Programmieren I

anhand von JAVA

Version 0.2.0
für den Unterricht an der tsbe

Kurt Järmann

Burgergasse 36a

3400 Burgdorf

kurt.jaermann@gibb.ch

16. August 2017

Geschlechtsneutrale Formulierung

Aus Gründen der einfacheren Lesbarkeit wird in den vorliegenden Unterlagen auf die geschlechtsneutrale Differenzierung, z. B. Benutzer/innen, verzichtet. Entsprechende Begriffe gelten im Sinne der Gleichbehandlung für beide Geschlechter.

Danksagung

Susanne Spahr, für Rechtschreibung und Satzstellung

Sebastian Kühn, für Rechtschreibung und Satzstellung

Inhaltsverzeichnis

[1] Der grösste Teil wurde aus der Wikipedia übernommen.
[2] Die bitweisen Operatoren werden in Java nur selten eingesetzt.

[3]Ein Grossteil wurde von ZUM kopiert

1. Einleitung

1.1. Lernziele

Ziel des vorliegenden Skriptes ist eine möglichst breite Einführung in die Computerprogrammierung. Um dieses Ziel zu erreichen wird versucht möglichst viel der Theorie in die Praxis umzusetzen oder die Theorie indirekt über eine praktische Übung zu erlernen.

1.2. Geschichtliches[4]

Die Urversion von Java – auch Oak (Object Application Kernel) genannt – wurde in einem Zeitraum von 18 Monaten vom Frühjahr 1991 bis Sommer 1992, unter dem Namen "The Green Project" von neun weiteren Entwicklern im Auftrag des US-amerikanischen Computerherstellers Sun Microsystems entwickelt. James Gosling war der Hauptentwickler. Der Name Oak hatte, nach Gerüchten, seinen Ursprung in einer Eiche (engl. oak), die vor dem Fenster von James Gosling stand. Der Name musste aufgrund rechtlicher Probleme (es gab bereits eine Software dieses Namens) jedoch verworfen werden. Man entschied sich für den Namen Java nach einer starken Kaffee-Sorte, die speziell für Espresso Verwendung findet (Java-Bohne) und die von den Entwicklern bevorzugt getrunken wurde.

Das Ziel war nicht nur die Entwicklung einer weiteren Programmiersprache, sondern einer vollständigen Betriebssystemumgebung, inklusive virtueller CPU, für unterschiedlichste Einsatzzwecke. Das System sollte – der Legende nach – beispielsweise eine Kaffeemaschine steuern können.

[4] Der grösste Teil wurde aus der Wikipedia übernommen.

Teil I.
Grundlagen

2. Kommentare

Als Kommentare bezeichnet man in Programmiersprachen besondere Code-Teile, die vom Compiler nicht in Maschinencode – in Java Bytecode – übersetzt werden, sondern nur den Lesern beziehungsweise den Programmierern dienen. Neben selbstsprechenden Namen für Variablen, Klassen und Methoden, sollte man Kommentare verwenden, um bestimmte Stücke des Quellcodes zu dokumentieren. Damit erleichtert man zum Einen anderen Entwicklern zu verstehen, was ein Codefragment macht und zum anderen ist man dadurch in der Lage auch ein Programm zu verstehen, das man zum letzten mal vor einigen Monaten angesehen hat. Auch benutzt man Kommentare, um Anmerkungen – wie TODOs – festzuhalten, die einem während dem Programmieren auffallen.

Java bietet uns drei Möglichkeiten von Kommentaren:

- Zeilenkommentar `// Kommentar`
- Blockkommentar `/* Kommentar */`
- JavaDoc-Kommentar `/** Kommentar */`

Ein Zeilenkommentar gilt nur für eine Zeile und wird mit einem `//` eingeleitet. Möchte man Kommentare über mehrere Zeilen hinweg setzten, so bietet sich der Blockkommentar da für an. Dieses wird mit einem `/*` eingeleitet und endet mit `*/`. Der JavaDoc-Kommentar ist ein besonderer Blockkommentar, der zum Beispiel Beschreibungen von Funktionen und / oder deren Parameter enthält. Beginnt man einen Kommentar mit `/**` und schliesst es mit `*/` ab, handelt es sich um diesen Typ von Kommentar. Mit dem im JDK mitgelieferten Tool javadoc ist es möglich diese Kommentare zu einer API-Dokumentation zu generieren.

Mit Kommentaren können wir Code auch zum Testen und zur Fehlersuche auskommentieren, um diese vor dem Compiler zu verstecken.

Folgendes Beispiel soll uns die Verwendung von Kommentaren in Java verdeutlichen:

```
1 /**
2  * Kommentar Demo
3  * @author ich
4  * @version 0.1
5  *
6 */
7 public class Kommentare {
8   /**
9    * Ich lerne programmieren!
10   * @param args Die Kommandozeilen Argumente als Array.
11   */
12  public static void main(String[] args) {
13    /*
14     * Mehrzeiliger
15     * Kommentar
16     */
17    System.out.println("ich kann rechnen "+(5+6)); // 8.tung ()
18    // achte auf die runden Klammern
19    // TODO betrachte diese Zeile in Eclipse!
20  }
21 }
```

Beim Kommentar von Zeile 1 bis 6 handelt es sich um einen JavaDoc-Kommentar, der besondere Schlüsselwörter wie @author enthält, die vom javadoc-Tool ausgewertet werden können.
Bei Zeile 13 bis 16 wird ein Blockkommentar gezeigt, der sich über mehrere Zeilen erstreckt.
Zu guter Letzt wird auf den Zeilen 18 und 19 der Zeilenkommentar gezeigt.

3. Elementare Datentypen und Variablen

Java nutzt, wie es für imperative Programmiersprachen typisch ist, Variablen zum Ablegen von Daten. Eine Variable ist ein reservierter Speicherbereich und belegt – abhängig vom Inhalt – eine feste Anzahl von Bytes. Alle Variablen (und auch Ausdrücke) haben einen Typ, der zur Übersetzungszeit bekannt ist. Der Typ wird auch Datentyp genannt, da eine Variable einen Datenwert, auch Datum genannt, enthält. Beispiele für einfache Datentypen sind: Ganzzahlen, Fliesskommazahlen, Wahrheitswerte und Zeichen. Der Typ bestimmt auch die zulässigen Operationen, denn Wahrheitswerte lassen sich nicht addieren, Ganzzahlen schon. Dagegen lassen sich Fliesskommazahlen addieren, aber nicht XOR-verknüpfen. Jede Variable hat einen vom Programmierer vorgegebenen festen Datentyp. Dieser ist zur Übersetzungszeit bekannt und lässt sich später nicht mehr ändern. Java achtet stark darauf, welche Operationen erlaubt sind. Auch von jedem Ausdruck ist spätestens zur Laufzeit der Typ bekannt, daher ist Java eine statisch und streng (stark) typisierte Programmiersprache. (Während in der Literatur bei den Begriffen statisch getypt und dynamisch getypt mehr oder weniger Einigkeit herrscht, haben verschiedene Autoren unterschiedliche Vorstellungen von den Begriffen streng (stark) typisiert und schwach typisiert.)

> Hinweis: In Java muss der Datentyp einer Variablen zur Übersetzungszeit bekannt sein. Das nennt sich statisch typisiert. Das Gegenteil ist eine dynamische Typisierung, wie sie etwa JavaScript verwendet. Hier kann sich der Typ einer Variablen zur Laufzeit ändern, je nachdem, was die Variable enthält.

Primitiv- oder Verweis-Typ

Die Datentypen in Java zerfallen in zwei Kategorien:

- Primitive Typen. Die primitiven (einfachen) Typen sind die eingebauten Datentypen für Zahlen, Unicode-Zeichen und Wahrheitswerte.
- Referenztypen. Mit diesem Datentyp lassen sich Objektverweise etwa auf Zeichenketten, Dateien oder Datenstrukturen verwalten.

Wir werden uns im Folgenden erst mit primitiven Datentypen beschäftigen. Referenzen werden nur dann eingesetzt, wenn Objekte ins Spiel kommen. Die nehmen wir uns im Kapitel »Klassen und Objekte«, vor.

3.1. Primitive Datentypen im Überblick

Java stellt dem Programmierer acht sogenannte elementare oder primitive Datentypen zur Verfügung. Sie bilden in diesem Sinne die kleinste Einheit, wenn es darum geht, eine Variable zu typisieren. Der Typ gibt dabei die Art der zu speichernden Daten an.

Alle elementaren Datentypen haben in Java eine feste Grösse und einen exakt vorgeschriebenen Wertebereich. Diese Angaben sind auf allen Systemen die gleichen, daher ergibt sich eine bessere Portabilität und Stabilität. Des Weiteren lassen sich die diversen Datentypen in Kategorien einteilen.

- Bool' sche Datentypen
- Integer (Ganze Zahlen)
- Fliesskommazahlen (Reelle Zahlen)
- (Schrift)Zeichen

3.1.1. Boolesche Datentypen

Es gibt einen booleschen Datentyp, welcher nur zwei Zustände kennt. Wahr oder falsch. Als feste Werte dienen dabei die Zustände True und False, welche als einzige mögliche Werte zuweisbar sind.

Typ	Byte	Wertebereich	Standartwert
boolean	1	true, false	false

Tabelle 1: Boolesche Datentypen

3.1.2. Integer Datentypen

Bei Integern handelt es sich um Zahlenwerte ohne jegliche Kommastellen. Java stellt Ihnen insgesamt vier Integertypen zur Verfügung, alle mit einem unterschiedlichen Wertebereich.

Typ	Byte	Wertebereich	Standartwert
byte	1	-2^7bis 2^7-1 -128 bis 127	0
short	2	-2^{15}bis 2^{15}-1 -32768 bis 32767	0
int	4	-2^{31}bis 2^{31}-1 -2147483648 bis 2147483647	0
long	8	-2^{63}bis 2^{63}-1 $\pm 9.223372037 * 10^{18}$	0

Tabelle 2: Integer Datentypen

3.1.3. Fliesskommazahlen Datentypen

Als Integerwerte mit Nachkommastellen kennt Java ebenfalls zwei Typen. Beide verwenden einen unterschiedlich grossen Wertebereich.

Typ	Byte	Wertebereich	Standartwert
float	4	$\pm 3.40282347 * 10^{38}$	0.0
double	8	$\pm 1.79769313486231570 * 10^{308}$	0.0

Tabelle 3: Fliesskommazahlen Datentypen

3.1.4. Zeichen Datentypen

Auch für die gängigen Zeichen stellt Java einen entsprechenden Datentyp zur Verfügung. Im Gegensatz zu anderen Programmiersprachen ist hier allerdings eine Besonderheit zu entdecken. Der gesamte Java-Zeichensatz basiert bereits auf dem Unicode-Standard, welcher mit der doppelten Byteanzahl alle Zeichentypen der Welt darstellen kann.

Typ	Byte	Wertebereich	Standartwert
char	2	alle Unicode Zeichen	\n0000

Tabelle 4: Zeichen Datentypen

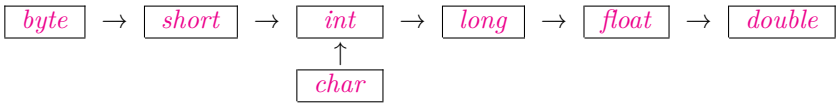
3.2. Benutzerdefinierte Datentypen

Es gibt mehrere Möglichkeiten auch eigene Datentypen zu definieren. Diese können jedoch nie den atomaren Status wie die primitiven Datentypen erreichen. Eigene Datentypen werden in Form von Klassen aufgebaut, welche wiederum die elementaren Datentypen von Java verwenden.

3.3. Standardkonvertierung

Java ist in der Lage, bereits beim Kompilieren die verschiedenen elementaren Datentypen unter Berücksichtigung ihrer Wertebereiche ineinander umzuwandeln. Dabei folgt Java einem festen Schema, um Über- und Unterläufe zu vermeiden.

Eine Konvertierung ist daher implizit nur von einem Datentyp kleineren Wertebereichs in einen Typ höheren Wertebereichs möglich, da sonst bei zu grossen Werten signifikante Stellen verloren gehen würden. Die folgende Grafik verdeutlicht diese Hierarchie.



Es ist erkennbar, dass sich nicht alle Datentypen einreihen. Boolesche Datentypen zum Beispiel lassen sich nicht in einen Wert übertragen. Der umgekehrte Weg der Konvertierung wird von Java bereits als Fehler erkannt und daher nicht *implizit* durchgeführt. Java wählt immer den nächsthöheren Datentyp.

3.4. Variablen Deklaration (definieren)

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Um Variablen zu nutzen, müssen sie deklariert werden. (In C(++) bedeuten Definition und Deklaration etwas Verschiedenes). In Java kennen wir diesen Unterschied nicht und betrachten daher beide Begriffe als gleichwertig. Die Spezifikation spricht nur von Deklarationen. Die Schreibweise einer Variablendeklaration ist immer die Gleiche: Hinter dem Typnamen folgt der Name der Variable. Sie ist eine Anweisung und wird daher mit einem Semikolon abgeschlossen. In Java kennt der Compiler von jeder Variable und jedem Ausdruck genau den Typ.

