbe Sequenz aus Zeichen aufweisen. So sind *Kurt Müller, Zürich; Entlisbergstr.* und *Kurt Müller, Zürich; Gottfried Kellerstr.* zwar weitgehend gleich, doch es sind **nicht dieselben**!

Strings werden in JAVA mit der Methode `equals()` auf inhaltliche Gleichheit untersucht:

```
String a, b;
a = ...;
b = ...;
if(a.equals(b)) {
  ...
}
```

Beachten Sie jedoch, dass die folgenden Vergleiche unterschiedliche Resultate liefern:

```
String a  = "xy";
String b1 = "xy";
String b2 = new String("xy");

if(a    ==   b1)    --> true
if(a    ==   b2)    --> false
if(a.equals(b2))    --> true
```

Weitere Infos im Anhang (s. Kap. A.19.5 auf Seite 166).

7.5 Plus Operator für String Objekte

Der `+`-Operator (Plus) auf Zahlen oder `char` angewendet, zählt die (binären) Werte zusammen. Ist jedoch mindestens einer der beiden Operanden ein String, so wird auch der andere als String betrachtet und die beiden Strings werden aneinandergereiht:

```
'#' + 4 + 2 + "␣!=␣" + '4' + 2
```

liefert nicht etwa « `#42 != 42` » sondern «41 != 42»

denn das Zeichen «#» hat den ASCII-Wert 35, 35 + 4 + 2 = 41 und der Plus-Operator assoziiert von links nach rechts! Das heißt, erst ab dem Auftreten des ersten Strings wird das Resultat in einen String umgewandelt. Der Datentyp `char` wird in JAVA **nicht** wie ein `String`, sondern wie ein `int` behandelt, wenn es ums Rechnen geht (`+`, `-`, `*`, `/`).

7.6 Unicode

JAVA verwendet für Strings den Datentypen `char` (16 Bit) um Unicode[42] Zeichen zu speichern. Damit können bis 65 000 Zeichen unterschieden werden[43].

Unicode ≠ UTF-8

Da in Files meist UTF-8 oder Latin (also 8-Bit-Zeichen) zur Textkodierung verwendet wird, so muss man sich trotz Unicode[44] bewusst sein, dass es zum Lesen bzw. Schreiben von Zeichen eine Umkodierung braucht.

7.6.1 `char`-Literale

Zu den Literalen ist noch zu erwähnen, dass die char-Literale auch für Strings eingesetzt werden. Die folgenden Escape-Sequenzen sind unterstützt:

- `\n` (new Line)
- `\b` (backspace)
- `\f` (formfeed)
- `\uABCD` (Unicode, «u» gefolgt von 4 Hex-Ziffern)[45]
- `\t` (tabulator)
- `\r` (carriage return)
- `\\` (backslash)
- `\'` (single quote)
- `\"` (double quote)

Merksatz für die existierenden *Escape*-Sequenzen:

Nur **B**eherrschung **f**ormatiert **u**nseren **T**ext **r**ichtig.

7.7 Bemerkung

Weitere Besonderheiten von Zeichenketten im Anhang (s. Kap. A.19 auf Seite 164).

[42] `http://www.unicode.org`

[43] Mit Escape-Sequenzen und diakritischen Zeichen («¨ », «ˆ» , ...) können noch viele weitere daraus zusammengesetzt werden.

[44] Auch wenn sich (im Falle von LATIN bzw. ASCII) diese Umkodierung z. B. lediglich auf das Weglassen der ersten 8 (bzw. 9) Bit beschränken kann!

[45] \u leitet eine 4-Stellige Hex-Zahl ein (= 16 Bit). Diese werden aber oft nicht als Escape-Sequenzen bezeichnet, sondern einfach als char-Literale.

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- Die Lösungen zur Aufgabe 7. 1 aus «Programmieren lernen» sind oben abgedruckt (s. Kap. 7.2 auf Seite 92).
- 7. 3 Wortweise umkehren
- 7. 5 Schriftlich addieren
- 7. 7 Cäsar
- 7. 8 Sekunden-Umwandlung
- 7. 9 IP-Adressen (1)
- 7.11 Dreiersystem (1)
- 7.12 Dreiersystem (2)
- 7.13 Nachkommastellen im Binärsystem
- 7.14 Deutsche Zahlnamen
- 7.16 7-Segment-Displays
- 7.17 ISBN/EAN
- 7.18 File-Statistik
- 7.20 Wortliste filtern
- 7.21 Wörter raten
- 7.22 Palindrome

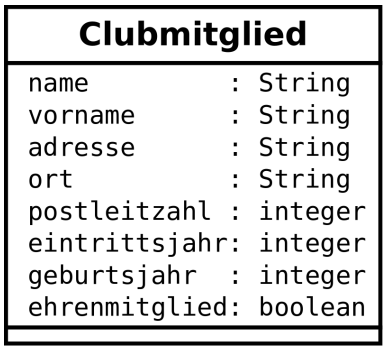
8 | Datenstrukturen und Sammelobjekte

Eigene Datentypen können aus den vorgegebenen Standardtypen (s. Kap. A.2 auf Seite 132) zusammengesetzt werden. Links sehen Sie die Attribute, rechts die zugehörige UML[46]-Notation.

Ein Clubmitglied weist z. B. die folgenden Attribute auf:

- Name, Vorname, Adresse, Ort: `String`
- Postleitzahl, Eintrittsjahr, Geburtsjahr: integer (in JAVA `int`)
- Ehrenmitglied: `boolean`

UML-Notation:



In JAVA sieht diese Datenstruktur wie folgt aus:

```
public class Clubmitglied {
  String   name          ;
  String   vorname       ;
  String   adresse       ;
  String   ort           ;

  int      postleitzahl  ;
  int      eintrittsjahr ;
  int      geburtsjahr   ;

  boolean  ehrenmitglied ;
}
```

Neue Clubmitglieder (Objekte) werden mit dem `new`-Operator geschaffen. Dieser holt sich vom System (Heap) genügend Speicher, um die Attributwerte festhalten zu können:

```
Clubmitglied c1, c2, c3;

c1 = new Clubmitglied();
c2 = new Clubmitglied();
c3 = new Clubmitglied();
```

[46] UML=Unified Modelling Language [GF14].

8.1 Neuer Datentyp

Datentypen, welche wie oben beschrieben wurden, können nun genauso als Variable verwendet werden, wie alle anderen auch. Im folgenden Beispiel wird eine Adresse gleich bei einer Person als Datentyp eingesetzt:

```
public class Adresse {
  String  strassenname;
  int     nr;
}

public class Person {
  String  name     ;
  String  vorname  ;
  int     jahrgang ;
  Adresse adresse  ;
}
```

8.1.1 Zugriff

Auf die Attribute (`name` , `vorname` , ...) wird nun analog zu den bekannten Attributen (`.length` , ...) mit dem `.`-Operator zugegriffen:

```
Person p;
p = new Person();
p.name = "Freimann";
System.out.println("Familienname:␣" + p.name);
```

Details zum `.`-Operator finden Sie in [GF14]; dort im Kapitel 13.4.

Aufgabe 8.1 «Personen Erfassen» Schreiben Sie hierzu ein JAVA-Programm, das zwei Personen erfasst und deren Daten inklusive **alter** (nicht Jahrgang) ausgibt.

8.2 Sammelobjekte

Spannend ist es, solche Objekte in Sammelobjekten – sog. Containern – zu speichern. Hier gibt es natürlich die Arrays. Doch Arrays haben trotz ihrer Effizienz doch einige Nachteile[47]. Daher bietet JAVA im Paket `java.util` weitere Sammelobjekte an, von denen die wichtigsten hier aufgelistet sind.

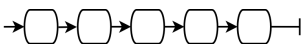
Collections werden verwendet, um Elemente hinzuzufügen um diese später wieder zu suchen oder Operationen auf allen Elementen durchzuführen. Die wichtigsten Methoden von *Collections* sind `add()`, `contains()`, `remove()` und `size()`.

- **List**: Die Elemente einer *List* sind linear geordnet, d. h. in der Reihenfolge, wie sie eingefügt wurden. Meist wird die `ArrayList` verwendet.

```
List<String> namen;
namen = new ArrayList<String>();
namen.add("Meier"  );
namen.add("Mueller");
namen.add("Keller" );
```

  Beispiel einer List ist die `ArrayList` (s. Kap. 8.2.2 auf Seite 104).

- Linked**List**: Geordnete Liste, bei der am Anfang und am Ende hinzugefügt, abgeholt und gelöscht werden kann. Somit kann eine LinkedList auch als Queue (First-in/First-out Warteschlange) oder Stack (First-in/Last-out Kellerspeicher) verwendet werden. Das Einfügen in der Mitte geht sehr rasch, da nur zwei Referenzvariable (Pointer) gesetzt werden müssen. Zugriff via Index und das Auffinden einzelner Elemente ist jedoch zeitaufwändig.



  Im folgenden Beispiel wird ein Element am Anfang der Liste gelöscht:

```
List<String> spiele;
spiele = new LinkedList<String>();
spiele.add("Fang␣den␣Hut");
spiele.add("Go"          );
spiele.add("Dame"        );
spiele.add("Poker"       );
spiele.add("Schach"      );
spiele.remove(0);       // -> geht rasch!

// nun alle ausgeben:
for(String spiel: spiele) {
  System.out.println(spiel);
}
```

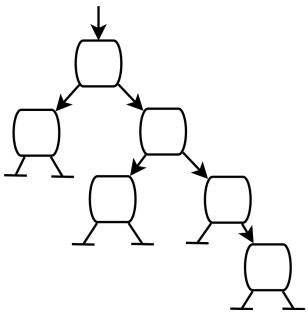
[47] Der größte Nachteil von Arrays ist die Unveränderbarkeit seiner Länge.

- (Hash)**Map** (Assoziative Arrays): Speichern der Objekte und Auffinden mit einem Schlüssel (Key).
  Beispiel Bücherliste:

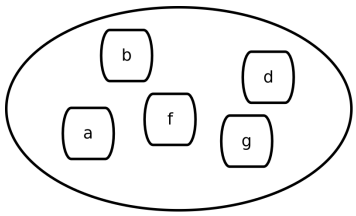
978-3-280-04066-9	Programmieren lernen
3-411-04011-4	Die deutsche Rechtschreibung
0-201-89683-4	The Art of Computer Programming
3-8266-0883-6	Shell-Skript Programmierung

  Details zur JAVA-`HashMap`: (s. Kap. 8.2.3 auf Seite 104)

- **Tree**: Baumstrukturen erlauben ein rasches, sortiertes Einfügen und Suchen. Zugriff via Index ist ähnlich aufwändig wie bei Listen.



- **Set** = Menge: Elemente können nur einmal angefügt werden.



  In JAVA bieten sich die beiden Sammelcontainer `TreeSet` und `HashSet` an. `TreeSet` fügt die Elemente automatisch sortiert ein (implementiert also `SortedSet`). Details (s. Kap. 8.2.4 auf Seite 105).

8.2.1 Beispiel Listen

Als konkretes Beispiel von Sammelobjekten wird hier die abstrakte Liste aus JAVA angegeben (`java.util.List`):

```
List<String> lst;
lst = Arrays.asList("Montag"    , "Dienstag", "Mittwoch",
                    "Donnerstag", "Freitag" , "Samstag" ,
                    "Sonntag");
```

Listen werden in Schleifen auf die Selbe Weise wie Arrays (s. Kap. 6.1 auf Seite 86) durchaufen:

```
for(String w: lst) {
  System.out.println(w);
}
```

8.2.2 ArrayList

Der am häufigsten verwendete Sammelcontainer (Collection) in JAVA ist die `ArrayList`. Die `ArrayList` hat im Gegensatz zu Arrays den Vorteil, dass Sie dynamisch erweiterbar sind. Mit anderen Worten: Der Programmierer muss sich nicht von vornherein auf eine Maximalgröße festlegen.

Eine Liste von Punkten (`Point`) kann z. B. mit einem `ArrayList`-Objekt folgendermaßen kurz und robust geschrieben werden:

```
ArrayList<Point> points = new ArrayList<Point>();
.... // Eintragen der Elemente in de Liste...
while(hatNochPunkte()) {
  points.add(naechsterPunkt());
}
```

8.2.3 HashMap

Als weiteres Beispiel wollen wir alle versicherten Personen einer mittleren[48] Versicherungsgesellschaft in einer Datenstruktur speichern. Die AHV Nummer als Index bietet sich an und soll gleich als Suchschlüssel verwendet werden. Arrays sind hier eine ganz schlechte Alternative. `ArrayLists` sind besser, jedoch das Suchen kann sich als zeitaufwändig herausstellen. Mit `ArrayLists` könnte man die Elemente nach Suchschlüssel (hier der AHV Nummer) sortieren und dann eine binäre Suche starten.

Das Einfügen von Elementen am Anfang oder in der Mitte wirkt sich bei `ArrayLists` ungünstig auf die Performance aus.

Eine Lösung zu obigem Problem ist die `Map`.

```
HashMap map = new HashMap();
map.put(ahv1, person1);
map.put(ahv2, person2);
//...
p = (Person) map.get("407.266.244.44");
```

Die `HashMap` speichert die Objekte in kurzen und rasch auffindbaren linearen Listen.

[48] 1 000 000 mittelgroße Datensätze können problemlos im RAM eines modernen Laptops verarbeitet werden.

8.2.4 Set

Ersetzen Sie in folgendem Code «ArrayList» durch «TreeSet» und besprechen Sie die Veränderungen in der Ausgabe. Zur Erinnerung: Ein `Set` (zu Deutsch eine Menge) kann jedes Element nur einmal enthalten.

```
import java.util.*;

public class NamensListe {
  public static void main(String[] args) {
    new NamensListe().top();
  }

  Collection<String> namensListe = new ArrayList<String>();

  void top() {
    einlesenNamen();
    ausgabeNamensListe();
  }

  void ausgabeNamensListe() {
    for(String str : namensListe) {
      System.out.println(str);
    }
  }

  Scanner sc = new Scanner(System.in);
  private void einlesenNamen() {
    while(true) {
       System.out.println("Bitte␣Namen␣eingeben␣(od.␣ENDE):␣");
       String name = sc.nextLine();
       if("ENDE".equals(name)) {
           return; // Abbruch, falls Anwender "ENDE" eintippt.
       }
       namensListe.add(name);
    }
  }
}  // end of class NamensListe
```

Weitere Infos zum JAVA *Collection Framework* findet sich im Anhang (s. Kap. A.21 auf Seite 179).

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 8. 4 Personen sortieren
- 8. 5 Ringpuffer mit Arrays

9 | Algorithmen

Algorithmen werden vorwiegend unabhängig von einer Sprache formuliert. Hier zeige ich einzig die Möglichkeit die Aufgabe «Heron» mit JAVA zu lösen:

```
float zahl = 5.0f;
float wurzel;

wurzel = Math.sqrt(zahl);
```

Wie Sie erkennen, ist der Algorithmus zum Wurzelziehen in JAVA bereits implementiert.

Viele Algorithmen, welche Programmierer immer und immer wieder selbst gelöst hatten, sind heute in die Programmiersprachen eingebaut. Dazu gehören nicht nur die mathematischen Berechnungen wie obige Wurzel, trigonometrische Funktionen und dergleichen. Auch das Sortieren von Objekten, das Erstellen von Eingabefeldern bis hin zum Schattenwurf dreidimensionaler Objekte gehört heute zum besseren Programmiergerüst.

Bemerkung 9.1. Wir lösen hier (in vergangenen und folgenden Aufgaben) vorwiegend Probleme, welche schon gelöst sind. In der Praxis wird dies wohl kaum getan. Viele dieser Problemstellungen sind einfacher mit einer Tabellenkalkulation oder mit einem Taschenrechner lösbar.