Beispiele:

```
int     age;          // Alter
double  income;       // Einkommen
char    gender;       // Geschlecht (f oder m)
boolean isPresident;  // Ist Präsident (true oder false)
int     x, y;         // Deklariert x und y als Integer
```

3.5. Variablen Initialisierung (deklaration mit Wertinitialisierung)

Den Variablen kann gleich bei der Deklaration ein Wert zugewiesen werden. Hinter einem Gleichheitszeichen steht der Wert, der oft ein Literal ist. Eine Zuweisung gilt nur für immer genau eine Variable:

```
int     age = 48;
double  income = 400000;
char    gender = 'm';
boolean isPresident = true;
// Gueltig aber unschön
double  x, y, bodyHeight = 183;
```

In Java braucht man kein Schlüsselwort wie *var*. Stattdessen gibt man den Typ der Variablen an. Für den Anfang benutzen wir *int*. *int* steht für Integer, das sind in Java positive und negative ganze Zahlen, die in 4 Bytes passen, also von -2^{31}bis 2^{31}-1. Man muss der Variable nicht sofort einen Wert zuweisen, dies muss aber geschehen bevor diese benutzt wird. (Ausnahmen, bei den Variablen, die automatisch initialisiert werden. Werden später erklärt.)

3.5.1. Erlaubte Variablennamen:

- beginnen mit einem Buchstaben (oder einem _, das ist aber nicht erwünscht)
- enthalten ansonsten nur Buchstaben und Ziffern und _ und $ ($ ist ebenfalls nicht erwünscht)
- Umlaute und ß sind erlaubt (können aber im Austausch mit anderen Betriebssystemen unter Umständen zu Problemen führen)
- Sonderzeichen wie ? ! + - etc. sind nicht erlaubt, da sie Operatoren sind

Es ist strikte Konvention, Variablennamen klein zu schreiben, innere Worte gross zu beginnen und keine _ zu verwenden, z.B. *einVariablenName*. Diese Schreibweise nennt man Camel Case.

3.5.1.1. Gute Namen, schlechte Namen Für die optimale Lesbarkeit und Verständlichkeit eines Programmcodes sollten Entwickler beim Schreiben einige Punkte berücksichtigen:

- Ein konsistentes Namensschema ist wichtig. Heisst ein Zähler no, nr, cnr oder counter? Auch sollten wir korrekt schreiben und auf Rechtschreibfehler achten, denn leicht wird aus necessaryConnection dann nesesarryConnection. Variablen ähnlicher Schreibweise, etwa counter und counters, sind zu vermeiden.
- Abstrakte Bezeichner sind ebenfalls zu vermeiden. Die Deklaration *int* TEN = 10; ist absurd. Eine unsinnige Idee ist auch die folgende: *boolean* FALSE= true, TRUE = false;. Im Programmcode würde dann mit FALSE und TRUE gearbeitet. Einer der obersten Plätze bei einem Wettbewerb für die verpfuschtesten Java-Programme wäre uns gewiss ...
- Unicode-Sequenzen können zwar in Bezeichnern aufgenommen werden, doch sollten sie vermieden werden. In *double* übelkübel, \u00FCbelk\u00FCbel; sind beide Bezeichnernamen gleich und der Compiler meldet einen Fehler.
- 0 und O und 1 und l sind leicht zu verwechseln. Die Kombination »rn« ist schwer zu lesen und je nach Zeichensatz leicht mit »m« zu verwechseln. [Eine Software wie Mathematica warnt vor Variablen mit fast identischem Namen.] Gültig – aber böse – ist auch: *int* ínt, ìnt, ît; *boolean* bôõleañ;

Beispiel:

```
int i;
```

Für diese Variable sind jetzt schon 4 Byte Platz im Arbeitsspeicher reserviert. Aber sie ist noch nicht initialisiert, d.h. ihr wurde noch kein Wert zugewiesen. Man kann sie deshalb noch nicht benutzen. Z.B.

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

würde beim Kompilieren eine Fehlermeldung “i might not have been initialized” verursachen.

In Java (und verwandten Sprachen wie C, C++, C# und JavaScript) weist man Variablen mit dem Gleichheitszeichen einen Wert zu.

```
i = 3;
```

Das ist nicht als "i ist gleich 3" zu lesen. Es ist kein Vergleich! Mögliche Leseweisen sind "setze i auf 3", "schreibe 3 in i", "i ergibt sich zu 3", "i wird 3".

```
class Variablen {
  public static void main(String[] args) {
    int i;
    i = 3;
    System.out.println(i);
  }
}
```

Gibt 3 auf der Konsole aus.

Natürlich kann man den Wert der Variablen auch ändern:

```
class Variablen {
  public static void main(String[] args){
    int i;
    i = 3;
    System.out.println(i);
    i = 5;
    System.out.println(i);
  }
}
```

Gibt 3 und eine Zeile tiefer 5 aus.

Die Definition der Variablen und die Initialisierung (erste Zuweisung eines Wertes) kann in einer Zeile erfolgen:

```
int i = 3;
```

Aber Achtung, nicht ein zweites Mal `int i=5;` schreiben. Damit würde man eine zweite Variable namens i definieren. Das ist nicht möglich. Der Compiler würde eine Fehlermeldung ausgeben, die besagt, dass die Variable i schon existiert.

```
class Variablen {
  public static void main(String[] args){
    int i = 3;
    System.out.println(i);
    int i = 5; // Fehler, doppelter Variablenname
    System.out.println(i);
  }
}
```

Natürlich kann man die üblichen Berechnungen ausführen.

```
class Variablen {
  public static void main(String[] args){
    int a = 17;
    int b = 3;
    int summe = a + b;
    System.out.print("a+b=");
    System.out.println(summe);
    int differenz = a - b;
    System.out.print("a-b=");
    System.out.println(differenz);
    int produkt = a * b;
    System.out.print("a*b=");
    System.out.println(produkt);
    int quotient = a / b;
    int rest = 17 % 3;
    System.out.print("a:b=");
    System.out.print(quotient);
    System.out.print("␣Rest␣");
    System.out.println(rest);
  }
}
```

An diesem Beispiel sieht man:

- Variablen können überall definiert werden, nicht nur am Anfang der Methode (Funktion).
- / ist eine ganzzahlige Division, wenn man zwei ints (ganze Zahlen) dividiert. Deshalb ergibt 17/3 das Resultat 5.
- % (Modulo)(Prozentzeichen, drücke Shift + 5) ermittelt den Rest bei der ganzzahligen Division. Deshalb ergibt 17%3 das Resultat 2. (Das Vorzeichen des Rests richtet sich nach dem Vorzeichen des Dividenden (= was vor dem % steht, also die Zahl, die geteilt wird). 17%3 und 17% − 3 ergibt 2, −17%3 und −17% − 3 ergibt -2.)

Anmerkungen:

Man muss keine Leerzeichen vor und nach dem Variablennamen setzen. Da =, +, - etc. keine erlaubten Symbole in Variablennamen sind, weiss der Compiler, dass dort der Variablenname zu Ende ist. z.B. `int summe=a+b;` `int summe = a + b ;`

Mehrere Zuweisungen in einem Schritt Zuweisungen der Form

```
a = b = c = 0;
```

sind erlaubt und gleichbedeutend mit den drei Anweisungen

```
c = 0;
b = c;
a = b;
```

Die explizite Klammerung a = (b = (c = 0)) macht noch einmal deutlich, dass sich Zuweisungen verschachteln lassen und Zuweisungen wie c = 0 Ausdrücke sind, die einen Wert liefern. Doch auch dann, wenn wir meinen, dass

```
a = (b = c + d) + e;
```

eine coole Vereinfachung im Vergleich zu

```
b = c + d;
a = b + e;
```

ist, sollten wir mit einer Zuweisung pro Zeile auskommen.

Die Reihenfolge der Auswertung zeigt anschaulich folgendes Beispiel:

```
int b = 10;
System.out.println( (b = 20) * b ); // gibt was aus?
```

4. Ausdrücke, Operanden und Operatoren

Beginnen wir mit mathematischen Ausdrücken, um dann die Schreibweise in Java zu ermitteln. Eine mathematische Formel, etwa der Ausdruck –27 * 9, besteht aus Operanden (engl. operands) und Operatoren (engl. operators). Ein Operand ist eine Variable oder ein Literal. Im Fall einer Variable wird der Wert aus der Variable ausgelesen und mit ihm die Berechnung durchgeführt.

```
int i = 12;
int j;
j = i * 2;
```

Die Multiplikation berechnet das Produkt von 12 und 2 und speichert das Ergebnis in j ab. Von allen primitiven Variablen, die in dem Ausdruck vorkommen, wird also der Wert ausgelesen und in den Ausdruck eingesetzt.

Dies nennt sich auch Wertoperation, da der Wert der Variable betrachtet wird und nicht ihr Speicherort oder gar ihr Name.

Die Arten von Operatoren Operatoren werden nach der Anzahl Ihrer Operanden unterschieden zwischen

- unäre Operatoren (= ein Operand),
- binäre Operatoren (= zwei Operanden) und
- ternäre Operatoren (= drei Operanden).

4.1. Arithmetische Operatoren

Operator	Name	Typ	Beispiel / Erklärung	Rang
+	Positives Vorzeichen	unär	+1*-1=-1	3
-	Negatives Vorzeichen	unär	-1*1=-1	3
+	Plus-Operator	binär	1+1	2
-	Minus-Operator	binär	1-1	2
*	Multiplikations-Operator	binär	1*1	2
/	Divisions-Operator	binär	1/1	2
%	Modulo-Operator	binär	Rest bei Division 5%2=1, weil 5/2=2 Rest 1	2
++	Inkrement	unär	Addition um 1 a++ entspricht a=a+1	1
--	Dekrement	unär	Negation um 1 a-- entspricht a=a-1	1

Tabelle 5: Arithmetische Operatoren

Die arithmetischen Operatoren können nur auf numerische Variablen angewendet werden, wie Integer, Double, aber auch Character.

4.2. Relationale Operatoren

Relationale Operatoren vergleichen zwei Werte miteinander. Das Ergebnis ist immer ein boolescher Wert, also *true* oder *false*.

Zu den relationalen Operatoren zählen:

Operator	Name	Rang
==	gleich	6
!=	ungleich	6
>	grösser als	5
<	kleiner als	5
>=	grösser als oder gleich	5
<=	kleiner als oder gleich	5

Tabelle 6: Relationale Operatoren

Insbesondere bei den Vergleichen == und != tritt oft nicht das erwartete Resultat ein. Da anstelle der Werte von Variablen die Speicheradressen verglichen werden. Oder bei Fliesskomma-Werten durch Rundungsfehler eine Ungenauigkeit eingeschleppt wird.

4.3. Logische Operatoren

Logische Operatoren verknüpfen Wahrheitswerte miteinander und haben als Resultat einen Wahrheitswert (*boolean*). Die weitaus häufigste Verwendung findet sich in der if-Anweisung.

Operator	Name	Beispiel / Erklärung	Rang
!	NOT	Negation / Umkehrung	1
&	AND	Und [5]	7
|	OR	Oder [5]	9
^	XOR	Exklusives oder [5]	8
&&	short-circuit AND	Doppeltes UND ist eine logische UND-Verknüpfung, bei der wir nur ein wahres Ergebnis erhalten, wenn beide Werte wahr sind. Ist an dieser Stelle bereits der erste Operator falsch (false) so wird der zweite Operand nicht mehr ausgewertet, da false und irgendwas bei einer logischen UND-Verknüpfung als Resultat immer false ist.	10
||	short-circuit OR	Doppeltes ODER ist eine logische ODER-Verknüpfung, bei der wir nur ein falsches Ergebnis erhalten, wenn beide Werte falsch sind. Ist an dieser Stelle bereits der erste Operator wahr (true) so wird der zweite Operand nicht mehr ausgewertet, da true und irgendwas bei einer logischen ODER-Verknüpfung als Resultat immer true ist.	11

Tabelle 7: Logische Operatoren

[5] Der Vollständigkeit halber.

4.3.1. Erklärungen zu short-circuit

Die beiden logischen Operatoren && und || 'short-circuit' evaluieren die rechte Seite nicht wenn dies nicht notwendig ist.
Falls die beiden Operatoren & und | als logischen Operatoren benutzt werden so evaluieren diese immer beide Seiten.

- `false` && ... - es ist nicht notwendig zu wissen was auf der rechten Seite steht, dass Resultat ist immer `false`
- `true` || ... - es ist nicht notwendig zu wissen was auf der rechten Seite steht, das Resultat ist immer `true`.

4.4. Bitweise Operatoren[6]

Bitoperanden manipulieren einzelne Bit der Operanden.

Operator	Name	Beispiel / Erklärung	Rang
~	Bitkomplement	unäres "nicht", invertiert alle Bits seines Operanden	1
&	AND	wenn beide Operanden 1 sind, wird ebenfalls eine 1 produziert, ansonsten eine 0	7
|	OR	produziert eine 1, sobald einer seiner Operanden eine 1 ist	9
^	Exklusives OR	wenn beide Operanden den gleichen Wert haben, wird eine 0 produziert, ansonsten eine 1	8
>>	Shift right	alle Bits des Operanden werden um eine Stelle nach rechts verschoben	4
>>>	Shift right	Rechtsverschiebung mit Auffüllung von Nullen	4
<<	Shift left	Linksverschiebung	4
<<<	Shift left	Linksverschiebung mit Auffüllung von Nullen	4

Tabelle 8: Bitweise Operatoren

4.4.1. Aufgaben:

4.4.1.1. Rechenfehler? Überlege dir was die Ausgabe von

```
double d=1/2;
System.out.println(d);
```

sein könnte. Kontrolliere deine Überlegung mit Eclipse!

4.4.1.2. Rechenfehler zum zweiten. Erkläre die erstaunliche Ausgabe von

```
float preis = 9.9f;
float einnahmenBei100Kunden = preis * 100;
System.out.println(einnahmenBei100Kunden);
```

4.4.1.3. Aufgabe 1 Es soll ein Programm geschrieben werden welches die Anzahl Sekunden für einen Tag, Monat, Jahr errechnet und die Resultate ausgibt.

[6] Die bitweisen Operatoren werden in Java nur selten eingesetzt.

4.4.1.4. Note Berechnen

In der Regel werden Noten so berechnet, dass rund 60% der maximalen Punktezahl die Note 4 ergibt.

Dies wird wie folgt gerechnet:

$$\frac{erzielte Punktzahl * 5}{maximale Punktzahl} + 1$$

Es soll ein Programm geschrieben werden welches aus der erreichten Punktzahl und der maximalen Punktzahl die entsprechende Note errechnet.

5. Kontrollstrukturen

Wikipedia. Kontrollstrukturen (Steuerkonstrukte) sind Anweisungen in imperativen Programmiersprachen. Sie werden verwendet, um den Ablauf eines Computerprogramms zu steuern. Eine Kontrollstruktur ist entweder eine Verzweigung oder eine Schleife. Meist wird ihre Ausführung über logische Ausdrücke der booleschen Algebra beeinflusst. Kontrollstrukturen können über spezielle Diagramme visualisiert werden (z. B. Nassi-Shneiderman-Diagramme[1][7]).

5.1. Verzweigung (if)

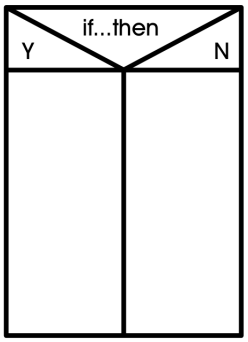


Abbildung 1: if nach Nassi-Shneidermann

Eine einfache Verzweigung wird durch das "if-then-else"-Konstrukt gelöst, wobei lediglich if und else in Java Schlüsselwörter sind.

```
if(<boolescher Ausdruck>){
  // Anweisung(en)
} else {
  // Anweisung(en)
  // Anweisung(en)
}
```

Beispiel:

```
int z1=5;
int z2=20;
int z3=300;

if(z1 < z2){
  System.out.println("Zahl 1 ist kleiner als Zahl 2");
}else{
  System.out.println("Zahl 2 ist kleiner oder gleich als Zahl 1");
}

// Funktioniert auch ohne {}
if(z1 > z2)
```

[7] Aus PROGRAM DESIGN von Alex Abrecht

```
  System.out.println("Zahl 2 ist kleiner als Zahl 1");
else
  System.out.println("Zahl 1 ist kleiner als oder gleich Zahl 2");

// Mehrfach
if(z1 < z2){
  System.out.println("Zahl 1 ist kleiner als Zahl 2");
}else if(z1 < z3){
  System.out.println("Zahl 1 ist kleiner als Zahl 3");
}else{
  System.out.println("nur wenn z1 > z2 UND z1 > z3 ist");
  System.out.println("wird dieser Text ausgegeben.");
}
```

5.1.1. Aufgaben:

5.1.1.1. Maximum zweier Zahlen Gegeben sind die beiden int-Variablen a und b. Es soll ein Programm erstellt werden, welches immer zuerst die grössere der beiden ausgibt.

5.1.1.2. Zwei Zahlen sortieren Gegeben sind die beiden Zahlen a und b. Es soll ein Programm erstellt werden, welches den grösseren der beiden Werte in die Variable a und den kleineren in die Variable b schreibt. Zur Kontrolle sollen die beiden Variablen ausgegeben werden.

5.1.1.3. Drei Zahlen sortieren Gegeben sind die drei Variablen a, b und c. Es soll ein Programm erstellt werden, welches die Variablen aufsteigend sortiert. Vertauschen Sie die Werte dieser Variablen so, dass zum Schluss $a < b < c$ gilt. Prüfen Sie Ihre Methode mit allen Permutationen der Zahlen 1, 2 und 3 (es gibt insgesamt 6 Kombinationen).

5.2. Mehrfachverzweigung (switch)

in Vorbereitung...

5.3. Schleife mit Anfangsprüfung (Kopfgesteuerte Schleife)

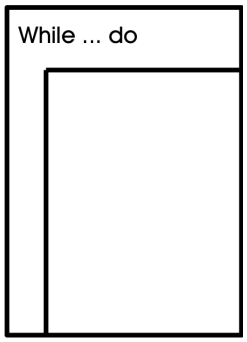


Abbildung 2: while nach Nassi-Shneidermann

In den meisten Programmiersprachen gibt es die While-Schleife als Kontrollstruktur. Sie dient dazu, eine Abfolge von Anweisungen solange auszuführen, wie die Bedingung erfüllt ist. Diese Bedingung wird geprüft, bevor die Anweisungsfolge abgearbeitet wird. Es kann also auch sein, dass die Abfolge gar nicht ausgeführt wird. Wenn die Bedingung ständig erfüllt ist, dann wird die Schleife zur Endlosschleife.

```
while(<boolescher Ausdruck>){
  // das, was wiederholt werden soll
}
```

5.3.1. Aufgaben:

5.3.1.1. Versteckspiel Als Suchender muss man zunächst bis 100 Zählen, dann laut „Ich komme...“ rufen. Erst dann darf man zum Suchen aufbrechen.

Dieses simple Spiel ist ein einfacher Algorithmus:

```
VERSTECKSPIELZÄHLUNG
Die erste Zahl ist 1.
Prüfe, ob die 100 erreicht wurde.
Wenn das noch nicht erfüllt ist, sage die Zahl und erhöhe die Zahl um eins.
Wenn doch, dann rufe "Ich komme"
```

Erstelle ein entsprechendes Java Programm.

5.3.1.2. Versteckspiel zum zweiten

- Lass die Zählung rückwärtslaufen.
- Lass die Zählung in zweier und dreier Schritten laufen.

5.3.1.3. Zusammenzählen Es soll ein Programm erstellt werden, welches die ersten 100 Zahlen zusammenzählt und das Resultat ausgibt.

5.3.1.4. Fakultät Es soll ein Programm erstellt werden, welches die Fakultät einer gegebenen Zahl berechnet.

5.4. Schleife mit Endprüfung (Fussgesteuerte Schleife)

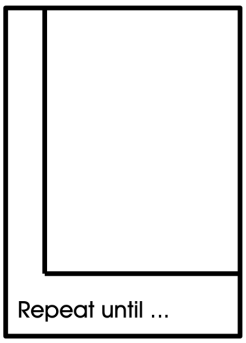


Abbildung 3: do..while nach Nassi-Shneidermann

Die do..while Schleife ist eine weitere Variante der while-Schleife, die jedoch nur durch ein kleines Detail im Programmverlauf und in der Syntax abweicht. Der Unterschied zum Original besteht darin, dass der Code der Schleife mindestens einmal ausgeführt wird, bevor die eigentliche Bedingung zum ersten mal getestet wird.

Ein sinnvolles Beispiel, wo es einen Nutzen bringt eine do-while-Schleife einzusetzen, wäre ein Programm zu wiederholen, bis der Benutzer die Frage, ob er es noch einmal ausführen will mit nein beantwortet ;-)

```
do{
  // das, was wiederholt werden soll
}while(<boolescher Ausdruck>);
```

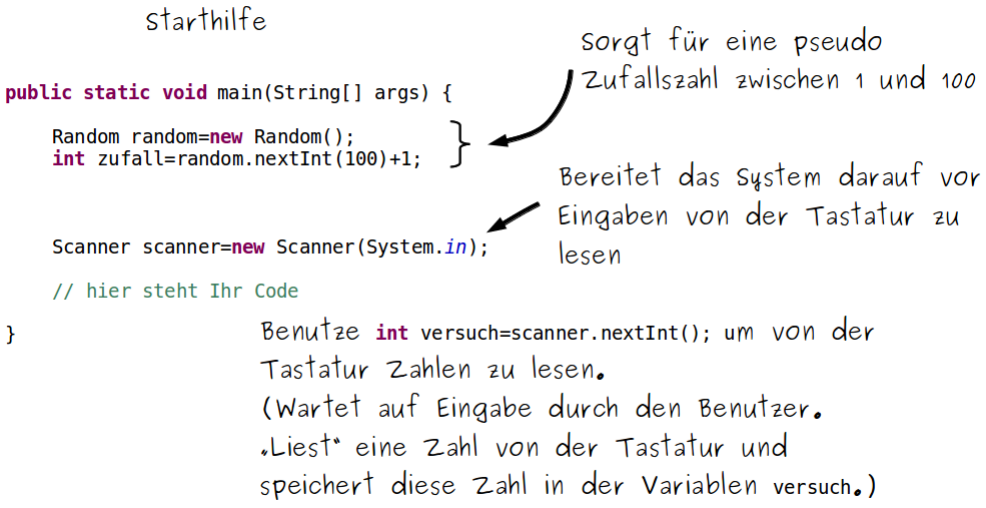
5.4.1. Aufgaben:

5.4.1.1. Fibonacci Es soll ein Programm erstellt werden, das die Fibonacci -Reihe bis zu einer gegebenen Obergrenze ausgibt. Die Fibonacci-Reihe errechnet sich wie folgt.

Die ersten zwei Elemente werden mit 1 initialisiert jedes weitere Element wird durch Addition der beiden vorausgegangenen Elemente berechnet. Z.B.:

1 1 2 3 5 8 13 21..

5.4.1.2. Ratespiel Es soll ein Programm erstellt werden, welches ein Ratespiel nachstellt. Dabei wird eine Zufallszahl erstellt. Der Spieler wird wiederholt nach einem Tip gefragt, bis er durch Raten die Zufallszahl gefunden hat.




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

    Random random=new Random();
    int zufall=random.nextInt(100)+1;


    Scanner scanner=new Scanner(System.in);

    // hier steht Ihr Code
}
```

Listing 1: Starthilfe zu Ratespiel

5.5. fixe Schleife for

Die sogenannte for-Schleife ist die wichtigste und flexibelste Schleife ihrer Art. Gegenüber anderen Schleifen setzt sie sich aus drei Parametern zusammen.

Im ersten Schritt werden alle Variablen erzeugt, was jedoch nur einmalig beim Eintritt in die Schleife geschieht. Danach wird die eigentliche Bedingung getestet und in Abhängigkeit des Ergebnisses der Anweisungsbock ausgeführt. Letztlich werden die Variablen aktualisiert (meist inkrementiert beziehungsweise dekrementiert).

Diese Prozedur wird so oft durchlaufen, wie die Bedingung *true* ergibt.

```
for(StartWert; Bedingung; Update){
  // das, was wiederholt werden soll
}

for(int i = 0; i < 10; i++){
  System.out.println(i);
}
// Beziehung zur while
int i = 0;
for(; i < 10;){
  System.out.println(i);
   i++;
}

// Schleife in der Schleife
for(int i = 0; i < 5; i++){
  for(int j = 0; j < 10; j++){
     System.out.println("i ist nun "+i+" und j ist "+j);
  }
}
```

5.5.1. Aufgaben:

5.5.1.1. Verschiedene Schrittweiten Es soll ein Programm erstellt werden, welches in 2er Schritten auf 100 Zählt.

5.5.1.2. Ein-mal-Eins auf dem Bildschirm ausgeben zum Ersten Es soll ein Programm erstellt werden, welches das kleine Einmaleins ausgibt. Hierbei müssen zwei verschachtelte Schleifen ver-

wendet werden (also eine Schleife innerhalb eines Blocks einer anderen Schleife). Die Ausgabe des Programms soll dann ungefähr so aussehen:

```
1 * 1 =   1
2 * 1 =   2
3 * 1 =   3
  ...
9 * 1 =   9
10 * 1 =  10
1 * 2 =   2
2 * 2 =   4
  ...
9 * 10 =  90
10 * 10 = 100
```

5.5.1.3. Ein-mal-Eins auf dem Bildschirm ausgeben zum Zweiten Schreiben Sie ein Java-Programm, welches das grosse Ein-Mal-Eins berechnet und tabellarisch auf dem Bildschirm ausgibt. Um die auszugebenden Zahlwerte geeignet einzurücken, sollten Sie bei der Ausgabe den Tabulator "\t" verwenden. Verwenden Sie die for-Schleife.

Beim grossem Ein-Mal-Eins werden alle Produkte i * j mit 0 < i <= 10 und 0 < j <= 10 gebildet. Die Ausgabe sollte also etwa wie folgt aussehen:

```
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

5.5.1.4. Tannenbaum Schreiben Sie ein Programm, welches mit Hilfe von Schlaufen folgende Figur ausgibt.

```
   *
  ***
 *****
*******
   |
   |
   |
   |
```

5.6. foreach

5.6.0.1. Überlegungen zu Schlaufen Vergleichen Sie die drei verschiedenen Schlaufen while , do-while und for. Prüfen Sie für sich selbst welche Ihnen am besten zusagt. Begründen und diskutieren Sie Ihre Ansichten mit einem Kollegen.

6. Array[8]

In der Informatik ist eine Datenstruktur eine bestimmte Art Daten zu verwalten und miteinander zu verknüpfen. Auf die Daten in einer Datenstruktur kann zugegriffen werden und die Daten können manipuliert werden. Datenstrukturen sind immer mit bestimmten Operationen verknüpft, um eben diesen Zugriff und diese Manipulation zu ermöglichen.

Das Array (auch »Feld« oder »Reihung« genannt) ist die einfachste verwendete Datenstruktur. Es werden hierbei mehrere Variablen vom selben Basisdatentyp gespeichert. Ein Zugriff auf die einzelnen Elemente wird über einen Index möglich. Im eindimensionalen Fall wird das Array häufig

[8] Ein Grossteil wurde von ZUM kopiert

als Vektor und im zweidimensionalen Fall als Tabelle oder Matrix bezeichnet. Arrays sind aber keinesfalls nur auf zwei Dimensionen beschränkt, sondern werden beliebig mehrdimensional verwendet. Wegen ihrer Einfachheit und grundlegenden Bedeutung bieten die allermeisten Programmiersprachen eine konkrete Umsetzung dieser Datenstruktur als zusammengesetzten Datentyp "Array" im Sprachumfang an.