Dennoch: Zum Lernen gehört es dazu, viel zu üben und zu erkennen, welche Denkarbeit und welche Konzepte hinter einer bestehenden Lösung stecken.

Wir lernen im Mathe-Unterricht das Ziehen einer Wurzel oder das Auflösen eines Gleichungssystems mit mehreren Unbekannten ja auch nicht nur zum Spaß **von Hand**.[49] Es geht darum, bestehende Verfahren zu kennen und daraus zu lernen; nur so ist es möglich, neue, bessere Algorithmen zu entwickeln.

[49] Wurzelziehen und Gleichungssysteme Lösen können Computer heute viel schneller; mehr noch: Haben wir keine Fehler in der Eingabe, so sind die Computerlösungen meistens auch fehlerfrei. Siehe z. B. `http://www.wolframalpha.com`.

9.1 Leseaufgaben

Das Lesen von Programmcode fördert das exakte Vorgehen und – sofern der Code sauber geschrieben ist – zeigt gleichzeitig auf, wie Code geschrieben werden sollte.

Die folgenden Übungen (9.1.1 - 9.1.7) sind immer nach demselben Muster zu lösen:

- Wählen Sie einfache Parameter (z. B. 0, 1, 2, -1).
- Zeichnen Sie immer eine Wertetabelle.
- Gehen Sie Schrittweise und exakt vor. Der Zeigefinger der linken[50] Hand dient hier als Programmzeiger, also dieses Register, das auf den nächsten auszuführenden Befehl zeigt.

  Die rechte Hand verwenden Sie, um die Wertetabelle nachzuführen.
- Machen Sie eine erste Aussage, was das Programm tun könnte.
- Wählen Sie komplexere Parameter (z. B. 7, 11, 16, -0.5, 3.844, ...).
- Überprüfen Sie (in gleichem Schrittweisen Vorgehen), ob Ihre Hypothese zutrifft.
- Optional: Belegen Sie die Hypothese.

9.1.1 Selektion

```
int was(int a, int b, int c)
{
  int d = a;
  if(b < d) {
    d = b;
  }
  if(d > c) {
    d = c;
  }
  return d;
}
```

[50] Für Linkshänder: «Der Zeigefinger der **rechten** Hand»

9.1.2 Iteration (Schleife)

```
int was(int a)
{
  int i = 0;
  int j = 1;
  int k = 0;
  while ( i < a) {
    k = k + j;
    j = j + 2;
    i = i + 1;
  }
  return k;
}
```

9.1.3 Selektion und Iteration

```
int was(int a)
{
  if(a < 0)
  {
    print("Fehler, a muss >= Null sein");
    return 0;
  }
  int b = 0;
  while(a > 0)
  {
    b = (10 * b);
    b = b + (a % 10);   // %: bedeutet hier modulo = Divisionsrest
    a = a / 10;         // Ganzzahlige Division (64 / 10 ergibt 6).
  }
  return b;
}
```

9.1.4 Zahlenspielerei

```
int was(int a)
{
  int b = 0;
  int c = 1;
  int d = 0;
  while(a > 0) {
                  // % == Divisionsrest
                  // (z. B. 243 % 10 == 3):
    d = a % 10;
    b = c * 11 * d + b;
    c = 100 * c;
    a = (a - d) / 10;
  }
  return b;
}
```

9.1.5 Fuß-gesteuerte Schleife

```
int was(int a)
{
  int b = 2;
  int c;
  if(0 == a) {
    return a;
  }
  do {
    c = a % b;
    if(0 == c) {
      return b;
    }
    b = b + 1;
  } while(b < a);
  return a;
}
```

9.1.6 Nur für Spieler ;-)

```
boolean was(int arr[]) { // Array aus 5 Integers [0..4]
  int r1 = arr[0];
  int r2 = r1;
  for (int i = 1; i < 5; i++) {
    if(arr[i]  != r1) {
      r2 = arr[i];
    }
  }
  if(r1 == r2) {
    return false;
  }
  int c1 = 0;
  int c2 = 0;
  for(int i = 0; i < 5; i++) {
    if(arr[i] == r1) {
      c1 ++;
    }
    if(arr[i] == r2) {
      c2 ++;
    }
  }
  return (2 == c1 && 3 == c2) || (3 == c1 && 2 == c2);
}
```

9.1.7 Ein sehr alter Algorithmus

```
double was(double a) {
  double aj, d, dx;
  double ai = a;
  if (a <= 0.0) {
    System.out.println("Nur␣positive␣Zahlen!");
    return 0.0;
  }
  do {
    aj = ai;
    d = (aj * aj - a) / 2.0 / aj;
    ai = aj - d;
    dx = aj - ai;
  } while ((dx * dx) > 0.00001);
  return ai;
}
```

Die Lösungen zu diesen Aufgaben finden wir im Anhang (s. Kap. B auf Seite 189).

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 9. 1 Heron
- 9. 2 Gray Code
- 9. 3 Primzahlen
- 9. 4 Primzahlsieb des Eratosthenes
- 9. 5 Römische Zahlen
- 9. 6 Wörter suchen
- 9. 7 Lexikographisch sortieren
- Kürzester Weg (Web-Code: inyb-yc9q)
- Das Tortenproblem (Web-Code: 8ijp-jc7u)

10 | Simulationen

JAVA wurde nicht wegen seiner Syntax bekannt. JAVA machte sich dank seiner Hilfsklassen und seiner enorm großen API populär. Für alle gängigen Probleme hat JAVA eine Lösung bereit, wie auch z. B. für das Erzeugen sog. Pseudozufallszahlen:

10.1 Random

Wer seine eigenen Pseudozufallszahlen braucht, kann Aufgabe 10.1 aus [GFG11] lösen. Das dort beschriebene Verfahren der «linearen Kongruenzmethode» funktioniert in allen Programmiersprachen. Doch JAVA bringt schon mächtige Zufallszahl-Implementationen mit sich:

Pseudozufallszahlen können in JAVA auf zwei Arten gefunden werden. Die einfache und schnelle Art ist die Methode `Math.random()`, die eine Zahl zwischen 0 und 1 liefert[51].

Um z. B. Prüfungsnoten zu bestimmen kann ein Lehrer nun wie folgt vorgehen:

```
int note = (int) (Math.random() * 6 + 1);
```

Eine zweite Variante ist das Verwenden eines Objektes der Klasse `java.util.Random`. Hiermit können häufig verwendete Zufallsgrößen (`int`, `long`, `boolean`) einfach erzeugt werden.

Ein weiterer Vorteil der Klasse `Random` ist die Methode `setSeed()`. Diese erlaubt es, in Testsimulationen immer wieder dieselbe Zufallssequenz zu erzeugen. Die Methode `nextGaussian()` liefert eine normal verteilte Zufallszahl mit Mittelwert 0.0 und Standardabweichung 1.0.

[51] Null ist dabei eingeschlossen, 1 ist ausgeschlossen. Mathematisch: [0, 1)

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

- 10. 1 Lineare Kongruenzmethode
- 10. 2 Ganze Zufallszahlen
- 10. 3 Würfel (1)
- 10. 4 Casino (Würfel 2)
- 10. 5 Web-Code
- 10. 6 Farbstiftspitzen
- 10. 7 Nummern-Ratespiel
- 10. 9 Poker verteilen
- 10.10 Geburtstagszwillinge
- 10.12 Reisegruppe
- 10.13 25 im Quadrat
- 10.14 Magisches Quadrat
- 10.15 Socken
- 10.16 Quintenzirkel
- 10.17 Neuweltkamele
- 10.19 Umschütten
- 10.20 Monte-Carlo-Methode zu Bestimmung von Pi
- 10.21 Räuber-Beute-Verhalten (Lotka/Volterra)
- 10.22 Conways Game of Life
- 10.24 Testdaten mit Reservoir (Sampling)
- Mit dem Auto, dem Zug, dem Fahrrad oder zu Fuß? (Web-Code: tfod-exee)
- Schülernoten (Web-Code: aj6w-74tw)

11 | Rekursion

Subroutinen, die sich selbst direkt oder indirekt aufrufen, nennt man **rekursiv**.
Zur Veranschaulichung habe ich zwei spezielle Beispiele herausgepickt.

11.1 Beispiel Fibonacci

Als Beispiel soll (auch in dieser Einführung) die Folge von Fibonacci[52] herhalten. Jedes neue Glied der Folge wird aus der Summe der beiden letzten Glieder gebildet (z. B. $8 = 3 + 5$).
Der Anfang der Folge lautet wie folgt:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
```

Natürlich kann das mit einer Schleife (iterativ) programmiert werden. Die Funktion `fib()` soll das Element an einer vorgegebenen Position ausgeben (z. B. `fib(7)` = 13).

Iterative Lösung

```
public int fib(int pos)
{
  if (pos < 3)
  {
    return 1;
  }
  int a = 1, b = 1;
  for(int i = 2; i < pos; i++)
  {
    int next = a + b;
    a = b;
    b = next;
  }
  return b;
}
```

Erklärung: In obigem Code bezeichnen `a`, `b` und `next` drei aufeinander folgende Glieder der Fibonacci-Reihe. Nach dem Berechnen von `next` werden `a` und `b` entsprechend neu «abgefüllt», sodass `next` jeweils nur als temporäre Variable einen gültigen Wert aufweist.

[52] Leonardo «Fibonacci» von Pisa (ca. 1180 – 1241) über das Wachstum einer Kaninchenpopulation (S. Wikipedia)

Rekursive Lösung

Rekursiv ist das Programm aber einfacher, intuitiver und somit weniger fehleranfällig und besser wartbar (leider aber auch weniger performant).

```
public int fib(int pos)
{
  if(pos < 3)
  {
    return 1;
  }
  return fib(pos - 2) + fib(pos - 1);
}
```

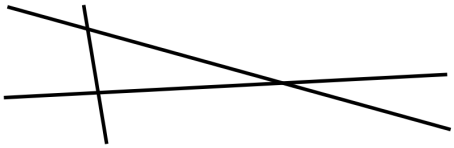
Wir sehen, dass die rekursive Programmierung viel weniger fehleranfällig ist. Der Code wird kurz und verständlich. In diesem Fibonacci-Beispiel hingegen ist der rekursive Code nicht wirklich performant (= rasch in der Ausführung). Die iterative Lösung (Beispiel davor) ist hier immens schneller, denn jede Zahl wird nur einmal berechnet. Wird in einer Rekursion die Funktion – wie hier – gar doppelt aufgerufen, so wächst die Laufzeit quadratisch an.

Lösung mit Formel

Am schnellsten (wenn auch am wenigsten verständlich) ist eine (geschlossene) mathematische Formel.

```
double phi     = 1.61803398874989484820458683436563811772030917 98;
double wurzel5 = 2.23606797749978969640917366873127623544061835 96;
public int fib(int pos)
{
  double result;
  result = Math.pow(phi, pos) / wurzel5;
  // Runden und zurueckgeben:
  return (int) (result + 0.5);
}
```

11.2 Beispiel Ebene zerschneiden



Das zweite Beispiel teilt eine Ebene mit beliebigen (also nicht parallelen) Geraden in Teilebenen. Eine einzige Gerade teilt die Ebene in zwei Teilebenen, zwei Geraden teilen die Ebene bereits in vier Teile, und kommt eine dritte Gerade hinzu, so kommen drei Teilebenen hinzu; die Ebene wird also insgesamt in sieben Teile geschnitten (s. obige Grafik).

Begründen Sie, warum bei der n-ten Gerade genau n Teilebenen hinzukommen.

Begründen und implementieren Sie die folgenden Algorithmen (iterativ, rekursiv, mit Formel):

Iterative Lösung

```
int teilebenen(int geraden)
{
  int teilebenen = 1;
  for(int i = 1; i <= geraden; i++)
  {
    teilebenen = teilebenen + i;
  }
  return teilebenen;
}
```

Rekursive Lösung

```
int teilebenen(int geraden)
{
  if(0 == geraden)
  {
    return 1;
  }
  return geraden + teilebenen(geraden - 1);
}
```

Lösung mit Formel

```
int teilebenen(int geraden)
{
  return 1 + geraden * (geraden + 1) / 2;
}
```

11.3 Vor- und Nachteile der Rekursion

Vorteile	Nachteile
• weniger Code • dadurch weniger Fehleranfällig • verständlicher • dadurch wartbarer	• evtl. **sehr** langsam in der Ausführung (Laufzeitverhalten nicht intuitiv) • möglicher Stack-Overflow (bei Vergessen der Abbruchbedingung) • schwer zu *debuggen*

Aufgaben aus «Programmieren lernen» [GFG11]

Zu den folgenden Beispielen aus der Aufgabensammlung «Programmieren lernen» [GFG11] existieren JAVA-Lösungen auf `http://www.programmieraufgaben.ch`:

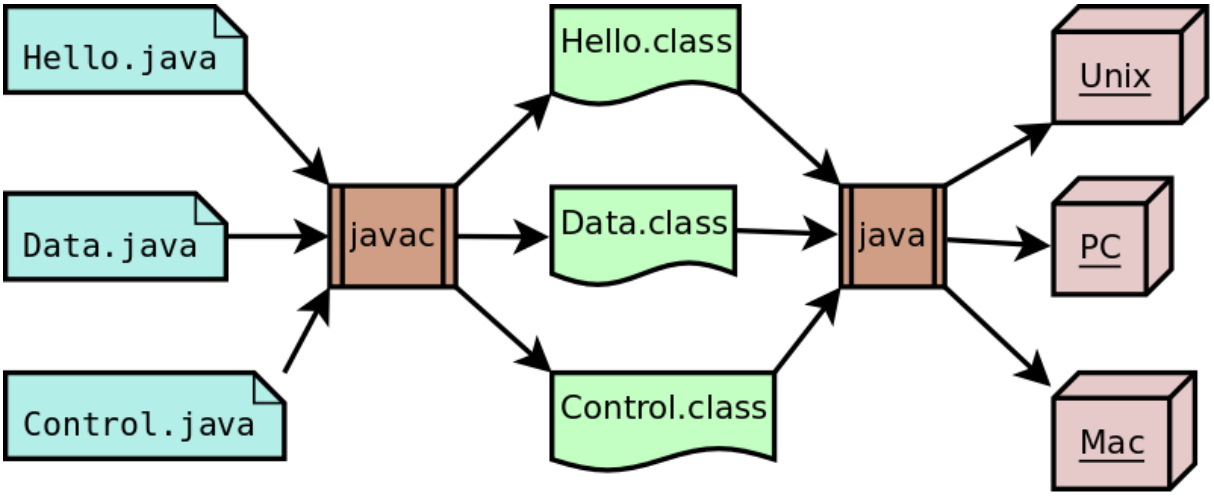
- 11. 1 Merge Sort
- 11. 2 Fibonacci
- 11. 3 ggT rekursiv
- 11. 4 Lotto
- 11. 5 Unterverzeichnisse ausgeben
- 11. 6 Türme von Hanoi
- 11. 7 Kamele beladen
- 11. 8 Schnurlängen
- 11. 9 Dichtestes Punktepaar
- 11.10 Malermeister Malcom

A | Anhang

«Was ich sonst noch alles sagen wollte»

A.1 Installation und erstes Java-Programm

Die JAVA-virtuelle Maschine (kurz JVM) ist heutzutage auf fast allen Rechnern vorinstalliert. Doch um JAVA zu programmieren, muss auch ein Übersetzungsprogramm, der sog. Compiler verwendet werden, der aus den JAVA-Quelltexten (dem eigentlichen Programmtext) den JAVA-Maschinencode produziert.



Der JAVA-Compiler, wie auch die JVM, sind Teile des «Java Developer Kits», kurz **JDK**.

A.1.1 Installation

Laden Sie den JAVA-Compiler ab der Webseite des Anbieters herunter:

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

Starten die das Installationsprogramm für Ihr Betriebssystem.

Die Installation kann einfach in einer Konsole (command-line, cmd, shell, BASH) überprüft werden. Dazu öffnen Sie die Konsole (Windows: Start -> Ausführen -> cmd; Mac/Unix/Linux: Terminal oder xterm).

Tippen Sie nun folgendes in die Konsole