6.1. Deklaration von Arrays

Arrays werden ähnlich wie die primitiven Variablen deklariert.

```
int[] einArray;
```

Die beiden eckigen Klammern zeigen an, dass es sich um ein Array handelt. Beim deklarieren eines Arrays wird kein Speicher für die Elemente alloziert, da an dieser Stelle noch nicht bekannt ist für wie viele Elemente dies denn erfolgen sollte.

6.2. Initialisierung von Arrays

Bei der Initialisierung von Arrays geht es hauptsächlich darum die Grösse zu bestimmen, damit ein zusammenhängender Speicherplatz reserviert werden kann.

6.2.0.1. Möglichkeit 1 mit Aufzählung der Elemente

Dabei werden die konkreten Elemente aufgezählt.

```
int[] array={i, j, k, ..}
```

wobei in diesem Beispiel i,j,k, .. vom Typ *int* sein müssen.
Als Beispiel die Anzahl der Tage innerhalb der Monate:

```
int[] array={31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31};
```

oder eine Liste mit den 7 Wochentagen

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

6.2.0.2. Möglichkeit 2 mit dem *new* Operator (bevorzugt)

Dabei wird zuerst der Speicher für die geforderte Anzahl der Elemente belegt und diese Elemente inizialisiert.

```
int[] array=new int[n];
```

wobei n >= 0;($n \in N$)
Erst danach wird den Elementen ein Wert zugewiesen.

```
// Eine Liste mit den 7 Wochentagen...
String[] wochenTage=new String[7];
wochenTage[0]="Montag";
wochenTage[1]="Dienstag";
wochenTage[2]="Mittwoch";
wochenTage[3]="Donnerstag";
wochenTage[4]="Freitag";
wochenTage[5]="Samstag";
wochenTage[6]="Sonntag";
```

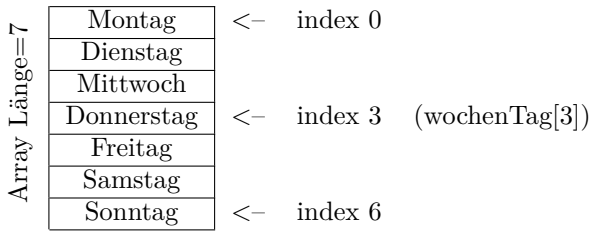
6.3. Zugriff auf Array Elemente

Da Java die Code Base 0 hat, werden die Elemente von 0 bis n-1 durchnummeriert. Hat also ein Array zwölf Elemente sind diese von 0 bis 11 durchnummeriert.

```
String[] s={"Januar","Februar","März",
            "April","Mai","Juni",
            "Juli","August","September",
            "Oktober","November","Dezember"};
System.out.println(s[2]); // Februar?
```

veranschaulicht:

String[] wochenTage=String{"Montag", ... , "Sonntag"};



Der Zugriff erfolgt also wie folgt:

```
String tmp=wochenTage[0]; // tmp enthaellt jetzt Montag
wochenTage[0]="Mo";       // Der Wert an der Index Stelle 0
                          // wurde von Montag nach Mo geaendert.
```

6.4. Traversieren von Arrays

Eine der wirklichen Stärke eines Array liegt in der Verarbeitung vieler Werte der Reihe nach. Das serielle Abarbeiten aller Elemente nennt sich traversieren. Am einfachsten kann ein Array mit einer for-Schleife traversiert werden.

```
String[] wt={"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};

for (int i = 0; i < wt.length; i++) {
  System.out.println( wt[i] );
}
// oder mit der foreach - Schlaufe (werden wir nicht behandeln)
for (String string : wt) {
  System.out.println(string);
}
```

6.4.1. Aufgaben:

6.4.1.1. In einem Array suchen, 1 Es soll aus einem selbstgewählten *int* Array das Element mit dem grössten Zahlenwert gefunden und ausgegeben werden.

6.4.1.2. In einem Array suchen, 2 Es soll aus einem selbstgewählten *int* Array das Element mit dem kleinsten Zahlenwert gefunden und sowohl die Position innerhalb des Arrays als auch dessen Wert ausgegeben werden.

6.4.1.3. Zufalls Array Es soll ein Array mit 10 Elementen erstellt werden, welches zufällige *int* Werte enthält. Pseudo Zufallszahlen werden mit Hilfe von

```
Random random = new Random();
int zufall1= random.nextInt();   // zufällige int Zahl
int zufall2= random.nextInt(6)+1;// 'Würfel Wurf'
```

erstellt.
Teste ob die beiden ersten Übungen auch mit zufälligen Arrays funktionieren.

6.4.1.4. Wetterstation (Diese Aufgabe wurde von Freitagsrunde kopiert)
Eine Wetterstation hat für 14 Tage folgende Temperaturwerte aufgenommen.

Tag	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Temperatur	12	14	9	12	15	16	15	15	11	8	13	13	15	12

- Speichere die Tabelle mit einem geeigneten Datentyp.
- Schreibe ein Programm, welches die Tabelle schön tabellarisch auf der Konsole ausgibt.
- Schreibe ein Programm, welches die Durchschnittstemperatur für die zwei Wochen bestimmt.
- Schreibe ein Programm, welches die maximale und minimale Temperatur ausgibt.
- Schreibe ein Programm, welches die beiden aufeinanderfolgenden Tage angeben kann, an denen es den grössten Temperaturumschwung gab.

6.4.1.5. Primzahlensieb Es gibt viele Möglichkeiten Primzahlen zu finden. Eine sehr einfache Art und Weise stellt das Primzahlensieb dar. Es soll ein Programm geschrieben werden welches die Primzahlen zwischen 0 und 100 findet (Funktioniert bestens auch von Hand).

```
1. Schreibe alle natürlichen Zahlen von 2 bis 100 auf.
2. Streiche alle Vielfachen von 2 heraus.
3. Gehe zur nächst grösseren nicht gestrichenen Zahl
   und streiche deren Vielfache heraus.
4. Wiederhole 3. sooft es geht.
5. Die übriggebliebenen Zahlen sind Primzahlen.
```

6.4.1.6. Sortieren Es soll ein *int* Array mit 10 Elementen sortiert werden.

- Versuchen Sie zuerst anhand von Spielkarten das Sortieren aus.
- Verfolgen Sie das Konzept von "divide and conquer" (teile und herrsche), die Überlegungen an lediglich 2 oder 3 Spielkarten. (Denken Sie an Schlaufen um einfache Dinge wiederhollt ausführen zu können.)
- Versuchen Sie verschiedene Methoden und finden Sie eine möglichst einfache Art und Weise.
- Erstellen sie einen Pseudocode, welcher sortiert.
- Implementieren sie ihren Pseudocode in Java

6.4.1.7. Römische Zahlen Es soll ein Programm erstellt werden welches Römische Zahlen in Arabische umwandelt.

Subtraktionsregel - Römische Zahlen umrechnen Neben der einfachen Umrechnung der Römischen Zahlen existiert auch eine verkürzte, heute verwendete Schreibweise, die sogenannte Subtraktionsregel. Der Grundgedanke der Subtraktionsregel ist, dass vier gleiche Zahlzeichen vermieden werden, indem eine kleinere Zahl vor eine größere geschrieben wird. Ein einfaches Beispiel dafür ist die 4. Sie wird durch IV dargestellt, was wörtlich mit fünf weniger eins übersetzt werden könnte. Für die subtrahierende Abbildung im Römischen Zahlensystem gelten folgende Regeln:

1. Regel: I steht nur vor V und X
2. Regel: X steht nur vor L und C
3. Regel: C steht nur vor D und M

1	2	3	4	5	6	7	8	9	10	40	50	100	500	1000
I	II	III	IV	V	VI	VII	VIII	IX	X	XL	L	C	D	M

Tabelle 9: Spickzettel Römische Zahlen

7. Klassen

Die Klasse ist wohl das wichtigste Sprachelement von Java. Zum Ersten ist eine Klasse ein selbst definierter (eigener) Datentyp. Zum Zweiten wird mit der Klasse die Idee, Daten und deren Verarbeitung nahe beieinander zu haben, umgesetzt.

Bisher haben wir bei unseren Beispielprogrammen stets Klassen verwendet, deren genaue Syntax uns bisher unklar war. Nun wollen wir alle Details der Programmierung eigener Klassen klären.

Als Beispiel wollen wir eine Klasse *Fahrzeug* erstellen.

```
public class Fahrzeug {
  double verbrauch;
  double fuellStand;
  int achsenAnzahl=2;
  String farbe;
}
```

Listing 2: Klasse Fahrzeug

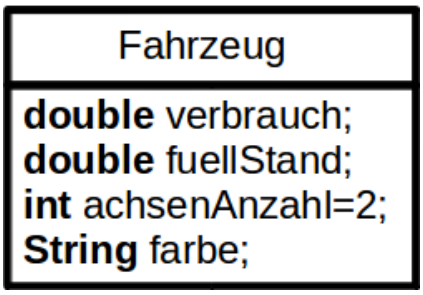
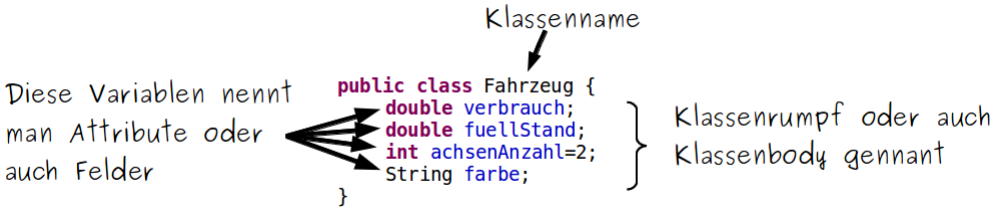


Abbildung 4: Klasse Fahrzeug

Die Deklaration beginnt mit dem Schlüsselwort *class*. Darauf folgt der Name der Klasse und in geschweiften Klammern der Klassenkörper. Die Variablen einer Klasse nennt man Felder oder Attribute. Diese Klasse kann nun deklariert werden.



Listing 3: Klasse unter der Lupe

```
Fahrzeug f;
```

Abbildung 5: f im Speicher

Beachte, dass dabei die internen Variablen NICHT gesetzt werden. Erst mit dem Schlüsselwort *new* wird Speicher für die internen Variablen belegt.

Kurze schreibweise:

```
Fahrzeug f=new Fahrzeug();
```

Dies bestimmt die Variable *f* vom Typ *Fahrzeug* und belegt (Instanziert) Speicher.

Üblicherweise wird jede Klasse in ihrer eigenen Datei plaziert. Also unsere Klasse Fahrzeug wird in die Datei Fahrzeug.java geschrieben. Dabei achtet Java streng auf präzise Schreibung. Die Benutzung kann folgendermassen erfolgen:

```
f=new Fahrzeug();
```

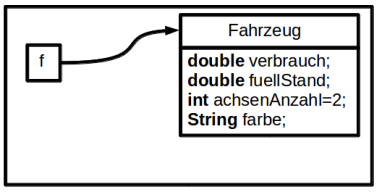


Abbildung 6: f im Speicher

```
public class AndereKlasse {
  public static void main(String[] args) {
    Fahrzeug auto1=new Fahrzeug();
    auto1.achsenAnzahl=5;
    auto1.farbe="rot";
    auto1.verbrauch=8;
  }
}
```

Listing 4: Klasse, welche die Klasse Fahrzeug benutzt

Die Variable *auto1* ist vom Typ *Fahrzeug* und besteht aus den Feldern *verb*rauch, *fuellStand*, *achsenAnzahl* und *farbe*. Auf die Felder wird zugegriffen indem zwischen dem Namen der Variable und dem Feld ein Punkt als Trenner verwendet wird. Man qualifiziert die Feldnamen mit dem Variablennamen. Der . (Punkt) Operator nennt sich Komponentenselektionsoperator weil mit dessen Hilfe die innere Komponente selektiert wird.

Beachte! Die Klasse Fahrzeug3 befindet sich in der Datei Fahrzeug.java. Die Benutzung der Klasse geschieht in unserem Beispiel aus der Klasse AndereKlasse4 welche sinngemäss in der Datei AndereKlasse.java abgelegt ist. Wird ein weiteres Fahrzeug benötigt wird, dieses ganz einfach deklariert und alloziert und damit initialisiert.

```
Fahrzeug auto1  =new Fahrzeug();
Fahrzeug einMini=new Fahrzeug();
Fahrzeug auto2  =new Fahrzeug();
```

Listing 5: Mehrere Fahrzeuge

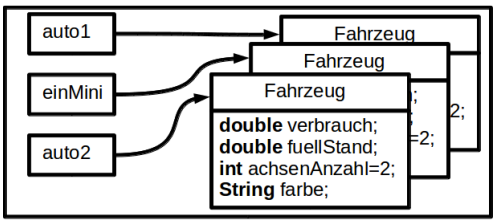


Abbildung 7: Mehrere Fahrzeuge im Speicher

Im Wesentlichen wird gleich wie mit den primitiven Datentypen vorgegangen, einzig wird für die Speicher-Allozierung das Schlüsselwort *new* verwendet. Unerwartet kompliziert wird es allerdings bei Vergleichen.

7.0.0.1. Vergleiche

Der Vergleich (und häufiger Fehler)

```
if(auto1 < einMini){
  ...
}
```

ist in Java unsinnig da dabei die Speicheradresse von *auto1* mit der Speicheradresse von *einMini* verglichen wird![9] Mit Speicheradressen wollen wir uns nicht herumschlagen, in Java ist es dem

[9] Ein Vergleich auf auto1==einMini macht Sinn, um zu testen ob beide Variablen auf das selbe Objekt weisen.