```
javac
```

und bestätigen Sie die Eingabe mittels der Enter-Taste.

Ist JAVA korrekt installiert, so erscheint eine Meldung wie folgt:

```
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate a...
...
```

Sollte JAVA nicht korrekt installiert sein, so erscheint ein Meldung im folgenden Stil:

```
Der Befehl "javac" ist entweder falsch geschrieben oder
konnte nicht gefunden werden.
```

oder

```
No command 'javac' found, did you mean:
  ...
```

Falls JAVA **nicht** korrekt installiert ist, so könnten die folgenden Tipps weiterhelfen:

- Neustart des Computers nach der Installation von JAVA.
- Aufsuchen des Programms `javac` und den gefundenen Pfad in die Systemvariable `%PATH%` zu verankern. Danach Neustart der Konsole (`cmd`).
- Neuinstallation des JDK (evtl. wurde JVM anstelle JDK installiert).
- Suche im Internet nach «Installation JAVA PATH». Und geben Sie dabei noch den Namen Ihres Betriebssystems an (z. B. Linux, Ubuntu, MacOS, Windows, ...).

A.1.2 Erstes Programm

Tippen Sie das erste Programm aus dem Einleitungskapitel (S. Seite 15) mit einem Texteditor ab und speichern Sie dieses unter dem Namen « `MeinErstes.java` ».

Öffnen Sie die Konsole und springen Sie mit dem Kommando `cd` [53] ins Verzeichnis, wohin sie die Quelldatei `MeinErstes.java` gespeichert hatten.

Tippen Sie nun

```
javac MeinErstes.java
```

Sollte gleich wieder die Kommandozeile (Eingabeaufforderung) erscheinen, so ist alles gut gelaufen.

Bei Erfolg gibt der JAVA-Compiler (`javac`) keine Meldung aus. Allfällige Fehlermeldungen werden ausgegeben und müssen zuerst verbessert werden (s. Kap. A.1.3 auf Seite 130).

Werden keine Fehler gemeldet, so können Sie nun mit dem Befehl

```
java MeinErstes
```

das Programm starten. Nun sollte « `Hallo Welt` » in der Konsole erscheinen. Geschieht dies nicht, versuchen Sie auch:

```
java -cp . MeinErstes
```

Sollte « `Hallo Welt` » immer noch nicht erscheinen, lesen Sie auf der nächsten Seite weiter im Kapitel «Fehlerbehandlung».

[53] `cd` bedeutet so viel wie «change directory» oder eben «wechsle das Verzeichnis».

A.1.3 Fehlerbehandlung

Wie überall, wo die Gattung «Homo» am Werk ist, können Fehler passieren. Die folgende Aufzählung zeigt, welche Fehlermeldungen bei den ersten JAVA-Programmen auftauchen können und wie diese zu beheben sind.

- `javac: file not found`: JAVA wurde nicht korrekt installiert.
- `cannot find symbol`: Ein Name wurde falsch geschrieben. Zum Beispiel `string` statt `String`. (Achtung: JAVA unterscheidet zwischen Groß- und Kleinschreibung).
- `package system does not exist`: `system` wurde klein statt groß geschrieben.
- `reached end of file while parsing`: Könnte eine fehlende geschweifte Klammer sein.
- `; expected`: Es wird ein Strichpunkt erwartet. Oft wird dieses Ende einer Anweisung vergessen.
- `Hello.java:1: class Hallo is public, should be declared in a file named Hallo.java`: Der Dateiname entspricht nicht dem Klassennamen (Hello ist nicht gleich Hallo). Beachten Sie auch Groß- und Kleinschreibung bei Filenamen.
- `Exception in thread "main" java.lang.Error: Unresolved compilation problem`: IDE-Fehler z. B. eclipse. Es wurde versucht, das Programm zu starten, obschon es noch Fehler aufweist.

Eine häufige Fehlermeldung ist hier
`Exception in thread "main" java.lang.NoClassDefFoundError: ...` Das bedeutet, dass die virtuelle Maschine zwar richtig installiert wurde, aber dass die kompilierte Klasse (bzw. das `MeinErstes.class`-File) nicht gefunden werden konnte. Stellen Sie sicher, dass das aktuelle Verzeichnis die Datei `Hello.class` enthält (dies ist mit dem Kommando `ls` bzw. `dir` zu testen).

Ein weiterer häufiger Fehler ist die Meldung

`Exception in thread "main"`

`java.lang.NoSuchMethodError: main"`

Dies wiederum bedeutet, dass die `main()` Methode entweder falsch geschrieben wurde, nicht `static` ist oder falsche Parameter aufweist: Korrekt ist:

```
public static void main(String[] args)
```

Bei Problemen lesen Sie dieses Kapitel nochmals durch bzw. suchen Sie im Internet nach der exakten Fehlermeldung. Bei "www.google.com" wird man meistens fündig. Sollte es jedoch in dieser Anleitung Fehler haben, so melden Sie dies bitte direkt dem Autor `phi@gressly.ch`.

A.2 Datentypen

JAVA kennt die folgenden Datentypen[54]: Streng objektorientierte Sprachen kennen keine «primitive Datentypen»: JAVA wählte diesen Weg aufgrund der Performance (also der Rechengeschwindigkeit).

Datentyp	Beschreibung	Beispiele zugehöriger Java-Literale (Konstanten)
`boolean`	wahr oder falsch	`true`, `false`
`byte`	ganze Zahl -128 bis 127	`55`, `-83`
`short`	ganze Zahl (2 Byte), -32 768 bis 32 767	`1000`, `-4000`
`int`	ganze Zahl (4 Byte)	`-200_000`, `1_500_000_000`
`long`	ganze Zahl (8 Byte)	`7_000_000_000_000_000`
`float`	Dezimalbruch (4 Byte)	`4.3f`, `3.882f`, `-2.8e3f`
`double`	Dezimalbruch (8 Byte) Doppelte Genauigkeit	`-3.141592653589793`, `3.48e-150`
`char`	Zeichen	`'x'`, `'7'`, `'@'`, `'-'`, `'#'`, `\n`, `\uCAFE`, ...
`String`	Zeichenkette, Wort	`"Hallo Welt"`

[54] In JAVA sind alle Zahlen positiv oder negativ darstellbar; mit anderen Worten handelt es sich um «Vorzeichen behaftete» Zahlen

Bemerkungen:

- JAVA verwendet für die Ganzzahltypen (`byte`, `short`, `int`, `long`) das so genannte Zweierkomplement (S. [GFG11] Aufgabe 1.9). Die Zahl -2 wird also im Bitmuster ...11 1111 1110 dargestellt.
- Große Zahlen `2_000_000` dürfen in JAVA, der verbesserten Leserlichkeit zuliebe, mit Unterstrichen (`_`) versehen werden.
- Bemerkung zu `float` und `double`: Diese beiden Datentypen unterscheiden sich lediglich durch die Genauigkeit. `float` ist weniger genau, benutzt auch weniger Speicher und ist in der Regel auch schneller bei Berechnungen als der `double`. Sowohl bei `float`, wie auch bei `double` kann mit einem hinten angestellten `f` (bzw. `d`) der Datentyp spezifiziert werden.
- Ein JAVA-`char` ist ein Unicode-Zeichen, das mit 16 Bit kodiert ist. Mit `\n` wird das «Neue-Zeile» Symbol geschrieben, was für einen Zeilenumbruch sorgt. Unicode-Werte können auch Hexadezimal eingegeben werden. Dabei wird ein `\u` vorne hin gestellt. Beispiel `\u03C6` $= \varphi$
- Bemerkung zum `String`: der `String` ist kein primitiver JAVA-Datentyp. Dabei handelt es sich um eine Klasse, welche den Regeln der Objektorientierung folgt. Ich habe den String dennoch aufgeführt, denn auch der `String` besitzt in JAVA eigene Literale (z. B. `"Hallo Welt"`).

A.3 Die Kommandozeile (Shell bzw. CMD)

Wollen wir JAVA-Programme ohne eine integrierte Entwicklungsumgebung (wie *eclipse* oder *netbeans*) schreiben, so können wir auf ein Terminal (Linux: *shell*, MS-Windows: Kommandozeileneingabe (CMD)) zurückgreifen. Hier die wichtigsten Kommandos:

Titel	***Linux (BASH)***	***Windows (CMD)***	***Beschreibung***
Laufwerk wechseln	`cd /x/y`	`c:`	Unter Linux muss das Laufwerk bereits eingebunden sein, damit in ein anderes Laufwerk gewechselt werden kann. Viele Distributionen binden Laufwerke im Verzeichnis `/media` ein. In diesem Beispiel ist `/x/y` mit dem aktuellen Namen des Verzeichnisses auszutauschen. Unter Windows werden Laufwerksbuchstaben verwendet, gefolgt von einem Doppelpunkt (`a:` , `b:` , ...).
Verzeichnis wechseln	`cd <pfad>`	`cd <pfad>`	Mit cd (= change directory) wird in ein Unterverzeichnis gewechselt. <pfad> ist hier mit dem Namen des gewünschten Verzeichnisses auszutauschen. Sowohl Unix, wie auch Windows erlauben eine Ergänzung mit der TAB-Taste: Es reicht, die ersten Buchstaben einzugeben und danach die TAB-Taste zur Befehlsergänzung zu tippen.
Top Verzeichnis	`cd /`	`cd \`	Wechselt ins oberste Verzeichnis (root).
Übergeordnetes Verzeichnis	`cd ..`	`cd..`	Wechselt ins übergeordnete Verzeichnis; also ein Verzeichnis «hinauf».
List	`ls`	`dir`	Stelle den Inhalt des aktuellen Verzeichnisses als Liste dar.
Löschen	`rm <abc>`	`del <abc>`	Lösche die Datei `<abc>` .
Verzeichnis erstellen	`mkdir <abc>`	`mkdir <abc>`	Erstelle ein Verzeichnis mit Namen `<abc>` .
Stopp	`CTRL-c`	`CTRL-c`	Halte die CTRL (STRG)-Taste und drücke gleichzeitig den Buchstaben «c». Unterbreche die aktuelle Ausführung (z. B ein nicht endendes JAVA-Programm).
Ende	`exit`	`exit`	Verlasse die Kommandozeile.
Hilfe	`man <cmd>`	`help <cmd>`	Hilfe zum Kommando `<cmd>` anzeigen. `man` steht hierbei für «manual».

A.4 Strukturierter Java Code

JAVA ist eine weitgehend objektorientierte Sprache. Dass damit aber auch Software nach dem strukturierten Paradigma geschrieben werden kann, sollte nach der Lektüre dieses Büchleins klar sein (JAVA unterstützt alle fünf[55] prozeduralen Konzepte). JAVA macht es einem hier aber nicht leicht, denn für alles wird sofort eine Klasse und ein Objekt erwartet. Der folgende «JavaStarter» (Codebeispiel nächste Seite) kann benötigt werden, um kleine strukturierte Programme wie in [GFG11] zu schreiben.

Vergessen Sie aber nicht, den Namen « `JavaStarter` » durch Ihren Programmnamen zu ersetzen. Dies muss an drei Stellen geschehen: a) Der Filename, b) der Klassenname und c) im der `main()` -Methode beim Erstellen des Objektes mit `new()` .

[55] 1. Sequenz, 2. Selektion, 3. Iteration, 4. Unterprogramme, 5. alle Programmabläufe sind ohne «goto» (also «Jumps») möglich.

```
/**
 * Java Starter, damit Sie rein strukturiert loslegen koennen,
 * ohne sich um Objekte kuemmern zu muessen.
 * Aendern Sie den Filenamen,
 *             den Klassennamen und
 *             den top()-Aufruf.
 * "Ihre Dokumentation kommt hier ... "
 */
public class JavaStarter {    // <- hier anpassen
  public static void main(String[] args)
  {
    new JavaStarter().top();  // <- hier anpassen
  }


  // Hier gehoeren die globalen Variablen hin:
  String anrede;


  // Ebenso werden Funktionen hier definiert:
  void hallo()
  {
      System.out.println("Hallo␣" + anrede);
  }


  // Hier ist der Haupt-Einstiegspunkt:
  void top()
  {
    // Hier ist Platz fuer lokale Variable:
    String temporaer = "Welt";

    // Hier folgen die Anweisungen:
    anrede = "Universum";
    System.out.println("Hallo␣" + temporaer);
    hallo();
  }

} // end of class JavaStarter
```

Tippen Sie diesen Code ab, oder laden Sie ihn ab `http://www.gress.ly/javastarter` herunter.

A.5 Reservierte Wörter

Als Namen (Variable, Subroutinen, Klassen, ...) können Sie beinahe jedes Wort verwenden. Großbuchstaben, Kleinbuchstaben, der «Underscore» (Bodenstrich), sogar Währungssymbole ($, £, ...) und Ziffern sind erlaubt, wenn letztere nicht an erster Position stehen.

Die folgenden Wörter heißen Schlüsselwörter. Diese haben in JAVA jedoch eine spezielle Bedeutung und dürfen nicht als Bezeichner eingesetzt werden:

abstract	default	if	private	throw
assert	do	implements	protected	throws
boolean	double	import	public	transient
break	else	instanceof	return	try
byte	enum	interface	short	void
case	extends	int	static	volatile
catch	finally	long	strictfp	while
char	final	native	super	
class	flat	new	switch	
const	for	null	synchronized	
continue	goto	package	this	

Die Bedeutungen finden Sie hier:

`http://www.santis-training.ch/java/javasyntax/keywords`

A.6 Ein- und Ausgabe

A.6.1 Ausgabe

Die Ausgabe von Werten in JAVA auf der Konsole ist relativ einfach:

`System.out.println(...);`

Texte und Zahlen werden in der Regel so formatiert, wie wir es gerne hätten. Um Ausgaben aneinander zu fügen, verwenden wir den Plus-Operator:

```
int    x;
String s;
x = 4;
s = "Hallo␣Welt";
System.out.println("Ausgabe:␣" + x + s);
```

Zur Erinnerung ans Kapitel «Zeichenketten» (s. Kap. 7.5 auf Seite 95) sei hier nochmals ein einfacheres Zahlenbeispiel gegeben.
Da der Plus Operator (+) auch für Zahlen verwendet wird, ist beim ersten Betrachten nicht klar, was hier ausgegeben wird:

```
int x, y;
x = 4;
y = 5;
System.out.println(x + y + "␣ist␣" + x + y);
```

Zunächst werden die Zahlen (von links nach rechts) als Zahlen zusammengezählt und danach wird die Zeichenkette `" ist "` angefügt. Schließlich wird `x` und `y` je in eine Zeichenkette verwandelt und dem bestehenden hinzugefügt. Resultat: `9 ist 45`!

Erinnern Sie sich an die Klammerungsreihenfolge von Links nach rechts.
Der «+»-Operator fügt genau dann Zeichenketten zusammen, wenn einer der beiden Operanden eine Zeichenkette ist. Ansonsten werden einfach die Zahlen zusammengezählt.

A.6.2 Eingabe

Daten für die Programmausführung werden entweder aus persistenten Speichern (Datenbanken, Files, ...) geladen, sie werden beim Programmstart mitgegeben, oder der Benutzer kann sie zur Laufzeit eingeben oder laden lassen.

Alle Texte und Zahlen, die in einem Programm verwendet werden, gehören nicht in den Quellcode. Texte und Zahlen können sich bei einer weiteren Ausführung des Programmes verändern. Somit müsste ja das Programm angepasst werden

Hier einige Beispiele von Texten und Zahlen, die nicht in den Quellcode gehören:

- Beschriftungen von Buttons (Knöpfen) und Labels (Diese müssen Sprachunabhängig gehalten werden.)
- Ausgabetexte (z. B. «sehr gut» für Schülernote 6).
- Konstanten (z. B. Die Zahl π besser aus `java.lang.Math` verwenden statt 3.1415926 in den Code fix eingeben.)

Geek Tipp 9

Nur die Zahlen Null (0) und Eins (1) bzw. die Literale `false` und `true` dürfen im Code auftauchen. Andere Zahlen oder sprachabhängige Zeichenketten im Quelltext stören die Programmentwicklung permanent.
Persistente Daten werden aus Datenbanken oder Eigenschaftsdateien (sog. Property-Files) gelesen. Transiente Daten werden zur Laufzeit z. B. vom Anwender erfragt.

Das Einlesen von Text aus der Konsole gestaltet sich schon schwieriger. Gleich ein Beispiel:

```
import java.util.Scanner;

public class Eingabe {

  public static void main(String[] args)
  {
    new Eingabe().top();
  }

  String name ;
  int    alter;

  void top()
  {
    System.out.print("Geben␣Sie␣Ihren␣Namen␣ein:␣");
    name = new Scanner(System.in).nextLine();
    System.out.println("Hallo␣" + name);

    System.out.print("Wie␣alt␣sind␣Sie:␣");
    alter = new Scanner(System.in).nextInt();
    System.out.println("Gratuliere␣im␣Nachhinein␣zum␣"
                       + alter + ".");
  }

} // end of class Eingabe
```

Zu bemerken ist einerseits die «import»-Anweisung. `import java.util.Scanner` sagt sowohl dem Compiler, wie danach auch der JAVA-virtuellen Maschine, wo die Klasse `Scanner` zu finden ist.

Achtung: Je nach Konsole können nicht zwei `int` (also ganze Zahlen) hintereinander eingelesen werden, denn dazwischen befindet sich i. d. R. noch ein «Newline»-Zeichen. Es bietet sich hier an, immer nur den `nextLine()`-Ausdruck zu verwenden und die Inhalte (ob Zahl, ...) im Nachhinein zu unterscheiden. Dies ermöglicht auch eine saubere Fehlerbehandlung:

```
String eingabe;
int    x = 0  ;

eingabe = new Scanner(System.in).nextLine();
try {
  x = Integer.parseInt(eingabe);
} catch (NumberFormatException nfe) {
  System.out.println("Fehler␣in␣Zahl:␣" + eingabe);
}
```

Ein komplexeres Beispiel wie dies gemacht werden könnte, finden Sie hier:

`https://github.com/pheek/javaInput/blob/master/Input.java`

A.6.3 Eingabe- und Ausgabeumlenkung

Die Ausgabe `System.out`, sowie die Eingabe `System.in` können leicht in Dateien umgeleitet werden.
Betrachten wir eine kleine Modifikation des vorangehenden Programmes (Beachten Sie, dass der Scanner nun nur einmal mit `new` erstellt wird):

```
import java.util.Scanner;

public class EinAus {

  public static void main(String[] args)
  {
    new EinAus().top();
  }

  Scanner sc = new Scanner(System.in);

  String name       ;
  String alterString;
  int    alter      ;

  void top()
  {
    System.out.print("Geben␣Sie␣Ihren␣Namen␣ein:␣");
    name = sc.nextLine();
    System.out.println("Hallo␣" + name);

    System.out.print("Wie␣alt␣sind␣Sie:␣");
    alterString = sc.nextLine();
    alter = Integer.parseInt(alterString);
    System.out.println("Gratuliere␣im␣Nachhinein␣zum␣"
                       + alter + ".");
  }

} // end of class EinAus
```

Erstellen Sie zudem die folgende Datei (`daten.txt`)mit den gewünschten eingaben:

```
James Gosling
60
```

Starten Sie nun das obige JAVA-Programm aus der Konsole:

```
java EinAus < daten.txt
```

Die Daten werden nun nicht mehr aus von der Konsole, sondern aus der Datei `daten.txt` gelesen.

Natürlich können die Resultate auch in eine Datei umgeleitet werden, anstatt auf der Konsole ausgegeben zu werden:

```
java EinAus < daten.txt > ausgabe.txt
```

A.7 Weglassen der geschweiften Klammer

Die geschweiften Klammern (« `{` », « `}` ») bei `if`, `else` und `while` können in JAVA weggelassen werden, sofern es sich im auszuführenden Anweisungsblock um genau eine einzige Anweisung handelt.

Eine ausführliche Beschreibung dieser «Falle» habe ich hier veröffentlicht:

`http://www.santis-training.ch/java/pitfalls.php#c`

(Siehe dort im Kapitel «Iteration, Selektion und Strichpunkte».)

A.8 Anmerkungen zur Iteration

A.8.1 Warnung Strichpunkte

Implementieren Sie folgendes in JAVA und erklären Sie das Resultat:

```
int x = 5;
if(10 == x);
{
  System.out.println("if: x = " + x);
}
while(x >= 5);
{
  x = x - 1;
  System.out.println("while: x = " + x);
}
```

JAVA kennt – im Gegensatz zu anderen Sprachen[a] – **keinen** Strichpunkt nach der Bedingung. Dies gilt sowohl für die Selektion (`if`), wie auch für die Iteration (`while`). Ein Strichpunkt an dieser Stelle entspricht in JAVA einer leeren Anweisung.
Obiges `if` zwar als falsch ausgewiesen, jedoch wird die Print-Anweisung trotzdem ausgeführt (sie kommt ja erst nach dem `if` -Block).

Ebenso schlittert die `while` -Schleife in eine sog. Endlosschleife. Denn der angegebene Block wird nie durchlaufen, sondern es wird lediglich permanent die leere Anweisung (`;`) ausgeführt.

[a] Die Sprache PL/I verlangt nach der Bedingung einen Strichpunkt.

A.8.2 Schleifeninvarianten

Als Schleifeninvariante oder Iterationsinvariante bezeichnen wir Programmanweisungen innerhalb von Programmschleifen, falls diese ebenso gut außerhalb der Schleife geschehen könnten. Betrachten Sie folgenden Code:

```
int i = 1;
int k = 17;
while (i <= 10)
{
  int r = 5 * k + 3;
  int z = r + i;
  print (z);
}
```

In obigem Code wird die Zahl `r` jedes Mal neu berechnet, obschon sie von den Schleifenvariablen (`i` und `z`) unabhängig also invariant ist. Der Code kann klarer und auch performanter (=schneller in der Ausführung) wie folgt geschrieben werden:

```
int i = 1;
int k = 17;
int r = 5 * k + 3; // von i unabhaengig
while(i <= 10)
{
  int z = r + i;
  print (z);
}
```

Bemerkung: Ein guter Compiler kann das optimieren, sodass unter Umständen beide obigen Programmteile gleich schnell ablaufen werden – gehen Sie jedoch nicht ungeprüft davon aus!

A.8.3 Selektion als Spezialfall

Die Selektion kann als Spezialfall der Iteration aufgefasst werden. Zusammen mit der `while()`-Schleife und dem Konzept von Variablen kann ein `if()` wie folgt erzeugt werden:

```
boolean firstTime = true;
while(firstTime && bed())
{
  doSomething();
  firstTime = false;
}
```

Obiger Code ist identisch mit:

```
if(bed())
{
  doSomething();
}
```

A.9 Metasprache zur Darstellung der Grammatiken

Ich verwende hier eine vereinfachte EBNF[56]-Notation, um die JAVA-Syntax darzulegen. Die hier benutzte Notation ist zwar nicht mehr ganz korrekt, dafür jedoch etwas einfacher.

Die folgenden Symbole werden verwendet und beziehen sich auf das Zeichen oder den eben verwendeten Klammerausdruck:

*	Das Feld darf mehrfach verwendet werden, darf aber auch weggelassen werden.
+	Das Feld darf mehrfach verwendet werden, muss aber mindestens einmal aufgeführt werden.
[...]	Der Ausdruck innerhalb der eckigen Klammern ist optional.
?	Alternative Schreibweise für *optional*: Feld kommt maximal einmal vor, kann aber auch wegfallen.
|	Entweder das Symbol links oder das Symbol rechts des Striches wird eingesetzt.
<>	Der Textblock innerhalb der spitzen Klammern muss durch den effektiven Wert ersetzt werden. Beispiel Person :== «<Name>» « <Vorname>» wird zu «Hans» «Meyer».
(...)	Klammerausdruck. Die obigen Symbole werden rechts des Ausdrucks geschrieben und gelten dann für den gesamten Klammerausdruck.
fett	Muss exakt so abgeschrieben werden (z. B. <MODIFIER>+ `class` ...).

Beispiele

```
a*     : "", "a", "aa", "aaa", ...
(xy)+  : "xy", "xyxy", "xyxyxy", ...
x*y?   : y, xx, xxxxy, ...
```

[56] Erweiterte Backus-Naur-Form (s. Wikipedia)

A.10 Syntax von Java-Subroutinen (Methoden)

```
[<modifier>]* <return-type> <name> ( [<parameter>]* )
[throws <exception>+ ]
{
  <Block>
}
```

(Die Syntaxstruktur findet sich hier: (s. Kap. A.9 auf Seite 147))

Dabei haben die Felder folgende Bedeutungen:

<modifier>	:==	Modifizierer: public, protected, private, static, strictfp, native, abstract, final, synchronized.
<return-type>	:==	Rückgabewert: void | <Datentyp>
<name>	:==	Identifier: Name der Methode
<parameter>	:==	Übergabewerte: [final] <Datentyp> <Name> Mit Kommata (,) getrennt.
<exception>	:==	Exception Type: Name der Exceptionklassen Auch mit Kommata (,) getrennt.
<Block>	:==	Anweisungen

Bemerkung: In JAVA können Subroutinen nur innerhalb von Klassen stehen. Es gibt keine Deklaration von Subroutinen außerhalb von Klassen, aber auch nicht direkt innerhalb von anderen Subroutinen.

A.11 Überladen (Overloading)

In JAVA (wie den meisten modernen Sprachen) können Methodennamen mehrfach definiert werden.

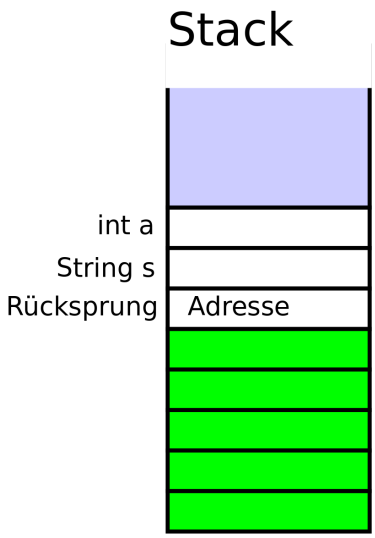
Dabei werden die Methoden alleine durch die Anzahl und Datentypen ihrer Parameter unterschieden. Dieses Verfahren ist unter dem Konzept «Überladen (Overloading)» oder auch «statischer Polymorphismus» bekannt.

Beispiel:

```
void loeschen(File        f                 );
void loeschen(Verzeichnis v                 );
void loeschen(Kundendaten kd, Kundenkartei k);
void loeschen(Durst       d                 );
```

A.12 Stack (Stapel)

Was geschieht denn eigentlich genau bei einem Subroutinen-Aufruf. Die technische Lösung bildet der Stapel (engl. Stack).




```
void sub(int a)
{
  String s = "a ist gleich " + a;
  System.out.println(s);
}
```

Beim Aufruf einer Subroutine ...

- ... werden die **Parameterwerte** (also die Argumente) auf den Stack kopiert und sind via Parameter (hier die Parameter-Variable `a`) zugreifbar,
- ... wird Platz für die **lokalen Variable** (hier `s`) geschaffen, damit die Subroutine auch temporäre Zwischenresultate speichern kann und
- ... wird zudem auch die **Rücksprungadresse** auf den Stack gelegt. Dies ist wichtig, damit das Programm bei Beenden der Subroutine auch weiß, wo im Hauptprogramm weitergefahren werden muss.

A.12.1 Das versteckte «`return`»

Die folgenden beiden Subroutinen sind absolut identisch:

```
void sub(int a)
{
  String s = "a␣hat␣den␣Wert␣" + a;
  System.out.println(s);
}
```

```
void sub(int a)
{
  String s = "a␣hat␣den␣Wert␣" + a;
  System.out.println(s);
  return;
}
```

Denn: Wenn wir die `return`-Anweisung am Ende einer Subroutine weglassen, so wird diese vom Compiler automatisch eingefügt.

Das `return` beendet die Subroutine automatisch, gibt den Speicher der temporären Variable auf dem Stack wieder frei und springt zum aufrufenden Programm zurück (daher wurde die Rücksprungadresse auch gespeichert).

Der Leserlichkeit halber wird diese abschließende `return`-Anweisung in der Praxis üblicherweise weggelassen.

A.13 Java-Klasse (Kompilationseinheit)

Jede Kompilationseinheit in JAVA beinhaltet in der Regel (genau) eine Klasse. Diese ist (abgesehen von Code-Kommentaren) nach dem folgenden Grundmuster (und in dieser Reihenfolge) aufgebaut:

[package]	genau eine Paketdeklaration, die aussagt, wo die Datei aufzufinden ist
[imports]	beliebig viele Angaben zu externen Referenzklassen und Paketen
[class]	normalerweise eine Klassendefinition

Dabei ist die Klasse (`class`) wie folgt aufgebaut[57]:

```
[<MODIFIER>]* class      <Klassenname>
              [extends    <Klassenname>]
              [implements <Klassenname>+]
{
  [member]*
}
```

A.13.1 Kommentare

An jeder Stelle[58] im Quelltext können Kommentare geschrieben werden. Diese dienen dem Verständnis des Codes und werden vom Compiler ignoriert.

- Alles was ab `/*` bis und mit `*/` steht, wird als Kommentar gesehen und gehört nicht zum ausführbaren Programmcode.
- Das Symbol `//` bezeichnet einen Kommentar: Alles bis zum Zeilenende wird vom Compiler ignoriert.

[57] Für die EBNF Notation (s. Kap. A.9 auf Seite 147)

[58] ... mit Ausnahme **innerhalb** von Operatoren (z. B. `++`) und Bezeichnern (z. B. Variablennamen) ...

A.13.2 Members einer Klasse

Innerhalb einer Klasse (`class`) können die folgenden *Klassenelemente* (Members) auftreten (wobei nur die ersten drei wirklich wichtig sind):

Member	Beschreibung	Beispiel	Bemerkung
Attribut	Objekt-Eigenschaft	`String name;`	Jedes Objekt hat seinen eigenen internen Zustand.

Methode	Ereignisse (Events), die auf die Objekte wirken können	`void add(float smd)`
		`{`
		`summe += smd;`
		`}`
			Ereignisse verändern typischerweise den internen Zustand (die Attribute) der Objekte.

Konstruktoren	Spezielle Methode, um Objekte zu erzeugen	`public Person () {...}`	Jedes Objekt wird durch einen Konstruktor (mittels `new`) erzeugt.
Statisches Attribut	Klasseneigenschaft	`static int obCount;`	Jede Klasse kann auch globale Variable aufweisen, die für alle Objekte verwendbar sind.
Statische Initialisierer	Code, der bei Klassenerzeugung ausgeführt werden soll	`static {...}`	Dieser Code wird beim Laden der Klasse ausgeführt. Ohne das Schlüsselwort `static` wird der Code bei der Objekterzeugung ausgeführt.
Dynamischer Initialisierer	Code, der nach `super()` , aber vor dem Rest des Konstruktors ausgeführt wird.	`{ ...}`	Dynamische Initialisierer werden selten gebraucht, da den Entwicklern oft nicht klar ist, wann diese ausgeführt werden. Finger davon lassen!
Statische Subroutinen	Klassenereignisse	`static int getYear()`	Werden meist für klassische Funktionen, die nicht von einem Objektzustand abhängig sind, verwendet.
Innere Klassen	Klassen innerhalb bestehender Klassen	`class Xyz {...}`	Beispiel: AHV Nummer als innere Klasse von Person.

1. Beispiel einer Kompilationseinheit:

```
public class Trampeltier extends Kamel implements Haustier
{
  float   gewicht;
  String  name   ;

  public Trampeltier(String name, Date geburt, float gewicht)
  {
      super(name, geburt);
      this.gewicht = gewicht;
  }

  float getGewicht()
  {
      return this.gewicht;
  }
}
```

2. Beispiel einer etwas komplizierteren Kompilationseinheit:

```
/* package */
package ch.programmieraufgaben;

/* imports */
import ch.programmieraufgaben.util.*;
import java.util.HashMap;

/* Klassendefinition der Klasse Kunde */
public class Kunde {
  String name, vorname;
  int kundennummer;

  public Kunde(String name, String vorname, int id)
  {
      this.name         = name;
      this.vorname      = vorname;
      this.kundennummer = id;
  }

  public String getName()
  {
      return name;
  }
}
```

Bemerkung A.1. Neben `class` sind auch `interface` oder `enum` als Inhalt einer Kompilationseinheit möglich. Zu `enum` gibt es ein eigenes Kapitel (s. Kap. A.15 auf Seite 157).

A.14 Variable

Variable bezeichnen Speicherstellen. An ihrer Stelle können (binär kodierte) Werte stehen. Jede Variable hat einen Namen (Bezeichner), einen Datentypen und allenfalls einen Wert. In JAVA kommen Variable an verschiedenen Stellen zum Einsatz:

Lokale Variable	Jeder Block (Subroutinen, Kontrollstrukturen, aber auch namenlose Blöcke) kann lokale Variable deklarieren. Wichtig: Die lokalen Variablen erhalten **keinen** Standardwert (Default). Sie müssen explizit definiert werden (z. B.: `int x; x = 7;`).
Parameter	Methoden und Konstruktoren können mit einer formalen Parameterliste versehen sein. Diese Variablen sind nur innerhalb der Methode sichtbar und leben nur solange (auf dem Stack) wie sich der ausführende Thread (Programmzeiger) innerhalb der Methode bzw. des Konstruktors befindet. Parameter haben keine Defaultwerte. Die Werte werden beim Aufruf der Methode oder des Konstruktors vom aufrufenden Programmteil übergeben. Genauer: Das aufrufende Programm wertet die Argumente aus, und die Werte (z. B. Zahlen, Referenzen) werden auf den Stack geschrieben.
Attribute	Variable, die innerhalb von Klassen, aber außerhalb von Methoden oder Konstruktoren deklariert werden, heißen (sofern nicht `static`) Attribute. Diese erhalten ihren Speicherplatz, sobald ein Objekt der entsprechenden Klasse (oder einer Subklasse) generiert wird (`new`). Attribute erhalten beim Generieren (sofern nicht anders definiert) ihren Defaultwert (z. B. 0 für `int`).
Klassenvariable	Mit `static` markierte Felder sind Klassenvariable. Diese Variablen können von allen Instanzen (Objekten) dieser Klasse verwendet werden und können im Gegensatz zu Attributen für verschiedene Objekte keine verschiedenen Werte annehmen. Pro Klasse existiert genau eine Speicherstelle für diese Variable, die von allen Objekten geteilt wird. Auch dürfen die statischen Methoden (Klassenfunktionen) auf diese Variable zugreifen. Egal wie viele Objekte es gibt, die Klassenvariable kommt nur einmal vor und ist in diesem Sinne global. In einer Interface-Deklaration braucht das Schlüsselwort `static` nicht angegeben zu werden. Die Variable erhält ihren Speicherplatz, sobald der *Classloader* die Klasse in die virtuelle Maschine lädt. Klassenvariable erhalten - sofern keine Definition angegeben ist - automatisch den Defaultwert.
Arraywerte	Arrays (selten auch Felder genannt) bestehen aus Variablen desselben Datentyps. Wenn ein Array generiert wird (`new A[<size>]`), erhalten alle Elemente Speicherplatz und ihren Defaultwert. Objektreferenzen werden mit `null` initialisiert.
Exception-Handler Parameter	Jeder Exceptionhandler (`catch(Exception e) {...}`) enthält ein Argument des Datentyps der entsprechenden Exception (Ausnahme). Sobald die Exception auftritt, wird ein Objekt der entsprechenden Klasse generiert. Dieses Objekt wird dem Argument übergeben und kann innerhalb des `catch` -Blocks benutzt werden.

A.15 Nominale Werte (`enum`)

«Eigene primitive Datentypen erstellen»

Oft kennt man von seinem Datentyp den Wertebereich nominal. Das heißt, der Wertebereich ist auf einige wenige Wörter oder Begriffe eingeschränkt (z. B. Wochentage, Ampelzustände, ...). Die erste der folgenden Möglichkeiten, um nominale Werte darzustellen, kennen wir bereits (Ganzzahltyp).

A.15.1 Darstellung als Ganzzahltyp

Bei dieser einfachen Implementation nominaler Werte wird jedem möglichen Wert eine ganze Zahl zugeordnet:

```
// Beschreibung : 0=Montag , 1= Dienstag , 2= Mittwoch , ...
int ferienanfang = 4; // Freitag
...
// Freitag:
if(4 == ferienanfang)
{
   ...
}
```

Nachteil: Ständiges Nachschlagen in der Dokumentation (Tabelle oder Programmkommentar) ist nötig. Zahlfehler werden vom Compiler nicht erkannt; m. a. W.: Es kann der Variable auch ein Wert zugeordnet werden, der in der Tabelle fehlt.

Daher bietet sich sofort das Arbeiten mit unveränderbaren Variablen an, sogenannten Konstanten:

A.15.2 Variable oder Konstante

Eine weitere Möglichkeit besteht darin, für jede vorkommende Zahl eine Variable (falls möglich eine Konstante, die in JAVA mit dem Schlüsselwort `final` gekennzeichnet werden) zuzuweisen:

```
final int MONTAG    = 0;
final int DIENSTAG  = 1;
final int MITTWOCH  = 2;
...

int ferienanfang = FREITAG;
```

Zwar müssen wir nun die Bedeutungen nicht permanent nachschlagen; ein Nachteil bleibt trotzdem: der Variable `ferienanfang` kann jeder `int`-Wert zugewiesen werden. Um das etwas zu umgehen, kann man auch mit Konstanten Strings arbeiten:

A.15.3 Konstante Zeichenketten

Die nun vorgestellte Variante unterscheidet sich zur vorangehenden dadurch, dass anstelle von ganzen Zahlen (`int`) Strings (also Zeichenketten) verwendet werden. Gleich ein Beispiel:

```
final String MONTAG    = "Montag"  ;
final String DIENSTAG = "Dienstag";
final String MITTWOCH = "Mittwoch";
...
String ferienanfang = FREITAG;
```

Vorteil: Eine künstliche Zuordnung zu den Zahlen fällt weg. Nachteil: Weiterhin kann jeder beliebige String der Variable `ferienanfang` zugewiesen werden.

A.15.4 Die technische Lösung

(Konstante Objekte und versteckte Konstruktoren)

Die Lösung: Damit keine anderen Objekte einer Variablen zugewiesen werden können, verwendet man in der Praxis konstante (finale) Objekte von Klassen, die keinen öffentlichen Konstruktor aufweisen:

```
public class Wochentag {
  public static final Wochentag MONTAG    = new Wochentag();
  public static final Wochentag DIENSTAG = new Wochentag();
  public static final Wochentag MITTWOCH = new Wochentag();
  ...
  private Wochentag() {}
} // end of class Wochentag
```

Verwendet werden diese Typen danach folgendermaßen:

```
Wochentag ferienanfang = Wochentag.FREITAG;
...
if(Wochentag.FREITAG == ferienanfang)
{
   ...
}
```

Liest sich doch schon besser, als `if(4 == ferienanfang) ...`

Damit nicht jeder Programmierer sich all diese Gedanken wieder machen muss, hat JAVA seit Version 1.5 so genannte Enumerationstypen (`enum`) eingeführt, welche sich technisch genauso verhalten, wie die eben gezeigte Lösung; sie sind aber einfacher zu benutzen. Verwenden Sie wenn immer möglich die folgende Vorgehensweise:

A.15.5 Enumerationstypen (Java-`enum`)

Schreiben Sie in der Kompilationseinheit `Wochentag.java` nur noch folgendes:

```
enum Wochentag {MONTAG, DIENSTAG, MITTWOCH, ...}
```

Verwendet wird dies dann genau so wie im Beispiel vorhin. Nun ist Wochentag ein eigener Datentyp und wird von der Syntax wie alle anderen Datentypen verwendet (siehe unten).

Im Gegensatz zur Lösung mit `int` oder mit `String` kann nun kein ungültiger mehr Wert zugewiesen werden und den Werten sieht man ihre Bedeutung direkt an (z. B. `Wochentag.FREITAG`).

Beispiel A.1. Beispiel: Ich will ausschließlich einen der vier folgenden Werte für meine Variable zulassen

0 (= Nord), 1 (= Ost), 2 (= Süd), 3 (= West):

so schreibe ich in einer Kompilationseinheit namens `Richtung.java` folgendes:

```
enum Richtung {NORD, OST, SUED, WEST}
```

Und verwende dies wie folgt:

```
Richtung sonnenUntergang = Richtung.WEST;
```

Geek Tipp 10

Verwende immer Enumerationstypen (enum) für nominale Werte!

A.16 Bit-Maskierung (Mengen)

Oftmals genügt ein nominaler Wert nicht, um unsere Daten darzustellen. Als Beispiel soll ein Kamel in einer Karawane dienen. Ein Kamel kann Wasserträger, Verpflegungsträger, Gepäckträger oder Personenträger sein. Es kann aber auch jede Kombination dieser Eigenschaften aufweisen. So kann die Variable Last nun einfach mit 4 Bit abgespeichert werden. Jedes Bit entspricht dabei einer der erwähnten Lasten:

```
Bit 1: Wassertraeger
Bit 2: Verpflegungstraeger
Bit 3: Gepaecktraeger
Bit 4: Personentraeger
```

Ist ein Kamel innerhalb einer Karawane nun Wasser-, Gepäck- und Personenträger (aber kein Verpflegungsträger), so schreiben wir dafür das Bitmuster 1101 in eine Variable. Bemerke: Das erste Bit (Bit mit Nummer 1) steht ganz rechts.

Mit sogenannten Masken-Konstanten können wir einem Kamel eine Eigenschaft hinzufügen, eine Eigenschaft wegnehmen oder einfach auf eine Eigenschaft prüfen. Hier die vier benötigten Masken-Konstanten:

```
int WASSER_TRAEGER        = 1; // Bit 1 = 0000 0001
int VERPFLEGUNGS_TRAEGER  = 2; // Bit 2 = 0000 0010
int GEPAECK_TRAEGER       = 4; // Bit 3 = 0000 0100
int PERSONEN_TRAEGER      = 8; // Bit 4 = 0000 1000

int typ; // Kombination aus obigen Bitwerten
```

Hinzufügen einer Eigenschaft:

```
typ = typ | GEPAECK_TRAEGER;
```

Entfernen einer Eigenschaft:

```
typ = typ & (~ GEPAECK_TRAEGER);
```

Prüfen einer Eigenschaft

a) auf Vorkommen:

```
if(GEPAECK_TRAEGER == (typ & GEPAECK_TRAEGER)) {...}
```

b) auf Nichtübereinstimmung:

```
if(GEPAECK_TRAEGER != (typ & GEPAECK_TRAEGER)) {...}
```

A.17 Abfragen im *Geek-Style*

Ein Computer-Geek ist ein «Besserwisser» in der Art, dass er technische Sachverhalte zwar tatsächlich meist besser weiß, aber auch permanent an seinem Informatik-Wissen zweifelt und dieses in Frage stellt. Insofern muss der folgende Geek-Style auch immer wieder in Frage gestellt werden...

Beachten und diskutieren Sie die folgenden Abfragearten. Verinnerlichen Sie sich den «Geek-Style»:

Titel		***Zahlen***	***Bool'sche Werte***
Standard		`if(x == 4)`	`if(ok == true)`
Gefahr		`if(x = 4)`	`if(ok = true)`
Nix verstanden?			`if(true == ok)`
Tippfehler?			`if(true = ok)`
Geek-Style		`if(4 == x)`	`if(ok)`
Tippfehler?		`if(4 = x)`	

Beachten Sie, dass `4 == x` natürlich mit `x == 4` mathematisch identisch ist. Die zweite Variante läuft jedoch in einzelnen Sprachen (C, ...) Gefahr, sich bei Tippfehlern fatal zu verhalten. Die Variante `4 == x` ist hier etwas gewöhnungsbedürftig.

Generell: Wer die Konstanten links lässt, schreibt robusteren Code:

```
String s = ...;
if("abc".equals(s))
{
  ...
}
```

Umgekehrt (`s.equals("abc")`) kann es hier sein, dass die Variable `s` noch nicht definiert ist und eine sog. `NullPointerException` würde das Programm unsanft beenden.

Bemerkung: Je nach Programmiersprache ist eine andere Art die sinnvollste. Wichtig ist, dass Sie die obige Tabelle verstehen und in Ihrer Programmiersprache versuchen, die Vor- und Nachteile der einzelnen Bedingungen (Boole'sche Ausdrücke) kritisch zu hinterfragen.

A.18 Spezialitäten von Arrays

Hier folgen noch einige Eigenheiten von JAVA-Arrays.

A.18.1 Anonyme Arrays

Arrays können seit der JAVA 1.1 Version auch anonym definiert werden. Falls z. B. eine Methode `public int sum(int a[])` besteht, die alle Werte eines `int`-Arrays zusammenzählt, kann diese Methode mit der Syntax für anonyme Arrays ...

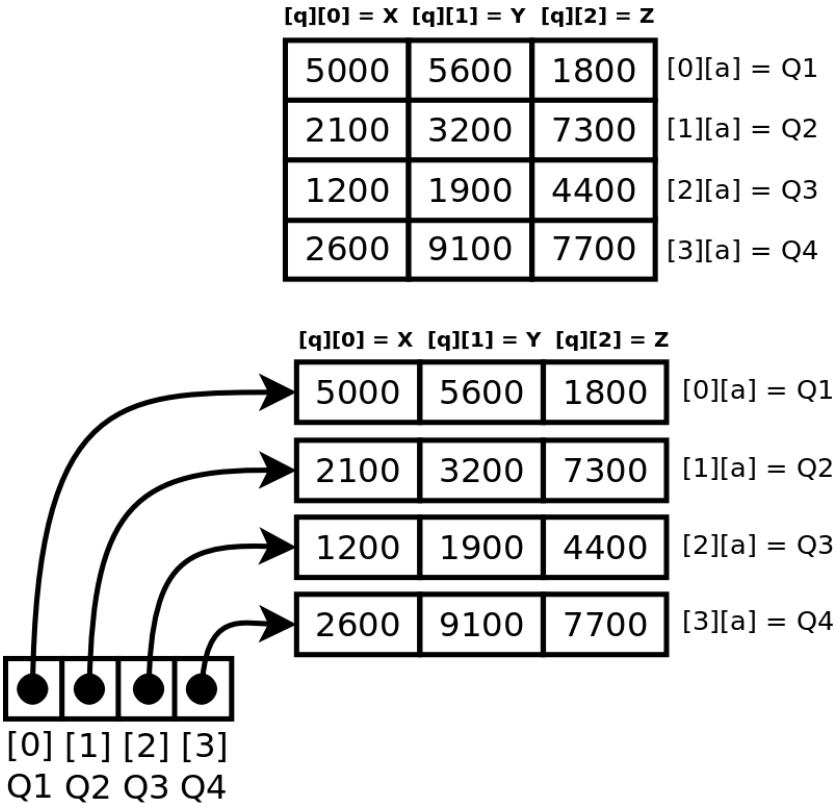
```
int summe = sum(new int[]{1, 5, 3, 2, 4, 4});
```

... aufgerufen werden.

A.18.2 Implementierung von Tabellen

Tabellen oder Matrizen kennt JAVA von sich aus nicht. Da aber ein Array auch aus Arrays bestehen kann, sind Tabellen einfach zu verwirklichen.

Hier nochmals das Beispiel aus dem Theorieteil (s. Kap. 6.2 auf Seite 87). Intern sieht die Umsetzung technisch wie folgt aus:



A.18.3 Arrays und Maps

Handelt es sich bei den Indizes nicht um Zahlen, so können (selbst für mehrdimensionale Tabellen) in JAVA `Map`s verwendet werden. Hier eine Umsatztabelle nach Quartalen (q1 - q4):

	q1	q2	q3	q4
Meier	48 000	36 000	52 000	68 000
Freimann	12 000	11 000	8 000	24 000
Huber	50 000	51 000	53 000	46 000
Müller	5 500	5 500	3 800	2 200
Guggisberg	38 000	36 000	48 000	54 000

Dabei wird eine Zeile wie folgt zugewiesen:

```
HashMap<String, Integer> meier;
meier = new HashMap<String, Integer>();
meier.put("q1", 48_000);
meier.put("q2", 36_000);
meier.put("q3", 52_000);
meier.put("q4", 68_000);
```

Um die Zeilen nun eine «Tabelle» zu füllen, kann wieder eine `Map` verwendet werden und wir erhalten mit diesem «Trick» die gewünschte Tabelle:

```
HashMap<String, HashMap<String, Integer>> tabelle;
tabelle = new HashMap<String, HashMap<String, Integer>>();

tabelle.put("Meier", meier);
tabelle.put("Huber", huber);
...
```

A.18.4 Tabellen mit Zeilen unterschiedlicher Länge

In folgendem Beispiel besteht die Tabelle aus drei unterschiedlich langen Zeilen:

```
int[][] matrix =
 {
   {37, 42},
   {12, 25, 18},
   {3, 9}
};

System.out.println(Arrays.toString(matrix[1]));
```

A.19 Besonderheiten von Strings

A.19.1 Stringlänge

JAVA verwendet für Strings Arrays (Felder) von Unicode Zeichen (`char`). Ein Unicode-Zeichen benötigt 2 Byte und Arrays werden mit `int` angesprochen. Somit könnte ein JAVA-String theoretisch 4GiB an Speicher einnehmen (2 Milliarden Zeichen!). Leider konnte ich dies nicht testen, da solche Tests in der Regel zunächst den Heap komplett auffressen. Stringlängen von über 600 000 000 Zeichen sind jedoch auf heutigen PCs kein Problem. Mit anderen Worten: Sie können in einer einzigen String-Variable den kompletten Text von über 500 Büchern speichern!

Die Länge eines Strings wird in JAVA mit der `length()` -Methode ermittelt:

```
meinString.length();
```

A.19.2 Teilstring

Teilstrings werden in JAVA wie folgt ermittelt: `str.substring(start, end)` . Zu bemerken ist, dass in einigen Programmiersprachen die Länge mitgegeben wird, in JAVA hingegen die gewünschte End-Position. Wollen Sie z. B. alle Zeichen ab Position 5 bis zur Position 10 aus einem String `str` extrahieren, so schreiben Sie in JAVA:

```
str.substring(5, 11)
```

Das Ende (`end`) wird in JAVA bei der String-Extraktion **nicht** mit eingeschlossen. `end` zeigt auf das erste Zeichen im String, das nicht mehr zum Teilstring gehören soll. Dieser offensichtliche Fehler ist ein Relikt aus der Programmiersprache C. Auch wenn es tatsächlich Algorithmen gibt, die damit effizient arbeiten können, so ist der gewählte Variablenname (`end`) mit Sicherheit falsch.

Aufgabe A.1 «Teilstring» Schreiben Sie ein JAVA-Programm, das vom Benutzer einen Text abverlangt, schneiden Sie den ersten und den letzten Buchstaben ab und geben Sie den Teilstring aus.

A.19.3 Unveränderbarkeit

Strings sind nicht veränderbar. Mit diesem Wissen erstaunt der *Output* des folgenden Codes nicht:

```
String a = "Hallo␣␣";
String b = "␣␣␣Welt";
a.trim(); b.trim();
System.out.println(a+b); // liefert "Hallo     Welt"
```

Die Idee war wohl aber eher die folgende:

```
String a  = "Hallo␣␣";
String b  = "␣␣␣Welt";
String a2 = a.trim();
String b2 = b.trim();
System.out.println(a2+b2); // liefert "HalloWelt"
```

Was ist geschehen? Die `trim()`-Funktion ändert nicht den gegebenen String, sondern es wird ein neuer String erzeugt, welcher *getrimmt*, also frei von führenden und nachfolgenden Leerzeichen, ist.

A.19.4 `StringBuilder`

Aus oben genannten Gefahren und weil zudem das Zusammenfügen von `String`s in JAVA eine *teure* Angelegenheit ist, verwendet man häufig auch den `StringBuilder`. Dieser reserviert sich von vornherein 16 Zeichen (32 Byte) und muss erst Speicher neu allozieren, wenn diese 16 Zeichen aufgebraucht sind.

Beispiel:

```
StringBuilder sb = new StringBuilder("Hallo");
sb.append("␣");
sb.append("Welt");
System.out.println(sb);
```

A.19.5 `equals`

Wir haben im Kapitel über Zeichenketten bereits gesehen, dass Strings nicht mit dem «`==`»-Operator verglichen werden sollten, wenn wir uns dafür interessieren, ob zwei Strings die selbe Zeichenfolge beinhalten.

Hier aber noch eine Möglichkeit, bei der der «`==`»-Operator trotzdem funktioniert. Hat man es in einem Programm sehr häufig mit denselben Zeichenketten zu tun[59], dann kann die `intern`-Funktion sehr nützlich sein:

`a = "Hallo"; b = "Hallo";`	`a == b`
`a = new String(b);`	`a != b`
`a = a.intern();`	`a == b`

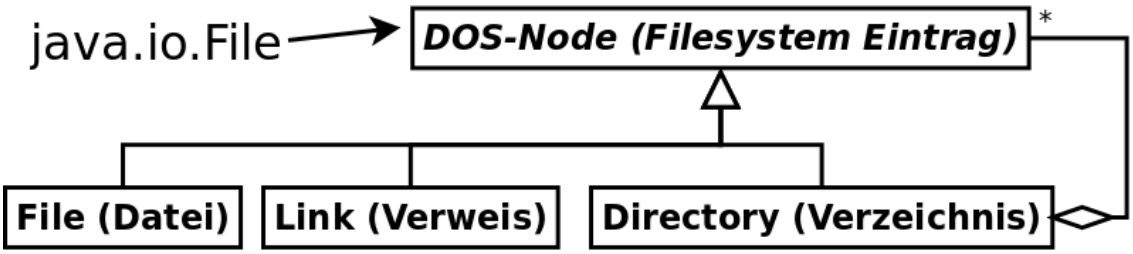
Im *internen* Konstanten String-Pool, kommt jeder Text genau einmal vor. Zum einen werden dort alle Literale aus dem Programmcode abgelegt. Mit der Funktion `intern()` kann ein Text ebenfalls dahin verschoben werden!

[59] Dies ist der Fall, wenn wenige Strings jeweils mehrere hundert Male im System auftauchen sollten.

A.20 Java-Dateien (file-handling)

Ist ein Programm einmal gestartet, so gibt es grundsätzlich zwei Arten von Daten. Die sog. persistenten Daten müssen auch zur Verfügung stehen, wenn das Programm einmal nicht mehr läuft. Dabei kann es sich um Kundendaten, Spielstände, Ausleihen in Bibliotheken etc. handeln. Andere sog. transiente (lat. kurzlebig) Daten sind jedoch nur gerade während der Programmausführung wichtig. Typische Vertreter sind Positionen von graphischen Komponenten (wie z. B. der Ort des OK-Buttons), der Ort des Cursors in Textverarbeitungen, berechnete statistische Werte, die jederzeit aus den persistenten Stammdaten wieder erzeugt werden könnten und und und (s. Kap. A.6.2 auf Seite 139).

Achtung - Eine Warnung vorab: Die Klasse `File` in JAVA bezeichnet leider keine Datei (engl. File) im landläufigen Sinne. Mit einem JAVA-`File()` wird ein Eintrag im File-System bezeichnet also ein Ordner(Directory), ein Link oder eben ein Dateiname (Filename); nicht aber der Inhalt einer Datei!



Die JAVA-File Klasse beinhaltet also lediglich einen Filenamen. Ob die Datei existiert und ob es sich dabei um ein Textfile, ein Bild oder gar um ein Verzeichnis handelt, interessiert JAVA vorerst nicht. Interessant sind daher vor allem folgende Methoden, wenn wir uns im Dateisystem bewegen:

```
exists()
isDirectory()
listFiles()
isFile()
canRead()
canWrite()
```

Aufgaben zu `File`

Aufgabe A.2 « `java.io.File` » Schreiben Sie ein Programm, das ein Verzeichnis einliest und alle darin enthaltenen Files ausgibt.

Zusatz 1: Geben Sie von jedem Inhalt jeweils an, ob es sich um ein Verzeichnis oder eine Datei handelt.

Zusatz 2: Tauchen Sie rekursiv in alle Unterverzeichnisse ab.

A.20.1 Reader und Writer

Eine typische Anwendung einer File-Verarbeitung ist das zeilenweise Einlesen, Verarbeiten und Ausgeben. Die folgende Demo-Anwendung liest eine Datei (Eingabe) ein, schreibt vor jede Zeile die Zeilennummer und schreibt dies in eine neue Datei (Ausgabe). Um das ganze etwas spannender zu machen, sollen bei Leerzeilen keine Zeilennummern ausgegeben werden, jedoch soll die Zeilennummer dennoch erhöht werden.

```
import java.io.*;

public class InOut {

  public static void main(String[] args) {
    new InOut().top();
  }

  final String  IN_FILE_NAME = "eingabe.txt";
  final String OUT_FILE_NAME = "ausgabe.txt";

  String line           ;
  int    lineNumber = 0;

  void top() {
    try(
      Reader           r   = new FileReader(IN_FILE_NAME );
      LineNumberReader in  = new LineNumberReader(r)      ;
      Writer           out = new FileWriter(OUT_FILE_NAME);
    )
    {
      line = in.readLine();
      while(null != line) {
        lineNumber = lineNumber + 1;
        if(line.trim().length() > 0) {
          out.write(lineNumber + ".␣");
        }
        out.write(line + "\n");
        line = in.readLine();
      }
    }
    catch(IOException iox) {
      System.out.println("Fehler␣mit␣Datei:␣" + iox);
    }
  } // end method top()

} // end of class InOut
```

Bemerkung: `null` wird von der Subroutine `readLine()` genau dann zurückgegeben, wenn keine Zeile mehr gelesen werden kann. Dies bezeichnet das Ende der Datei (engl. EOF = «End of File»).

A.20.2 Abkürzung im `while()`-Header

Eine spezielle Möglichkeit, das Vor- bzw. Nachlesen abgekürzt zu notieren, bietet JAVA im `while()`-Header.

Anstelle von

```
String line = in.readLine(); // Vor-Lesen
while(null != line) {
  behandle(line);
  line = in.readLine();      // Nach-Lesen
}
```

kann abkürzend folgendes stehen:

```
String line;
while(null != (line = in.readLine())) {
  behandle(line);
}
```

Hier wird zunächst ein `String` eingelesen und in die Variable `line` geschrieben. Da in JAVA eine Zuweisung gleichzeitig ein Ausdruck (Term) mit einem Wert ist, kann dieser Wert gleich im Vergleich eingesetzt werden. Sobald der Wert `null` von `readLine()` zurückgegeben wird, so bezeichnet dies das Ende der Datei.

Dies funktioniert, da die Zuweisungs-Anweisung in JAVA auch als Ausdruck verwendet werden kann.

A.20.3 Anfügen

Zeilen oder Zeichen können auch ans Ende einer bestehenden Datei angefügt werden. Dazu benötigt der `FileWriter` beim Erstellen einen zweiten Parameter (`true`).

Das folgende Beispiel zeigt, wie Textzeilen ans Ende der Datei `Ausgabe.txt` angefügt werden:

```
File out = new File("Ausgabe.txt");
try(FileWriter fw = new FileWriter(out, true)) {
  fw.append("Eine␣letzte␣Zeile\n");
} catch (IOException iox) {
  System.err.println("Fehler:␣" + iox);
}
```

Aufgaben zu `Reader` und `Writer`

Aufgabe A.3 «Top» Schreiben Sie ein Programm namens «Top», das von einer Text-Datei die ersten sieben Zeilen ausgibt.

Zusatz: Lesen Sie den Filenamen über die `main`-Parameter ein (s. Kap. A.23 auf Seite 185).

Aufgabe A.4 «Zähler» Schreiben Sie ein Programm, das von einer Text-Datei alle Zeilen zählt und diese Anzahl auf der Konsole ausgibt.

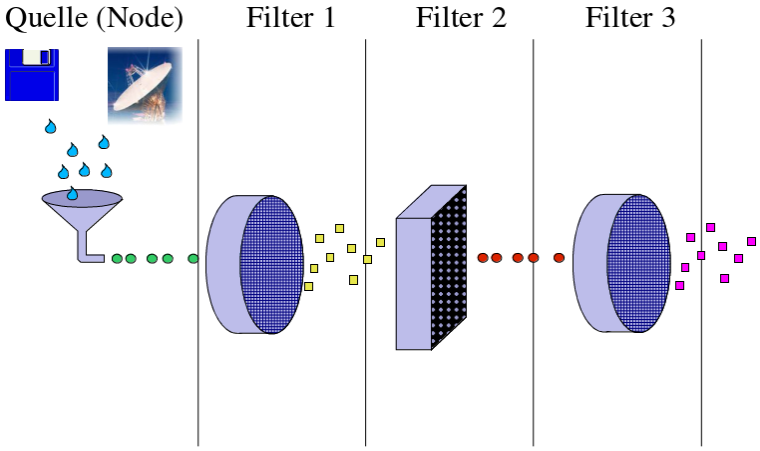
Zusatz: Zählen Sie alle Zeilen, Zeichen, alle Buchstaben und alle Wörter.

Zum Testen können Sie folgenden Text verwenden:

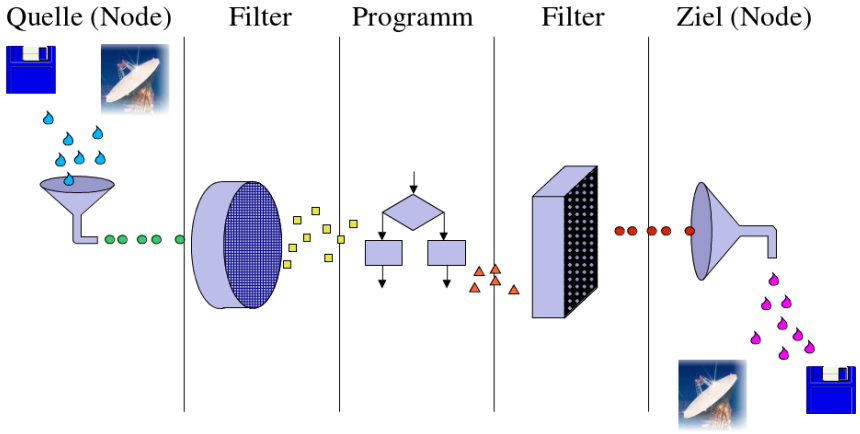
Dieser Text besteht
aus neun Zeilen und
zwanzig Wörtern,
davon eines ge-
trennt.
Total sind es
hundertneunzehn
Buchstaben mit
drei Satzzeichen.

A.20.4 Filterketten (Filter chaining)

JAVA verwendet für die Datenverarbeitung sog. **Filter**. Ein Filter kann man sich vorstellen wie eine Umwandlung von einem Datenstrom in einen anderen. So kann ein Filter zuständig sein, dass die Daten zunächst aus einer `*.zip`-Datei gelesen werden. Der zweite Filter verwandelt die Bytes anhand einer Kodierung in Unicode-Character (`char`). Ein dritter Filter ist zuständig, um die Datei zeilenweise einzulesen und so fort:



Im klassischen EVA-Prinzip[60] werden die Daten typischerweise von einem Datenstrom (Datei oder Netzwerk-Ressource) gelesen[61]. Nach dem Filtern in unser gewünschtes Format werden die Daten verarbeitet. Zum Schluss wird die Ausgabe aus unserem Programm wieder in ein neues gewünschtes Format *gefiltert* und so in den Ausgabedatenstrom gesendet:



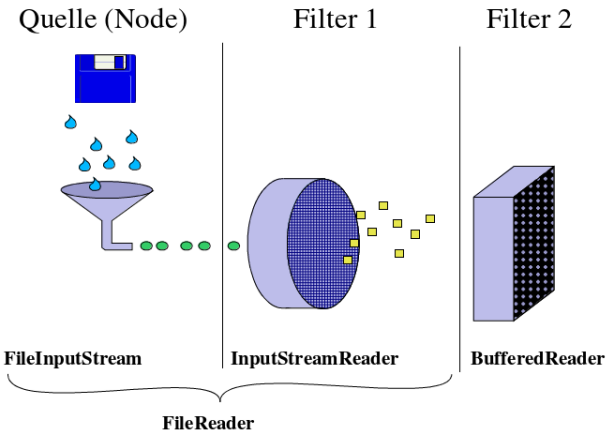
[60] EVA = Eingabe-Verarbeitung-Ausgabe

[61] Ein Datenstrom ist in JAVA eine Aneinanderreihung von `byte`. Beim Lesen aus einem Datenstrom, insbesondere bei Musik oder Videos von einer Netzwerk-Ressource, sprechen wir oft auch von **streamen**.

A.20.5 Klassisches Filter-Chaining

Das folgende Beispiel (nächste Seite) zeigt ein *Filter-Chaining*, um aus einer Datei einen Text zeilenweise auslesen zu können. Ein `FileReader` ist im Grunde nichts anderes als das *Chaining* eines `FileInputStream` mit einem `InputStreamReader`. Der `FileInputStream` liest byteweise von einer Datei (im Gegensatz z. B. von einem *Stream* einer `URLConnection`, welche Daten von einer Netzwerkressource liest). Der `InputStreamReader` ist zuständig, die Text-Kodierung (utf-8, Latin, EBCDIC, ...) aufzulösen und um aus den `byte` `char` herzustellen. Zu guter Letzt ist ein `BufferedReader` eingesetzt, der die Zeilenenden erkennt und im Wesentlichen eine Funktion `readLine` zur Verfügung stellt:

Klassisches Filter-Chaining




```
import java.io.*;

public class FilterChaining {

  public static void main(String[] args) {
    new FilterChaining().top();
  }


  void top()
  {
    String inFileName = "test.txt";
    String encoding   = "utf-8"   ;
    // Verwenden Sie zum Test die folgenden Kodierungen:
    // "utf-8", "iso-8859-1", "Cp437", "Cp850", "Cp500"
    try(
      FileInputStream   fis = new FileInputStream  (inFileName)   ;
      InputStreamReader isr = new InputStreamReader(fis, encoding);
      BufferedReader    br  = new BufferedReader   (isr)          ;
    )
    {
      String line;
      while (null != (line = br.readLine()))
      {
        System.out.println("Gelesen: " + line);
      }
    } catch (IOException iox) {
      System.out.println("Fehler in der Fileverarbeitung: " + iox);
    } // end of try-catch
  } // end of top()

} // end of class FilterChaining
```

Bemerkung A.2. Für die Bedingung in der `while()`-Schleife siehe Seite 170.

A.20.6 Datenströme (Streams)

Den *Readern* und *Writern* aus dem vorangehenden Kapitel liegen sog. Datenströme zugrunde. Ein Datenstrom ist eine Reihe von Bytes, wohingegen die Reader und Writer Zeichen (char) liefern und somit zur Textverarbeitung geeignet sind.

Viele Anwendungen gehen aber tiefer als die reine Behandlung von Texten und Verarbeiten die Daten byteweise. Dabei kann es sich um Files, aber auch um Datenströme von einem Netzwerk handeln.

Die folgende Tabelle hilft zu entscheiden, welche der JAVA-Objekte im Zusammenhang mit `Readers`, `Writers` und `Streams` verwendet werden sollte.

	byte (8 bit)	char (16 bit) / String
Lesen	`InputStream`	`Reader`
Schreiben	`OutputStream`	`Writer`

Das folgende Hauptprogramm gibt eine Datei Byteweise auf der Konsole aus:

```
void top() {
  try(
    FileInputStream fis = new FileInputStream("inFile.bin");
  ) {
    int inByte;
    while(-1 != (inByte = fis.read()))
    {
      System.out.println(inByte + "␣");
    }
  } catch(IOException iox) {
    System.out.println("Fehler␣in␣der␣Dateibehandlung:␣" + iox);
  }
}
```

Beachten Sie, dass der Befehl `read()` die Zahl -1 zurückgibt, wenn das Ende der Datei erreicht ist. Die Bytes werden in Werten von 0 bis 255 wiedergegeben und nicht wie sonst in JAVA üblich von -128 bis +127.

Bemerkung A.3. Die Bedingung in der `while()` -Schleife ist hier analog zur `readLine()` -Funktion für Zeichenketten (s. Kap. A.20.2 auf Seite 170).

Encoding Zeichen werden durch Bytes (in JAVA 8-bit / ASCII 7-Bit) bzw. durch Characters (in JAVA 16 Bit) repräsentiert. Bytes werden vorwiegend in Datenströmen (Files bzw. Netzwerkverbindungen TCP/IP) verwendet.

Nun können wir uns die Frage stellen, warum es dann die Unterscheidung in `byte` und `char` überhaupt braucht. JAVA-`byte` sind einfach eine Aneinanderreihung von je 8 Bit. Welche Buchstaben sich dahinter verbergen ist aber von System zu System verschieden. Die Zeichen (`char`) werden durch sogenannte Kodierungen (encodings) festgelegt. Ein `FileReader`, wie wir ihn oben verwendet hatten, verwandelt eine Datei (`byte`-Strom) in einen Zeichenstrom, indem die Kodierung des aktuellen Betriebssystems verwendet wird. Auf modernen Betriebssystemen ist das meist die UTF-8 Kodierung.

Unicode 16-Bit Zeichen werden in JAVA im RAM verwendet. Je nach Kodierung liegt einem Zeichen ein anderes Bitmuster (oben als Hex-Wert angegeben) zugrunde. Im RAM der JAVA virtuellen Maschine (JVM) wird jedoch stets Unicode verwendet. So brauchen wir uns lediglich beim Lesen und Schreiben außerhalb des Programmes (Files, Network I/O, ...) um die Kodierung zu kümmern.

Die folgende Tabelle zeigt den Sachverhalt anhand der Zeichen «K» und «ä»:

	Files / Networking (`Streams`)	**Java (char / Strings) (`Readers` / `Writers`)**
	Bytes (`byte`)	Characters (`char`)
	8 Bit (= 1 Oktett)	16 Bit (= 1 char)
	256 mögliche Zeichen	65'536 mögliche Zeichen
Zeichen	Kodierungen:	Kodierungen in JAVA: Immer Unicode (`www.unicode.org`)
K	#4B = ASCII 'K'	0x004B = 'K' (= «00» + ASCII)
	#D2 = EBCDIC 'K'	
ä	#E4 = ANSI 'ä'	0x00E4 = 'ä' (= «00» + ANSI)
	#8A = Mac 'ä'	
	#84 = Cp-437 'ä'	
	#C3A4 = UTF-8 'ä'	

JAVA verwendet intern für `char` (**und somit auch für Strings**) die Unicode-Kodierung. Siehe `http://www.unicode.org`.

Aufgabe A.5 «Kodierungen» Lesen Sie eine Datei mit dem folgenden Programm ein (dabei wird das File `test.txt` je nach Kodierung anders auf der Konsole wieder ausgegeben).

```
void top() {
  String inFileName = "test.txt";
  String encoding   = "utf-8"   ;
  // Verwenden Sie zum Test die folgenden Kodierungen:
  // "utf-8", "iso-8859-1", "Cp437", "Cp850", "Cp500"
  try(
    FileInputStream   fis = new FileInputStream  (inFileName)   ;
    InputStreamReader isr = new InputStreamReader(fis, encoding);
  ) {
    int nextChar;
    while (-1 != (nextChar = isr.read()))
    {
      char ch = (char) nextChar;
      System.out.print(ch);
    }
  } catch (IOException iox) {
    System.out.println("Fehler␣in␣der␣Fileverarbeitung:␣" + iox);
  }
}
```

Aufgabe A.6 «Filevergleich» Zwei Dateien sollen geöffnet werden und der Inhalt byteweise verglichen werden. Wenn die Dateien identisch sind, dann und nur dann soll `true` zurückgegeben werden - ansonsten `false`.

Aufgabe A.7 «Lesbar» Eine Datei soll zum Lesen geöffnet werden. Daraus soll eine neue Datei geschrieben werden, die nur die Buchstaben und die Leerschläge und die Satzzeichen Punkt und Komma enthält, welche in der Originaldatei vorhanden waren. Alle anderen Zeichen sollen ignoriert werden.

A.20.7 Randomaccess-Files

Manchmal wollen wir eine Datei aber nicht nur sequentiell lesen sondern innerhalb der Datei Ausschnitte lesen bzw. schreiben.

Gleich ein Beispiel:

```
File             dataFile = new File("dataFile.raw");
RandomAccessFile f        = new RandomAccessFile(dataFile, "rw");