Entwickler egal wo im Speicher die Variablen abgelegt werden! Vermutlich wollte der Entwickler die Anzahl der Fahrzeugachsen oder das Gewicht vergleichen, ungefähr so:

```
if(auto1.achsenAnzahl < einMini.achsenAnzahl){
  ...
}
```

Listing 6: Vergleich von Klassen Attributen

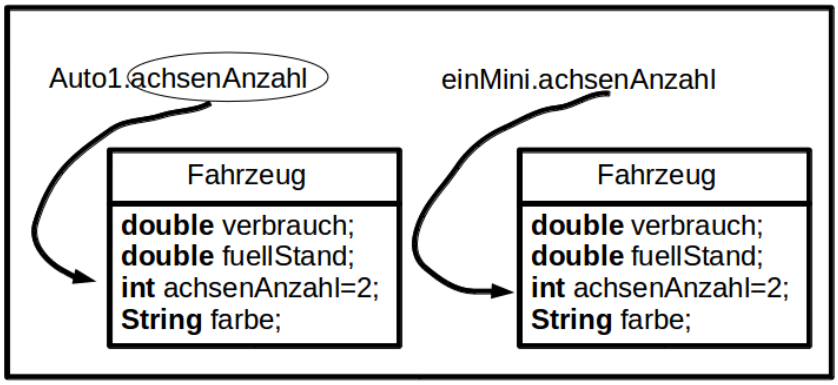
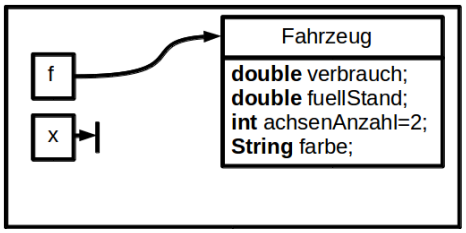


Abbildung 8: Klasse Fahrzeug im Speicher

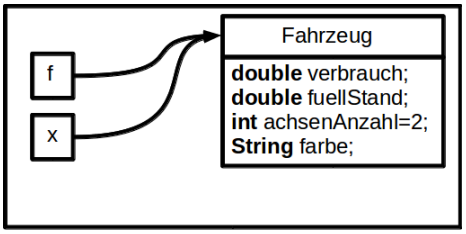
7.0.0.2. Zuweisungen zwischen Objektvariablen

Objektvariablen können einander zugewiesen werden.

```
Fahrzeug f=new Fahrzeug();
Fahrzeug x;
```




```
Fahrzeug f=new Fahrzeug();
Fahrzeug x;
x=f;
```



Die beiden Variablen f und x zeigen nun auf dasselbe Objekt. Wird nun mit

```
f.verbrauch=9;
```

der internen Variable (Attribut) ein neuer Wert zugewiesen so ergibt ab sofort

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

die Ausgabe 9, da beide Variablen auf dasselbe Objekt verweisen.

7.1. Attribute (Instanzvariablen)

Ein Attribut speichert die Eigenschaft eines Objektes. Attributwerte werden in Variablen gespeichert. Der Unterschied zwischen einem Attribut und einer Variablen ist dabei einzig der Ort an dem die

Variable deklariert wird.

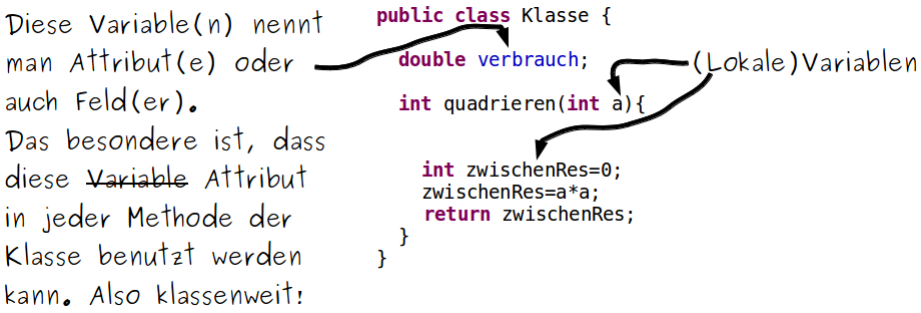


Abbildung 9: Attribute versus Variablen

7.1.1. Aufgaben:

7.1.1.1. Erstelle einer Klasse Erstelle eine Klasse *Fahrzeug*. Das Fahrzeug soll mehrere Attribute haben. Schreibe eine zweite Klasse, welche mehrere *Fahrzeuge* instanziert (in den Arbeitspeicher bringt).

7.1.1.2. Verändern von Attributen Erweitere die beiden Klassen aus der vorangegangenen Übung so, dass ein Fahrzeug während dem Programmablauf die Farbe wechselt und betankt wird.

7.1.1.3. Versuche ein Array von Fahrzeugen zu erstellen Es soll ein Array mit 10 Fahrzeugen erstellt werden.

7.1.1.4. Überlegungen So wie unsere Klasse Fahrzeug implementiert ist, ist es problemlos möglich unser Fahrzeug mit 1000 Litern zu betanken! Darf ein Auto mit 1000 Litern betankt werden? Oder darf ein Fahrzeug mit -200 Litern betankt werden wenn nur 50 Liter möglich sind? ;-)

7.2. Methoden in Klassen

Programme werden in der Praxis zu gross um an einem Stück geschrieben zu werden. Nach dem Grundsatz “Divide et impera” zerlegt man daher Programme in kleinere Anweisungsfolgen, welche logisch zusammen gehören und vergibt diesen einen Namen. Diese Anweisungsfolgen, können dann beliebig oft unter diesem Namen aufgerufen werden. Solche benannte Anweisungsfolgen nennt man in Java Methoden.

Methoden sind Funktionen innerhalb von Klassen, und weil bei Java alle Funktionen in Klassen stehen und nie ausserhalb, gibt es bei Java nur Methoden. Ein anderer Begriff für Methode ist Elementfunktion.

Andere Begriffe für Funktion sind Prozedur, Subroutine und Unterprogramm. Je nach Art der Programmiersprache werden andere Begriffe verwendet. Es gibt auch da kleine Unterschiede, die hier aber egal sind.

Methoden werden in Java wie folgt geschrieben:

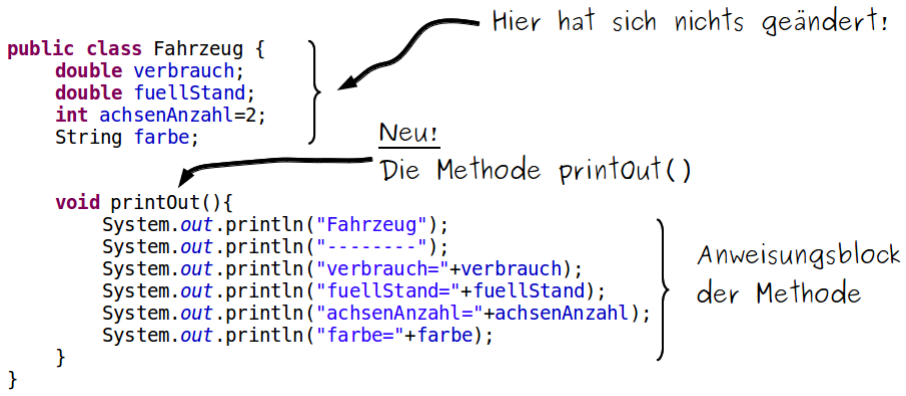
```
[RückgabeTyp] Methodenname(Parameterliste) {Anweisungsblock}
```

Eine Methode haben wir bereits in den vorangegangenen Kapiteln kennen gelernt.

```
void main(String[] args){ ... }
```

Methoden sind die Arbeitstiere in der Programmierwelt, in den Methoden steht der grösste Teil des Codes. Aus diesem Grund haben die Methoden vielfach Verben als Namen, ganz nach dem Moto *tueDies*() oder *tueDas*() oder *quadriereZahl*(5) oder *betankeFahrzeug*(50) u.s.w.

Als Erstes wollen wir unsere Klasse Fahrzeuge derart erweitern, dass sie selbst die Werte der (inneren) Attribute ausgeben kann.




```
public class Fahrzeug {
    double verbrauch;
    double fuellStand;
    int achsenAnzahl=2;
    String farbe;

    void printOut(){
        System.out.println("Fahrzeug");
        System.out.println("--------");
        System.out.println("verbrauch="+verbrauch);
        System.out.println("fuellStand="+fuellStand);
        System.out.println("achsenAnzahl="+achsenAnzahl);
        System.out.println("farbe="+farbe);
    }
}
```

Abbildung 10: Methode printOut()

Die Ausgabe kann so aussehen:

```
Fahrzeug
--------
verbrauch=8.0
fuellStand=0.0
achsenAnzahl=5
farbe=rot
```

Listing 7: Ausgabe von printOut()

Die Benutzung der Methode kann man sich so vorstellen...

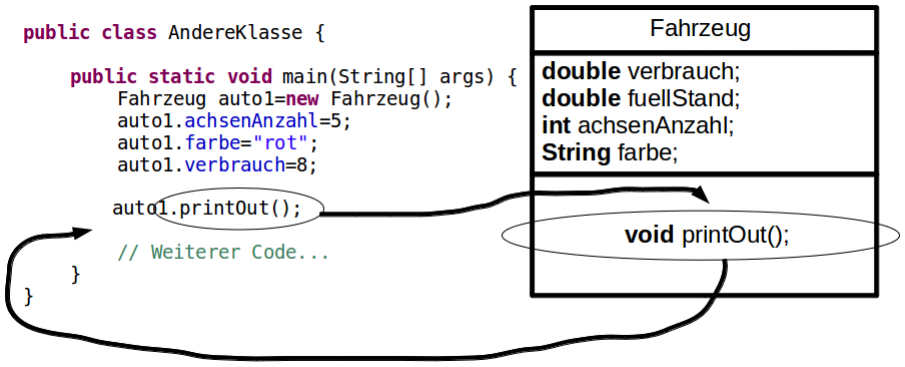


Abbildung 11: Benutzung der Methode printOut()

oder auch so:

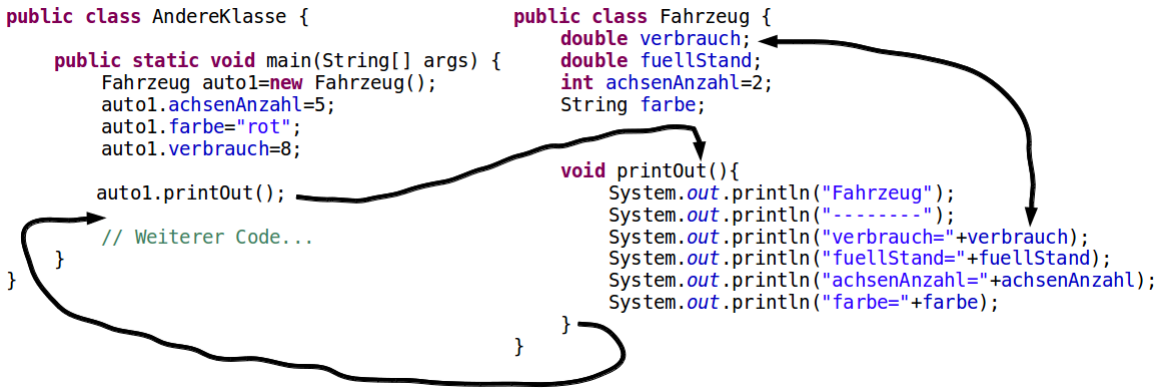


Abbildung 12: Benutzung der Methode printOut()

Wir wollen versuchen unsere Klasse Fahrzeug mit weiteren sinnvollen Methoden zu bereichern.

7.2.1. Parameter

Wikipedia.Parameter sind Werte, die vom Aufrufenden an eine Methode übergeben werden, damit diese in der (entfernten) Methode verwendet werden können. Die Parameter werden dabei durch Kommatas voneinander getrennt. Das Aufrufen einer Methode wird auch als "Meldungübergabe (message passing)" bezeichnet. Sehen Sie sich das näher am Beispiel an....

Als nächstes soll ein Fahrzeug betankt werden können. Dabei soll es möglich sein eine bestimmte Menge zu tanken. Bei diesem Betankungsvorgang soll das Attribut *fuellStand* entsprechend verändert werden. Wir schreiben also eine zusätzliche Methode.

```
void betanken(int menge){
  fuellStand=fuellStand+menge;
}
```

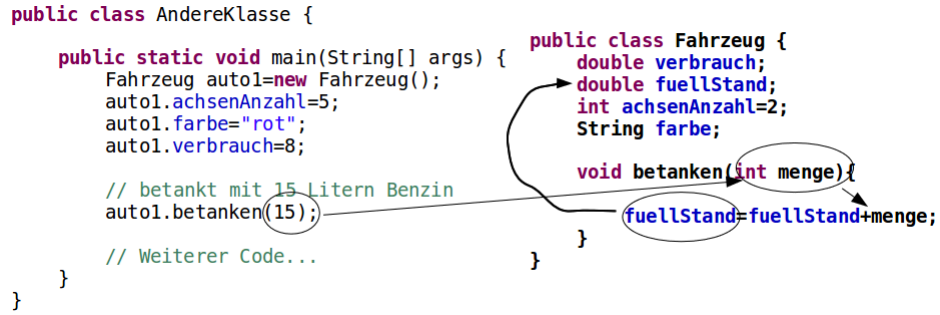


Abbildung 13: Methoden Aufruf 2

Wichtig dabei ist, dass der Wert 15 in `auto1.betanken(15);` übergeht in die (lokale) Variable *menge*. Die Variable *menge* kann innerhalb der Methode `betanken(..)` wie jede andere Variable benutzt werden und wird bei beenden der Methode `betanken(..)` automatisch aus dem Speicher entfernt. Umgangssprachlich "stirbt" die Variable mit dem beenden von `betanken(..)`.

Nun ist es also möglich ein Fahrzeug mit einer beliebigen Menge Benzin zu betanken.

7.2.2. Rückgabewert

Nach einem Betankungsvorgang wäre aber gut zu wissen, wie viel Benzin denn nun im Tank vorhanden ist. Wir erinnern uns, dass es jederzeit möglich ist den Zustand einer Variablen mit

`int neuerStand=auto1.fuellStand` zu erfragen. Viel einfacher aber wäre es, wenn die Methode `betanken(..)` das Resultat ihrer Berechnung gleich zurückgeben würde. Dazu muss die Methode lediglich geringfügig geändert werden.

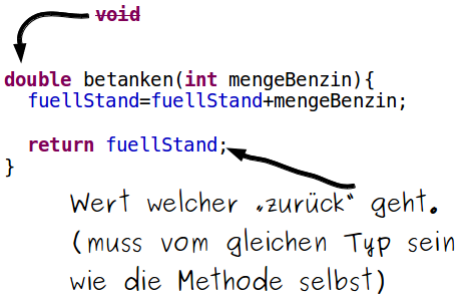


Abbildung 14: Methoden Aufruf 2

Die neue Funktionalität muss gleich mal getestet werden!

```
public class AndereKlasse {
  public static void main(String[] args) {
    Fahrzeug auto1=new Fahrzeug();
    auto1.achsenAnzahl=5;
    auto1.farbe="rot";
    auto1.verbrauch=8;

    auto1.printOut();
    // betankt mit 15 Litern Benzin
    double neuerStand=auto1.betanken(15); // <- Betanken und
                                          // aktualisierter Tankwert
                                          // beziehen
    System.out.println(neuerStand);
    auto1.printOut();
  }
}
```

7.2.3. Spezielle Methoden

Wikipedia. In der objektorientierten Programmierung ist es unüblich direkt auf die Attribute einer Klasse zu zu greiffen, da damit gegen Prinzipien der Objektorientierung (und "gutem" Programierstil) verstossen wird. Die Gründe werden wir im Teil Objektorientierung genauer erörtern.

7.2.3.1. Getter Getter-Methoden werden auch als Accessor-Methoden bezeichnet. Diese Getter "geben" Attribute zurück. Sind also ganz gewöhnliche Methoden, welche durch die spezielle Namensgebung zu Getter-Methoden werden.

```
public class Fahrzeug {

  String farbe; // <-- Attribut

  public String getFarbe() { // <-- Getter-Methode
    return farbe;
  }
}
```

7.2.3.2. Setter Setter-Methoden werden auch als Mutator-Methoden bezeichnet. Diese Setter "setzten" Attribute.

```
public class Fahrzeug {

  String farbe; // <-- Attribut
```

```
  public void setFarbe(String farbe) { // <-- Setter-Methode
    this.farbe = farbe;
  }
}
```

Setter- und Getter-Methoden werden also zum kapseln der Attribute verwendet. Dadurch kann:

1. Der direkte Zugriff auf Attribute von "aussen" wirksam verhindert werden.
2. Eine Prüfung des zu setzenden Wertes durchgeführt werden.
3. Das Attribut (zeitlich vor dem Zugriff) aktualisiert werden.

7.2.3.3. Konstruktoren

Frei nach java beginners

Konstruktoren sind spezielle methodenähnliche Klassenstrukturen, die den Namen ihrer Klasse tragen und beim Erzeugen von Objekten der Klasse über das Schlüsselwort *new* aufgerufen werden. Eine Klasse kann keinen, einen oder mehrere unterschiedliche Konstruktoren besitzen. Sie dienen dazu, ein neu gebildetes Objekt einer Klasse in einen definierten Anfangszustand zu versetzen. Welcher dies ist hängt davon ab, welcher Konstruktor bei der Objektbildung aufgerufen wird. Ein leerer (Standard-) Konstruktor muss nicht angegeben werden, er wird bei Fehlen von der JVM automatisch erzeugt (daher sind wir bisher noch keinem begegnet :-)).

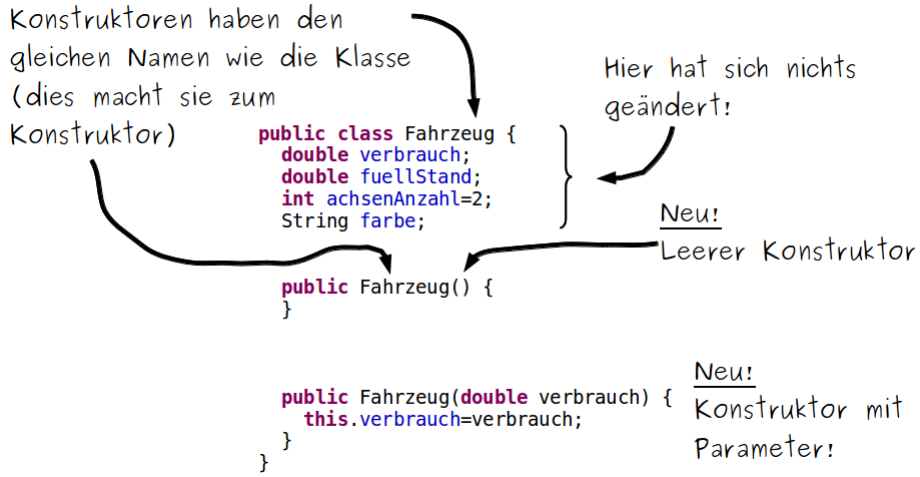


Abbildung 15: Konstruktoren

Das Beispiel zeigt eine Klasse mit zwei Konstruktoren. Der erste ist ein Standardkonstruktor, der zweite dient zur Instanzierung eines *double* Attributes. Beim Aufruf dieses Konstruktors wird somit ein *double*-Attribut bereits beim Erzeugen einer Instanz initialisiert. Soll also ein Objekt der Klasse *Fahrzeug* erzeugt werden, kann dies ohne oder mit Angabe eines Anfangswertes für die Instanzvariable *verbrauch* geschehen:

Konstruktoren müssen einigen Regeln folgen:

- Sie tragen immer den Namen der Klasse
- Sie besitzen keine Modifizierer (static, final...)
- Sie können überladen werden (mehrere Konstruktoren)
- Sie besitzen keinen Rückgabewert
- Rufen sie ihrerseits keinen anderen Konstruktor auf, werden sie vom Compiler um den Aufruf des Konstruktors der Basisklasse super() erweitert

7.2.4. Aufgaben zu den Methoden:

Erstellen Sie eine neue Klasse für diese Übungen:

7.2.4.1. Quadrieren einer Zahl zum Ersten Es soll eine Methode geschrieben werden welche eine *int* Zahl entgegennimmt, diese Quadriert und danach ausgibt.

1. Finde einen geeigneten Namen für die Klasse
2. Finde einen geeigneten Namen für die Methode
3. Schreibe die Methode

7.2.4.2. Quadrieren einer Zahl zum Zweiten Erweitere die Methode so, dass diese das Resultat an den Aufrufer zurück gibt.

7.2.4.3. Potenzieren Es soll eine Methode geschrieben werden welche die Potenz errechnet. Dazu muss die Methode 2 Stück *int* Parameter entgegennehmen (Basis und Exponent) und daraus die Potenz errechnen. (Informieren Sie sich zuerst über die Mathematischen Spezialfälle)

7.2.4.4. Parallelschaltung zweier Wiederstände Erstellen Sie eine Methode welche zwei *double* Werte entgegen nimmt und daraus den gesamt Widerstand berechnet.

$$R_{gesamt} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}} = \frac{R_1 \cdot R_2}{R_1 + R_2}$$

1. Die Unerschrockenen unter uns versuchen 3 oder 4 Wiederstände in Parallelschaltung zu berechnen.
2. Die echt Unerschrockenen erstellen eine Methode, welche mit x Parallelen Wiederständen rechnen kann.
3. Sind Sie ein echt Unerschrockener und langweilen sich? Googeln sie nach “java ellipse”.

7.2.5. Die Übergabe Parameter

Bei den meisten Programmiersprachen werden die Parameter an Methoden als Kopie der ursprünglichen Variablen übergeben.

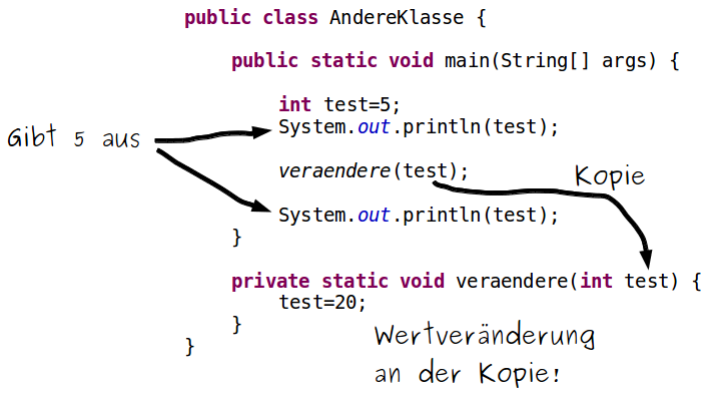


Abbildung 16: Wertübergabe als Kopie

Bei diesem Beispiel ist gut zu erkennen wie die Veränderung an der Kopie erfolgt.
Gänzlich anders scheint sich folgender Code zu verhallten:

```
public class AndereKlasse {

    public static void main(String[] args) {

        Fahrzeug f=new Fahrzeug();

        f.printOut();

        veraendere(f);

        f.printOut();

    }

    private static void veraendere(Fahrzeug x) {
        x.verbrauch=20;
    }
}
```

Kopie (der Referenz)

Das Objekt wird verändert!

Abbildung 17: Wertveränderung am Objekt

Dieses Verhallten wird als “pass-by-reference” bezeichnet, jedoch wird von Entwickler welche andere Programmiersprachen kennen das Wort Referenz oft anders als es Java benutzt interpretiert! Siehe stackoverflow. Ich selbst verzichte an dieser Stelle auf eine Erklärung, ergänze da für mit einer Zeichnung.

```
public class AndereKlasse {

    public static void main(String[] args) {

        Fahrzeug f=new Fahrzeug();

        f.printOut();

        veraendere(f);

        f.printOut();

    }

    private static void veraendere(Fahrzeug x) {
        x.verbrauch=20;
    }
}
```

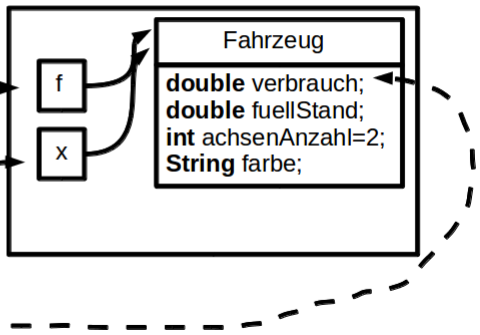


Abbildung 18: Wertveränderung am Objekt

7.3. Zugriffsmodifizierer

oracle und highscore.

Sehr viele objektorientierte Sprachen unterstützen Zugriffsattribute. Mit ihnen kann jeweils festgelegt werden, inwiefern eine andere Klasse auf die eigene Klasse, auf eine Eigenschaft oder Methode zugreifen darf - entweder sie darf oder eben nicht.

Je nachdem, wer zugreift, und je nachdem, welches Zugriffsattribut gilt, ist der Zugriff gestattet oder nicht. Folgende Matrix gibt detailliert Auskunft, wer wann Erlaubnis hat zuzugreifen.

Modifizierer	Die Klasse selbst	Klasse im selben Package	Unterklasse	Rest der Welt
public	erlaubt	erlaubt	erlaubt	erlaubt
protected	erlaubt	erlaubt	erlaubt	X
no modifier	erlaubt	erlaubt	X	X
private	erlaubt	X	X	X

Tabelle 10: Zugriffsmodifizierer

Die Matrix kann ausgehend von Zugriffsattributen oder von Klassen gelesen werden. Beginnen wir mit den Zugriffsattributen.

Das Zugriffsattribut *private* erlaubt ausschliesslich der eigenen Klasse, auf die entsprechende Eigenschaft oder Methode zuzugreifen. Es handelt sich hierbei also um den stärksten Schutzmechanismus. Andere Klassen - egal, ob Elternklasse, aus dem eigenen Paket oder eine wildfremde Klasse - können nicht auf mit *private* geschützte Merkmale zugreifen. Sie wissen gar nicht, dass es derartige Merkmale in der Klasse gibt.

Mit *package* geschützte Merkmale können sowohl von der eigenen Klasse als auch von Klassen aus dem eigenen Paket verwendet werden. Eine Elternklasse kann normalerweise nicht auf derartige Merkmale zugreifen - ausser sie gehört dem Paket an und ist somit auch eine Paket-Klasse.

Merkmale, die mit *protected* geschützt sind, können von der eigenen Klasse und von Klassen aus dem Paket verwendet werden. Zusätzlich gilt, dass Elternklassen auf vererbte *protected*-Merkmale in Kindklassen zugreifen können. Es ist hierbei nicht notwendig, dass die Kindklasse im gleichen Paket liegt.

Das Zugriffsattribut *public* bedeutet, dass jede beliebige Klasse auf das Merkmal zugreifen darf - von eigener Klasse über Eltern- und Paket-Klassen bis hin zu wildfremden Klassen darf das Merkmal jeder verwenden.