f.seek(64);                    // Auf Position 64 springen
byte daten[] = new byte[32]; // 32 Byte Platz bereitstellen
f.read(daten);                 // 32 Byte aus dem File lesen
f.writeInt(1234);              // Die Zahl 1234 (als 4 Byte Int) an
                               // Stelle 96 Schreiben (Die Position
                               // ist ja 64+32)
f.close();                     // File wieder schliessen
```

Aufgabe A.8 «PNG» Schreiben Sie ein Programm, das von einer PNG (Portable Network Graphics) Datei die Header ausgibt.

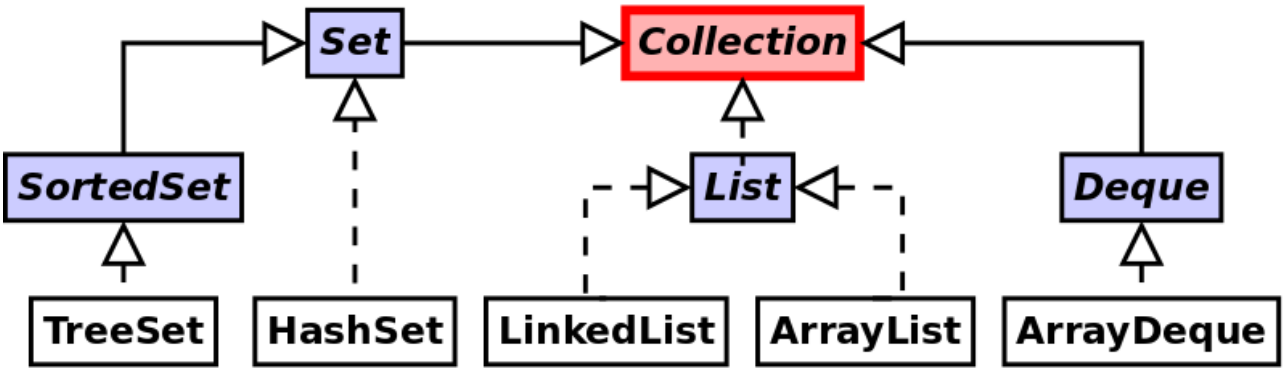
Das PNG-Fileformat ist wie folgt aufgebaut:

- Erste 8 Byte = Magic number = Byte values 137, 80, 78 71 13 10 26 10
- Danach folgen beliebig viele *Chunks*:
  - Ein Chunk beginnt mit der Chunk Größe (4 Byte), danach kommt der Chunk Name (4 Byte), danach kommen die Chunk Daten (die Länge wurde in der Größe angegeben). und zuletzt folgen nochmals 4 Byte Prüfsumme (CRC).
  - Der Letzte Header hat Länge 0 und heißt IEND. Sobald dieser Header erreicht ist, folgen keine PNG relevanten Daten mehr.

Verwenden Sie die Methoden
`read()`, `readInt()`, `skipBytes()` und `close()`.

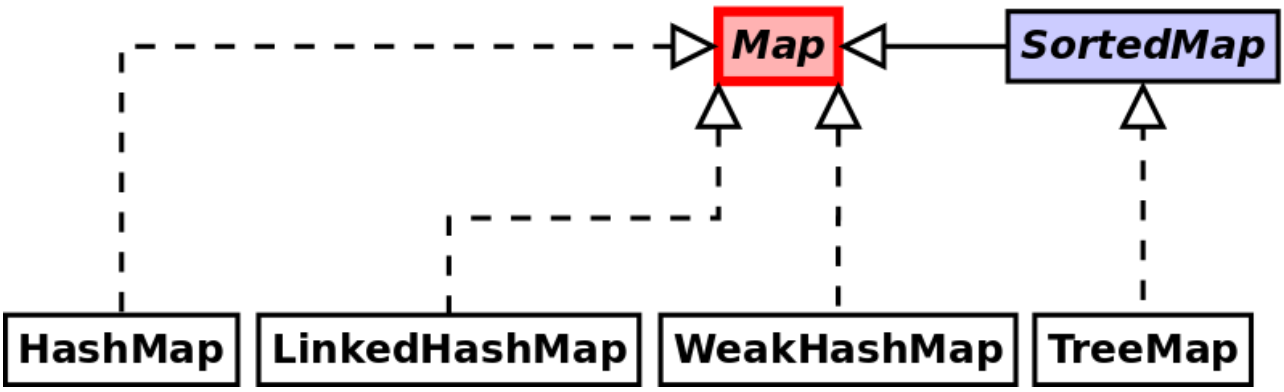
A.21 Java Collection Framework

A.21.1 Collection



- **Collection**: Interface, das alle Collections implementieren müssen.
- **List**: Die Elemente sind geordnet, d. h. in derselben Reihenfolge, wie sie eingefügt werden.
- **ArrayList**: Auf die Elemente kann via Index zugegriffen werden. Das Einfügen am Anfang oder in der Mitte ist zeitaufwändig.
- **LinkedList**: Geordnete Sequenz, bei der am Anfang und am Ende hinzugefügt, abgeholt und gelöscht werden kann. Somit kann eine `LinkedList` auch als `Queue`, `Deque` oder `Stack` verwendet werden. Das Einfügen in der Mitte geht sehr rasch.
- **Set**: Set = Menge: Elemente können nur einmal angefügt werden.
- **HashSet**: Set, der mit einer Hash-Tabelle Elemente einfügt und wieder finden kann. Es entspricht etwa einer `HashMap`, bei der aber nur Keys und keine Values eingefügt werden.
- **SortedSet**, **TreeSet**: Set, der die Elemente sortiert. Bei der Implementation `TreeSet` wird ein Baum als abstrakter Datentyp eingesetzt. Die `compareTo()` Funktion wird verwendet, um Elemente an die richtige Position einzufügen.

A.21.2 Assoziative Arrays: Java-`Map`



Oftmals reichen ganze Zahlen nicht als vernünftige Indizes aus. Hier kommen Maps zum Zuge.

- **Map**: Interface mit den wichtigen Methoden `put(key, value)`, `containsKey(key)`, `containsValue(value)`, `get(key) : value`, `remove(key)` und `size()`. Zum Iterieren werden die beiden Methoden `keySet()` und `values()` verwendet, welche die Schlüssel bzw. die Datenobjekte zurückgeben.
- **HashMap**: Klassische assoziative Arrays. Bei assoziativen Arrays sind die Indizes nicht Zahlen, sondern beliebige Schlüsselobjekte.
- **LinkedHashMap**: Wie die `HashMap`, jedoch wird zusätzlich eine Liste geführt, in welcher Reihenfolge die Objekte hinzugefügt wurden.
- **WeakHashMap**: Objekte dürfen bei Nichtgebrauch vom Garbage Collector vernichtet werden.
- **SortedMap**, **TreeMap**: Wie `SortedSet` bzw. `TreeSet`. Im Gegensatz zu diesen oben genannten Objekten arbeiten die Maps auf den Schlüsseln und nicht auf den Datenobjekten.

A.21.3 Auffinden der richtigen Sammlung

Oft ist es nicht einfach zu sehen, welche Collection bzw. Map dann für ein Problem die geeignetste Form ist. Hier eine Hilfe in Zweifelsfällen.

Ist ein schneller Zugriff über Schlüsselobjekte nötig?

- `Map`

Dürfen länger nicht benutzte Objekte vom Garbage Collector abgeräumt werden?

- `WeakHashMap`

Ist die Reihenfolge der Objekte relevant?

- `List`
- `LinkedHashMap`

Werden nur ab und zu, bzw. beim Initialisieren Daten hinzugefügt bzw. entfernt?

- `ArrayList`

Gibt es eine natürliche Ordnung (z. B. alphabetisch) auf den Daten bzw. auf den Schlüsseln?

- `TreeSet` bzw. `TreeMap`

Kann jedes Objekt nur einmal gespeichert werden? (Duplikate sind nicht gestattet.)

- `Set`

A.21.4 Eigene Sammelobjekte

Natürlich ist es auch möglich, eigene generische Datentypen herzustellen. Ein Beispiel findet sich im Anhang: (s. Kap. A.22 auf Seite 182)

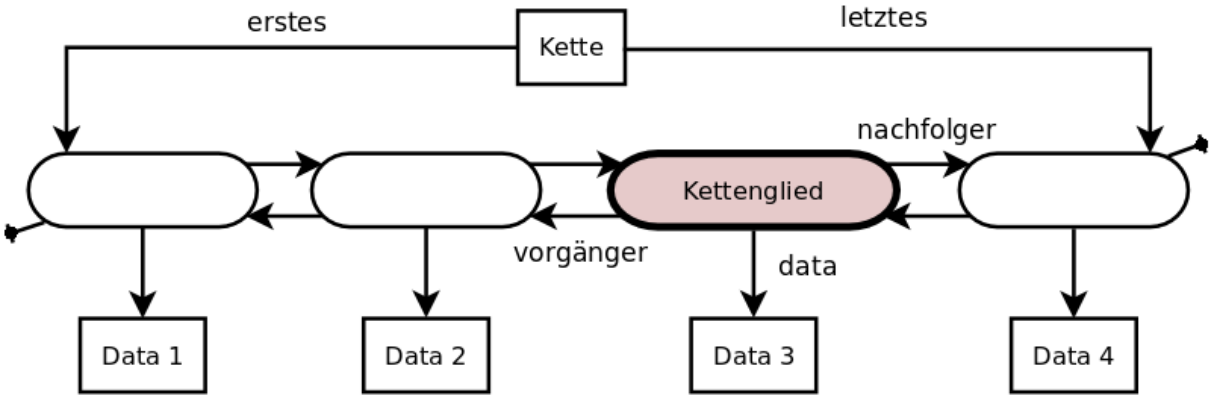
A.22 Eigene Sammelobjekte: Beispiel Kette (LinkedList)

Natürlich können wir für eine effiziente Softwareentwicklung Sammelobjekte (Container) wie `ArrayList`, `Set`, `Map`, ... einfach verwenden. Für Interessierte ist es jedoch genauso spannend, ein solche Sammlungen einmal selbst zu schreiben oder zumindest ein bestehendes Sammelobjekt zu analysieren. Ein solches Verständnis hat zudem den Vorteil, dass bei mehreren Möglichkeiten der beste Sammelcontainer ausgewählt wird — oder gar selbst effizienter implementiert oder gar neu geschrieben werden kann.

Zunächst müssen wir uns im Klaren sein, welche Eigenschaften **fast alle** Sammelcontainer aufweisen. Dies sind: Hinzufügen, Entfernen, Zählen, Suchen, ... Wichtig ist aber auch, dass man einem Container jedes beliebige Objekt hinzufügen kann. Der Container kümmert sich überhaupt nicht um die Inhalte[62].

Somit verwenden die Sammelcontainer ihrerseits Referenzvariable auf die Nutzobjekte. Diese Referenzen können in Arrays, aber auch in eigenen *Verpackungs-Objekten* geführt werden.

Wir wollen als Beispiel eine verkettete Liste implementieren (`Kette`, «linked List»). Sie verwendet ein `Kettenglied`, um die Referenzen auf die Nutzdaten (`data`) zu speichern. Ebenso muss natürlich ein Zeiger auf das vorangehende `vorgaenger`, wie einer auf das nachfolgende Kettenglied (`nachfolger`) gespeichert werden:



Das erste Kettenglied hat keinen `vorgaenger` (wie auch das letzte keinen `nachfolger` hat). Dies ist in der Grafik durch einen NIL-Pointer[63] dargestellt.

[62] Eine Ausnahme bilden die Sammelcontainer, welche die enthaltenen Objekte sortieren können. Hier wird meist noch eine Funktion zum Vergleich der Objekte gefordert (z. B. `gleich()`, `kleinerAls()`, `vergleicheMit()`).

[63] NIL = «not in list» wird in JAVA mit dem `null`-Pointer implementiert.

Die Implementation der Kette kann dann wie folgt aussehen:

(Ich zeige hier lediglich exemplarisch die Methode `existiert()`. Die Klasse mit allen Sortier-, Hinzufüge-, Auffind-, Entferne- etc. -Funktionen ist natürlich ziemlich lange.)

```
public class Kette<T> implements Iterable<T>{
  Kettenglied<T> erstes ;
  Kettenglied<T> letztes;
  int anzahlGlieder;

  public Kette() {
    listeLeeren();
  }

  public void listeLeeren() {
    erstes        = null;
    letztes       = null;
    anzahlGlieder = 0   ;
  }

  public boolean existiert(T objekt) {
    Kettenglied<T> akt = erstes;
    while(null != akt)
    {
      if(akt.getData() == objekt)
      {
          return true;
      }
      akt = akt.nachfolger;
    }
    return false;
  }

  ... //weitere Funktionen
```

Ein einzelnes Kettenglied könnte nun wie folgt aussehen:

```
public class Kettenglied<T> {
  Kettenglied<T> vorgaenger; // prev
  Kettenglied<T> nachfolger; // next
  private T data;

  public Kettenglied(T data) {
   this.data = data;
  }

  public T getData() {
    return data;
  }
} // end of class Kettenglied
```

Der generische Datentyp `<T>` sagt JAVA, dass hier ein beliebiges Objekt verwendet werden darf. Es muss sich jedoch bei allen Objekten um Objekte derselben Klasse handeln.

A.23 Eingabe via Kommandozeilen-Argument

Die meisten Programmiersprachen können ihre Programme ab einer Konsole (shell, cmd) starten. Die meisten davon wiederum kennen sogenannte Kommandozeilen-Argumente.

Die JAVA-`main()`-Methode muss eine Parameterliste vom Datentyp `String[]` aufweisen. Diese Liste wird in den meisten Übungsaufgaben und Lehrbüchern mit `args` bezeichnet. Dieser Array besteht aus den Strings, die beim Aufruf der Klasse der virtuellen Maschine mitgegeben wurden.

Starten wir z. B. ein JAVA-Programm Namens `Hitch.class` aus der Konsole mit dem folgenden Befehl:

```
>java Hitch "Hallo␣Welt" 42 "marvin␣is␣cool"
```

So erhält die Variable `String[] args` in der JAVA-spancodemain-Methode die drei Argumente

- `args[0]` hat den Wert "Hallo Welt"
- `args[1]` hat den Wert "42" und
- `args[2]` hat den Wert "marvin is cool".

Ach ja, hier noch ein möglicher Quelltext des Programmes "Hitch":

```
package buchBeispiele.anhang;

public class Hitch {

  public static void main(String[] args)
  {
    new Hitch().top(args);
  }


  void top(String[] parameter)
  {
    int i = 0;
    for(String s: parameter)
    {
      System.out.println("args[" + (i++) + "]␣hat␣Wert␣" + s);
    }
  }
} // end of class Hitch
```

A.23.1 Typische Anwendung

Oft werden die `String[] args` verwendet um Filenamen (z. B. `inFileName` und `outFileName`) anzugeben. Wenn jemand einen der beiden Namen vergisst, so soll eine Angabe zur typischen Verwendung (usage) ausgegeben werden:

```
public class MainUsage {

  public static void main(String[] args)
  {
    if(2 != args.length)
    {
      printUsage()  ;
      System.exit(1); // Fehlercode > 0
    }
    new MainUsage().top(args);
  }

  void top(String[] args)
  {
    String  inFileName = args[0];
    String outFileName = args[1];
    ... // TODO: Hauptprogramm hier.
  }

  static void printUsage()
  {
    System.out.println("Verwende␣dieses␣Programm␣so:");
    System.out.println("java␣MainUsage␣<inName>␣<outName>");
  }

} // end of class MainUsage
```

Mittlerweile gibt es auch einige Frameworks, die einem das Parsen von umfassenden Kommando-Argumenten erleichtern[64].

[64] Siehe z. B. Usage Scenarios bei `http://commons.apache.org/proper/commons-cli/`

B | Aufgabenverzeichnis / Lösungen

Neben den offiziellen Aufgaben aus dem Buch [GFG11] und der zugehörigen Webseite hat es im vorliegenden Skript weitere Aufgaben, welche hier zusammengefasst sind:

- Aufgabe Teilausdrücke (s. Kap. 1.1.1 auf Seite 18)

  Lösung : `w`, `3.2`, `x`, `2`, `y`, `sin(y)`, `2 * sin(y)`, `x + 2 * sin(y)`, `3.2 * (x + 2 * sin(y))` und natürlich der Gesamtausdruck `w + 3.2 * (x + 2 * sin(y))`.

- `int` (s. Kap. 2.1 auf Seite 34) Hier wird ein Überlauf (Overflow) generiert.

- Die richtige Wahl (s. Kap. 3.10 auf Seite 51)

  Kompiliert werden alle, außer D. Korrekt sind alle außer C und D. Die beste Lösung ist G.

- Name und Alter ausgeben (s. Kap. 5.1 auf Seite 68)

- Leseaufgabe zu den Parametern (s. Kap. 5.7 auf Seite 79)
  c = 17
  r = 46
  x=33, y=42, z=33

- Person Erfassen (s. Kap. 8.1 auf Seite 100)

- Leseaufgaben zu Algorithmen (s. Kap. 9.1 auf Seite 110)

  - «Selektion» Finde das Minimum ((4, 6, 3) liefert 3)
  - «Iteration» Quadratzahl (4 liefert 16, 11 liefert 121)
  - «Selektion und Iteration» Umdrehen der Ziffernfolge (236 wird zu 632)
  - «Zahlenspielerei» Verdoppeln der Ziffernfolgen (236 wird zu 223366)
  - «Fuß gesteuerte Schleife» Finde den kleinsten Teiler (63 liefert 3, 17 liefert 17)
  - «Nur für Spieler» Finde heraus, ob der Array ein sog. *Full-House* darstellt: Ein Paar + ein Trippel.
  - «Ein sehr alter Algorithmus» Wurzelziehen nach Heron: Berechnet die Quadratwurzel

- Teilstring (s. Kap. A.1 auf Seite 164)

- Verzeichnis ausgeben (s. Kap. A.2 auf Seite 167)

- Top (s. Kap. A.3 auf Seite 171)

- Zähler (s. Kap. A.4 auf Seite 171)

- Kodierungen (s. Kap. A.5 auf Seite 177)

- File Vergleich (s. Kap. A.6 auf Seite 177)

- File lesbar (s. Kap. A.7 auf Seite 177)
- Dateistruktur PNG (s. Kap. A.8 auf Seite 178)

C | Link-Verzeichnis

- `http://commons.apache.org/proper/commons-cli/` : Umgang mit Kommandozeilenargumenten
- `https://github.com/pheek/javaInput/blob/master/Input.java` : Vereinfachtes Einlesen von Zahlen und Text über die Konsole
- `http://www.gress.ly/javastarter` : Erstes JAVA-Programm als Download. Doch ich rate allen, das Programm abzutippen und so eine erste Übung zu erhalten.
- `http://www.oracle.com/technetwork/java/javase/downloads/index.html` : Download JAVA
- `http://www.programmieraufgaben.ch` : Aufgabenbuch zu diesem Skript
- `http://smoch.santis-basis.ch/index.php/halbaddierer` : Funktionsweise und Nachbau eines Halbaddierers
- `http://www.santis-training.ch/java/javasyntax/keywords` : Liste der JAVA Schlüsselwörter
- `http://www.santis-training.ch/java/javasyntax/operatoren.php` : Liste der JAVA Operatoren
- `http://www.santis-training.ch/java/pitfalls.php#c` : Fehler und Fallen in Programmiersprachen, vorwiegend in JAVA
- `http://www.unicode.org` : Vollständige Unicode Zeichentabelle
- `http://www.wolframalpha.com` : Lösen mathematisch formulierter Probleme online (z. B. Gleichungssysteme)

D | Literaturverzeichnis

Literatur

[GF14] Gressly-Freimann, Philipp: *Objekte und Klassen.* Philipp Gressly Freimann, 2014 `http://www.programmieraufgaben.ch`

[GFG11] Gressly-Freimann, Philipp ; Guggisberg, Martin: *Programmieren lernen.* Orell Füssli, 2011 `http://www.programmieraufgaben.ch`. – ISBN 978–3–280–04066–9

[Him01] Himanen, Pekka: *Die Hacker-Ethik und der Geist des Informations-Zeitalters.* Riemann Verlag, 2001. – ISBN 978–3570500200

[Knu97] Knuth, Donald E.: *The Art of Computer Programming.* Addison Wesley, 1997 `http://www-cs-faculty.stanford.edu/ uno/`. – ISBN 0–201–89683–4

[Tzu00] Tzu, Sun: *Die Kunst Des Krieges.* -500

E | Stichwortverzeichnis