Soweit die Bedeutung der Matrix aus Sicht der Zugriffsattribute. Wie sieht es mit den Klassen aus?

Eine Klasse kann grundsätzlich auf jedes beliebige Merkmal der eigenen Klasse zugreifen - völlig egal, ob das Zugriffsattribut private, package, protected oder public gesetzt ist.

Eine Paket-Klasse kann auf alle Merkmale einer anderen Klasse im Paket zugreifen, soweit diese nicht mit dem Zugriffsattribut *private* geschützt sind.

Elternklassen haben Zugriff auf mit *protected* und mit *public* geschützte Merkmale ihrer Kindklassen. Auf Merkmale, die mit *private* und *package* geschützt sind, kann hingegen von der Elternklasse aus nicht zugegriffen werden (ausser natürlich die Elternklasse liegt im gleichen Paket wie die Kindklasse und das Zugriffsattribut *package* wird verwendet).

Fremde Klassen, die weder Elternklasse einer anderen Klasse sind noch im gleichen Paket liegen, haben nur Zugriff auf Merkmale, die mit *public* deklariert sind.

Zugriffe sollten so restriktiv wie nur möglich eingestellt werden. Damit werden Eigenschaften und Methoden einer Klasse vor versehentlicher Änderung bzw. vor versehentlichem Aufruf durch andere Klassen geschützt, und es wird Fehlern vorgebeugt.

7.4. Klassen versus Arrays

Sowohl Klassen als auch Arrays bestehen aus mehreren Elementen. Was ist der Unterschied zwischen beiden und wann soll welches Konstrukt angewendet werden?

Array	Klasse
Besteht aus mehreren gleichartigen Elementen, z.B. aus lauter *int* Werten	Können aus mehreren verschiedenartigen Attributen bestehen, z.B. aus einem *int* und einem *boolean*-Wert.
Die Elemente eines Arrays haben keinen Namen und werden stattdessen über einen Index angesprochen, z.B. wochenTage[2]	Die Attribute einer Klasse haben einen Namen und werden über diesen Namen angesprochen, z.B. *fahrzeug.farbe*.
Die Anzahl der Elemente wird bei der Erzeugung des Arrays festgelegt, z.B. *double* einArray=*new double*[5];	Die Anzahl der Felder wird bei der Deklaration (während des Programmierens) der Klasse festgelegt.

Tabelle 11: Klassen versus Arrays

Ein Array ist im wesentlichen eine Liste von immer gleichen Mustern. Eine Liste mit Kuchenrezepten oder eine Liste mit Zahlen oder eine Liste mit ...

Eine Klasse ist das Rezept nach welchen es möglich ist (viele) Fahrzeuge zu backen ;-) Diese Fahrzeuge können in eine Liste (Array) gebracht werden um diese Fahrzeuge einfach und schnell bearbeiten zu können.

Man sollte also Arrays verwenden, wenn man es mit lauter gleichartigen Elementen zu tun hat, die keinen eigenen Namen haben. Hingegen sollte man Klassen verwenden, wenn man verschiedenartige Elemente hat und diese über einen Namen ansprechen will.

Teil II.
Objektorientierte Programmierung

[10]In diesem Kapitel wollen wir die wichtigsten Konzepte und ein allgemeines Verständnis für die objektorientierte Programmierung schaffen. Auf diese werden wir eingehen und sie in kleinen Testprogrammen praktisch anwenden.

Mitte der 60er-Jahre des letzten Jahrhunderts kam es zu einer Softwarekrise.

Experten diskutierten die Ursachen der Krise und die Gründe für die gestiegene Fehlerrate. Ein Teil der Softwareexperten kam zu dem Schluss, dass die Softwarekrise nicht mit den herkömmlichen Programmiersprachen zu bewältigen sei. Sie begannen deshalb, eine Generation von neuen Programmiersprachen zu entwickeln. Die Entwickler dieser Sprachen kritisierten an den herkömmlichen Programmiersprachen vor allem, dass sich die natürliche Welt bisher nur unzureichend abbilden lasse. Um dem zu entgehen, gingen sie von natürlichen Begriffen aus und wandelten sie für die Programmierung ab. Da sich alles um den Begriff des Objekts dreht, nannten sie die neue Generation von sprachen 'objektorientiert'.

8. Grundlegende Konzepte

Wikipedia. Grundidee ist, die reale Welt in Computer nachbilden zu können. Dazu muss die reale Welt vereinfacht und verallgemeinert werden. Eine Vereinfachung ist, alles auf Objekte zu abstrahieren. Ein Objekt ist dabei eine genaue Abbildung eines Gegenstandes aus unserer Umgebung. Ein Objekt hat feste Attribute (Eigenschaften, wie Farbe oder Temperatur) und bestimmte Fähigkeiten (Methoden), mit welchen dieses Objekt mit seiner Umwelt interagieren kann. Eine Klasse ist eine Verallgemeinerung eines Objektes und legt fest, welche Attribute und Methoden ein Objekt haben kann ohne diese konkret zu bestimmen.

Ein Kuchenrezept(Klasse) beschreibt abstrakt welche Zutaten(Attribute, Variablen) und welchen Tätigkeiten(Methoden) zu einem konkreten Kuchen(Objekt, Instanz) führen. Wenn nun mehrere Kuchen gebacken werden, kann immer das gleiche Rezept benutzt werden, es wird aber nie exakt zwei gleiche Kuchen geben, auch wenn noch so genau nach dem Rezept vorgegangen wird!

Verallgemeinert hat der Mensch zwei Arme, zwei Beine, n Nasenhaare, seine Augen haben eine Farbe, er hat eine Grösse und ein Gewicht(Attribute). Konkret kann jeder Mensch verschieden schnell rennen, mit seinen Händen winken oder nach Dingen greifen(Methoden). Die Abstraktion geht soweit, dass eine Klasse Attribute anderer Klassen besitzen kann oder gar von sich selbst. Zum Beispiel könnte ein Mensch die Klasse Blutgruppe haben oder ein Attribut auf weitere Menschen, wie auf die Mutter oder den Vater oder ein Array, welches die Kinder enthält.

8.0.1. Grundbegriffe

Die Grundbegriffe der objektorientierten Programmierung sind folgendermassen zusammengefasst:

- Alles ist ein Objekt.
- Objekte kommunizieren durch Nachrichtenaustausch.
- Objekte haben ihren eigenen Speicher.
- Jedes Objekt ist ein Exemplar einer Klasse.
- Die Klasse modelliert das gemeinsame Verhalten ihrer Objekte.
- Ein Programm wird ausgeführt, indem dem ersten Objekt die Kontrolle übergeben und der Rest als dessen Nachricht behandelt wird.

[10]Es wird versucht die UML Notation zu benutzen.Wikipedia

8.0.2. Prinzipien

Neben diesen Grundbegriffen sind folgende Prinzipien wichtig:

- Abstraktion
- Vererbung
- Kapselung
- Beziehungen
- Persistenz
- Polymorphie

8.1. Abstraktion

Da eine der Grundideen ist, die reale Welt in Computern nach zu bilden und die Objektorientierte Programmierung dies auch fördert, wird häufig eine übertriebene Konstruktion von der natürliche Welt erstellt. Die Kunst besteht darin die Wirklichkeit so genau wie nötig und so einfach wie möglich abzubilden. Dieses Ziel ist nicht immer einfach zu verfolgen und die Analyse der für das Programm wesentlichen und richtigen Bestandteile bereitet unter Umständen Probleme.

Um Kompaktheit zu erreichen, ist es notwendig, die meist extrem komplizierten natürlichen Objekte und deren Beziehungen soweit es geht zu abstrahieren, also zu vereinfachen. Der Fachbegriff für diese Technik nennt sich demzufolge auch Abstraktion.

8.2. Vererbung

Wikipedia. Die Vererbung (englisch inheritance) ist eines der grundlegenden Konzepte der Objektorientierung und hat grosse Bedeutung in der Softwareentwicklung. Die Vererbung dient dazu, aufbauend auf existierenden Klassen neue zu schaffen, wobei die Beziehung zwischen ursprünglicher und neuer Klasse dauerhaft ist. Eine neue Klasse kann dabei eine Erweiterung oder eine Einschränkung der ursprünglichen Klasse sein. Neben diesem konstruktiven Aspekt dient Vererbung auch der Dokumentation von Ähnlichkeiten zwischen Klassen, was insbesondere in den frühen Phasen des Softwareentwurfs von Bedeutung ist. Auf der Vererbung basierende Klassenhierarchien spiegeln strukturelle und verhaltensbezogene Ähnlichkeiten der Klassen wider.

Die vererbende Klasse wird meist Basisklasse (auch Super-, Ober- oder Elternklasse) genannt, die erbende abgeleitete Klasse (auch Sub-, Unter- oder Kindklasse). Den Vorgang des Erbens nennt man meist Ableitung oder Spezialisierung, die Umkehrung hiervon Generalisierung, was ein vorwiegend auf die Modellebene beschränkter Begriff ist. In der Unified Modeling Language (UML) wird eine Vererbungsbeziehung durch einen Pfeil mit einer dreieckigen Spitze dargestellt, der von der abgeleiteten Klasse zur Basisklasse zeigt. Geerbte Attribute und Methoden werden in der Darstellung der abgeleiteten Klasse nicht wiederholt.

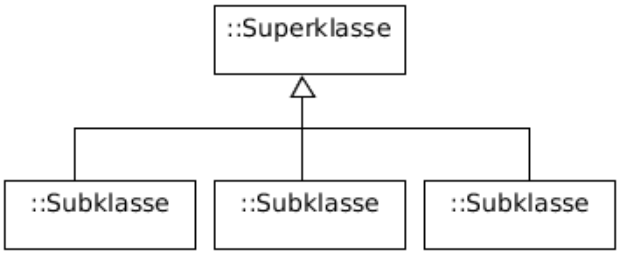


Abbildung 19: Vererbung spezifischer Eigenschaften auf die Subklasse

Nun wollen wir uns ansehen, wie diese Technologie genutzt werden kann. Eine Ableitung von einer Klasse zur anderen zieht zwangsläufig eine Klassenhierarchie nach sich, in der alle Klassen und ihre Beziehungen untereinander dargestellt werden können.

Um nun eine Ableitung vorzunehmen, nutzt man das *extends* - Schlüsselwort im Kopf der Klassendefinition. Exakt diesem schliesst sich der Name der Superklasse kann. Dazu folgender syntaktischer Bauplan.

```
public class Superklasse {
}

public class Subklasse extends Superklasse{
}
```

Auch wenn unsere Klassen hier noch keine Elemente haben, so würde die Subklasse automatisch alle Variablen und Methoden der Superklasse erben.

Pferd und Zebra sind eng verwandt, in mancherlei Hinsicht aber doch sehr verschieden. Diese Unterschiede sind von anderer Qualität als die Unterschiede zwischen zwei Pferden: Als Beispiel soll Eigenschaft *Farbe* von Pferden dienen welche wohl eher schwierig für ein Zebra zu definieren ist, aus diesem Grund wurde für das Zebra anstelle der *Farbe* die Eigenschaft *Muster* gewählt.

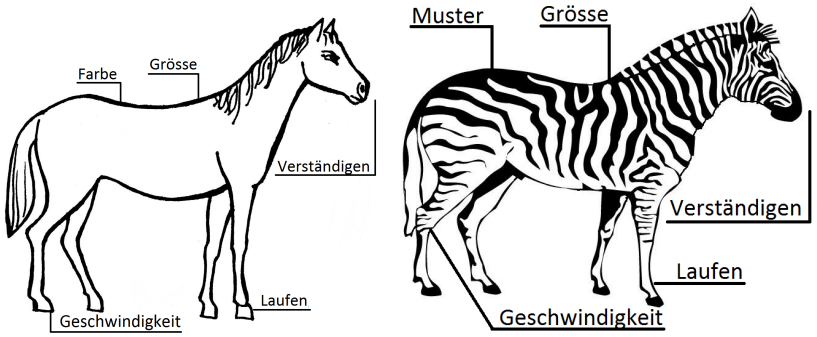


Abbildung 20: Objekte verschiedener Klassen unterscheiden sich in Ihrer Form

Es ist also in den Fällen, in denen es auf die Unterschiede zwischen Farbe und Muster ankommt, immer besser, einem Zebra die Eigenschaft Muster zu geben und es einer anderen Klasse zuzuordnen. In allen anderen Fällen genügen die gemeinsamen Attribute.

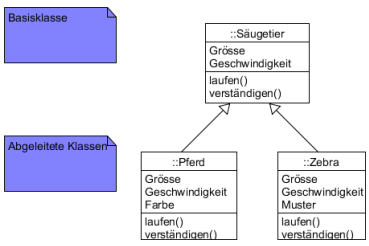


Abbildung 21: Die Basisklasse überträgt Basiseigenschaften und verhalten.

8.2.1. Abgeleitete Klassen

Angenommen, Sie möchten gern eine neue Klasse namens *Muli* auf Basis der Klasse *Säugetier* erzeugen. In der objektorientierten Programmierung spricht man davon, von *Säugetier* eine neue Klasse namens *Muli* abzuleiten. Die neue Klasse *Muli* erbt wie schon zuvor *Pferd* und *Zebra* das Verhalten und die Attribute Grösse und Höchstgeschwindigkeit der Basisklasse *Säugetier*. Sie stammt von *Säugetier* ab.

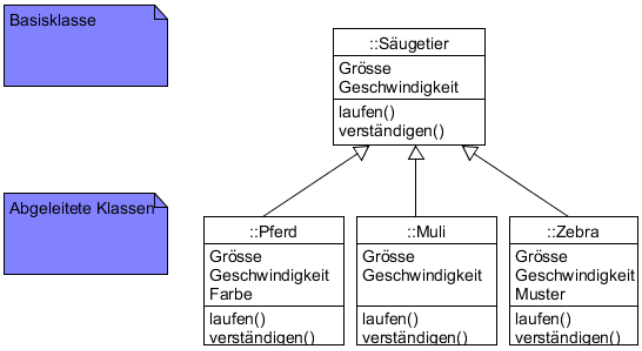


Abbildung 22: Die neue Klasse *Muli* ist eine von *Säugetier* abgeleitete Klasse

Bei der Ableitung einer Klasse gibt es Bestandteile, die der Subklasse nicht mit vererbt werden.

- Konstruktoren
- Destruktoren
- statische Datenelemente
- statische Methoden
- private Elemente

Diese Elemente müssen für jede Subklasse vom Programmierer neu definiert werden, da ihre Funktionalität für die Subklasse nicht mehr zutreffend ist. Das hängt damit zusammen, dass einmal jeder Konstruktor seine Teilklasse individuell aufbaut und dass in den Subklassen meist neue Elemente hinzukommen, die der neue Konstruktor ebenfalls behandeln muss.

Auch statische Elemente werden nie vererbt, da sie ja global an eine Klasse und nicht an ein konkretes Objekt gebunden sind. Die Einschränkungen bei privaten Elementen werden noch genauer behandelt.

8.2.2. Mehrfachvererbung

In der Natur ist sie üblich, in den Programmiersprachen Java und Smalltalk jedoch nicht erlaubt: die Mehrfachvererbung. Sie wäre dann praktisch, wenn Sie zwei Klassen verschmelzen wollten, zum Beispiel die Klasse *Pferd* mit der Klasse *Esel*. Die neue Kreuzung *Muli* würde Attribute und Verhalten beider Basisklassen erben. Aber welche Attribute und welches Verhalten? Sollen sich Mulis verständigen und laufen wie Pferde oder wie Esel? Bei derartigen Szenarien kommt die Softwareentwicklung an die Grenze des technisch Sinnvollen. Es ist nicht sinnvoll, Erbinformationen nach dem Zufallsprinzip zu übertragen, um die Natur zu imitieren. Der Anwender wünscht sich im Regelfall Programme, die über definierte Eigenschaften verfügen und deren Verhalten vorhersehbar ist.

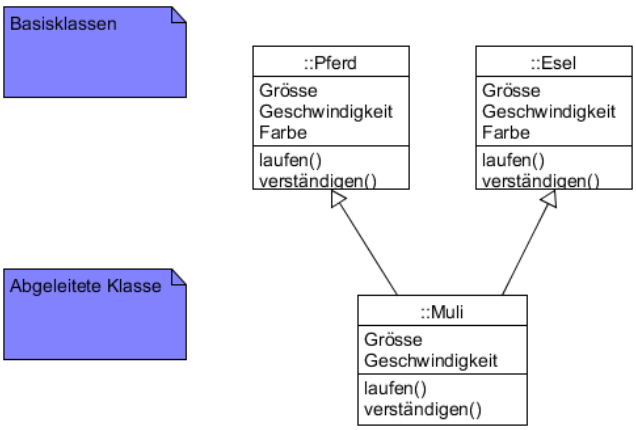


Abbildung 23: Mehrfachvererbung am Beispiel einer Kreuzung

8.3. Kapselung

Eines der wichtigsten Merkmale objektorientierter Sprachen ist der Schutz von Klassen und Attributen vor unerwünschtem Zugriff. Jedes Objekt besitzt eine Kapsel, die die Daten und Methoden des Objekts schützt. Die Kapsel versteckt die Teile des Objekts, die von aussen nicht oder nur durch bestimmte andere Objekte erreichbar sein sollen. Die stellen, an denen die Kapsel durchlässig ist, nennt man Schnittstellen.

Die wichtigste Schnittstelle der Klasse Pferd ist sein Konstruktor. Über diese spezielle Methode lässt sich ein Objekt der Klasse Pferd erzeugen. Ein anderes Beispiel für eine solche Schnittstelle ist die Methode Laufen der Klasse Pferd. Das Objekt *FauleSocke* besitzt eine solche Methode *Laufen*, und *Adam*, ein Objekt der Klasse *Mensch*, kann diese Methode verwenden. Er kommuniziert mit *FauleSocke* über diese Schnittstelle und teilt darüber *FauleSocke* mit dass er laufen soll. Das Objekt *Adam* darf nicht alle Daten von *FauleSocke* verändern. Zum Beispiel soll es ihm selbstverständlich nicht erlaubt sein, die Grösse des Pferds zu Ändern. Gäbe es eine öffentlich zugängliche Methode wie zum Beispiel *Wachsen*, so könnte er *FauleSocke* damit verändern.

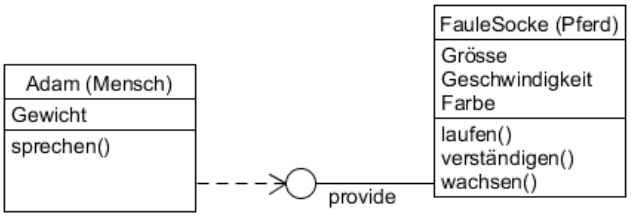


Abbildung 24: Objekte kommunizieren nur über Schnittstellen

8.4. Beziehungen

Klassen und deren Objekte unterhalten in einem Programm die unterschiedlichsten Beziehungen untereinander. In den vorangegangenen Abschnitten haben Sie bereits mehrere Formen der Bezie-

hungen kennengelernt: den Aufruf von Methoden und Vererbungen. An dieser Stelle möchte ich Ihren Blick für die zwei grundlegend verschiedenen Arten von Beziehungen zwischen Klassen und Objekten schärfen:

- Beziehungen, die nicht auf Vererbung beruhen
- Vererbungsbeziehungen

8.4.1. Beziehungen, die nicht auf Vererbung beruhen

Man unterscheidet bei dieser Form von Beziehungen drei verschiedene Unterarten:

- Assoziationen (Verknüpfungen)
- Aggregationen (Zusammenlagerungen)
- Kompositionen(Zusammensetzungen)

8.4.1.1. Assoziation Assoziation ist die einfachste Form einer Beziehung zwischen Klassen und Objekten. Die Abhängigkeiten sind bei dieser Beziehungsart im Vergleich zur Vererbung gering. Man sagt auch, die Objekte sind lose gekoppelt. Eine Assoziation besteht zum Beispiel, wenn ein Reiter-Objekt namens *Adam* einem Pferde-Objekt namens *FauleSocke* die Botschaft laufen() sendet. Die beiden Objekte *Adam* und *FauleSocke* existieren getrennt und erben nichts voneinander.

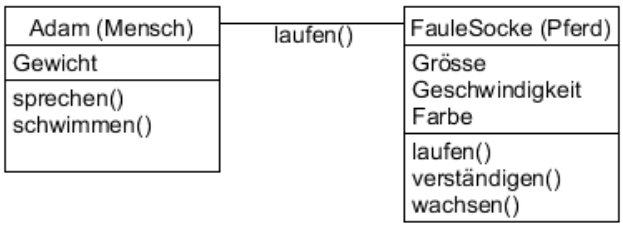


Abbildung 25: Eine einfache Assoziation zwischen Mensch und Pferd

8.4.1.2. Aggregation Eine Steigerung der Assoziation ist die Aggregation. Eine solche Beziehung besteht dann, wenn ein Objekt aus anderen Objekten besteht. Zum Beispiel soll Pferdefutter aus einer nicht näher bestimmten Anzahl von Karotten bestehen. Das bedeutet zum Beispiel, dass Pferdefutter eine 'Besteht-aus-Beziehung' zur Karotte unterhält.

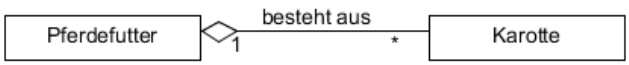


Abbildung 26: Aggregration zwischen Pferdefutter und Karotten.

Diese Beziehung ist aber von einer völlig anderen Qualität als im vorangegangenen Beispiel zwischen einem Menschen und einem Pferd. Während Pferd und Menschen allein und unabhängig

voneinander existieren können, setzt sich das Pferdefutter (unter anderem) aus Karotten zusammen. Wichtig ist hierbei wieder, dass beide Objekte nichts voneinander erben und jedes Karotten-Objekt auch allein lebensfähig ist, was dieses Beispiel von der strengeren Komposition unterscheidet.

8.4.1.3. Komposition Die stärkste Form der Beziehungen, die nicht auf Vererbung beruhen, stellt die Komposition dar. Wie bei der Aggregation liegt wieder eine 'Besteht-aus-Beziehung' vor, sie ist aber im Gegensatz zur Aggregation abermals verschärft. Die Abhängigkeiten sind nochmals stärker. Ein Beispiel für eine Komposition ist das Verhältnis zwischen einem Pferd und seinen vier Beinen. Hier besteht eine sehr enge Beziehung, denn ein Bein ist - im Gegensatz zur Karotte - als selbstständiges Objekt vollkommen sinnlos. Bei der Erzeugung eines Pferde-Objekts bekommt dieses automatisch vier Beine, die im Zusammenhang mit anderen Klassen nicht verwendet werden können.

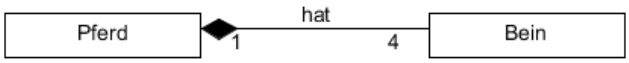


Abbildung 27: Ein Pferd und seine vier Beine als Komposition

Pferdebeine sind also ohne ein geeignetes Objekt der Klasse Pferd nicht lebensfähig. wenn ein Pferde-Objekt stirbt, so sterben auch seine Pferdebeine.

8.4.2. Vererbungsbeziehungen

Vererbungsbeziehungen nennen sich auch Generalisierung (Verallgemeinerung) oder Spezialisierung (Verfeinerung). Dies sind nicht etwa Unterarten der Vererbung, sondern alternative Begriffe für Vererbungsbeziehungen. Welchen der zwei alternativen Begriffe man verwenden möchte, hängt vom Blickwinkel ab, aus dem man die Vererbungsbeziehung betrachtet.

8.4.2.1. Generalisierung Wenn Sie die Basisklasse aus dem Blickwinkel der abgeleiteten Klasse betrachten wollen, ist Generalisierung der passende Begriff dazu. Zum Beispiel ist die Klasse *Säugetier* eine Generalisierung der Klassen *Pferd* oder *Zebra*. Mit anderen Worten: Die Klasse *Säugetier* ist der allgemeine Begriff (Oberbegriff) für die Klassen *Pferd* und *Zebra* (Klassifizierung).

8.4.2.2. Spezialisierung Wenn Sie die abgeleitete Klasse aus dem Blickwinkel der Basisklasse betrachten wollen, ist Spezialisierung der passende Begriff dazu. Zum Beispiel sind die Klassen *Pferd* oder *Zebra* eine Spezialisierung der Klasse *Säugetier*. Mit anderen Worten: Die Klassen *Pferd* und *Zebra* stellen eine Verfeinerung der Klasse *Säugetier* dar.

8.4.2.3. Probleme mit der Vererbung Vererbungsbeziehungen stellen eine sehr starke Kopplung zwischen Klassen und damit auch zwischen Objekten her. Eine solch starke Kopplung hat nicht nur Vorteile, sondern auch gravierende Nachteile, wie das folgende Beispiel zeigt: Eine Klasse namens *Wal* soll aus der Klasse *Säugetier* erzeugt werden. Die neue Klasse erbt wie die Klassen *Pferd* und *Zebra* die Attribute Grösse und Geschwindigkeit sowie die Methoden Laufen und Verständigen - Moment mal: Laufen? Fast alle Säugetiere können laufen, Wale jedoch nicht.

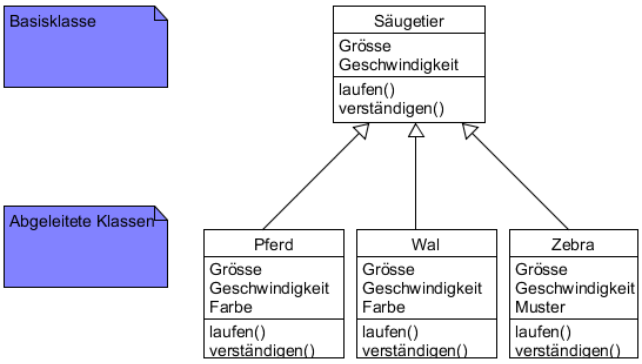


Abbildung 28: Durch Vererbung vererben sich auch Designfehler.

Hier ist genau das passiert, was tagtäglich zu den Problemen der objektorientierten Programmierung gehört: Die Funktionalität der Basisklasse ist nicht ausreichend analysiert worden. Vereinfacht gesagt: Hier liegt ein Designfehler vor, den man dadurch beheben muss, dass man die Methode *Laufen* durch die Methode *Fortbewegen* ersetzt.

8.5. Designfehler

Sie können sich vielleicht vorstellen, dass es sehr unangenehm ist, wenn die Basisklasse aufgrund eines Designfehlers geändert werden muss. Durch die starke Beziehung zwischen Basisklasse und abgeleiteter Klasse pflanzen sich etwaige Änderungen lawinenartig in alle Programmteile fort, in denen Objekte des Typs *Pferd* und *Zebra* mit der Methode *Laufen* verwendet wurden. An allen Stellen des Programms, wo die Methode *Laufen* der Klasse *Säugetier* verwendet wurde, muss sie durch die Methode *Fortbewegen* ersetzt werden. Im Fall von Designfehlern stellt sich die Technik der Vererbung als grosser Nachteil heraus. Vererbung hat neben diesem Manko auch den Nachteil, dass sich nicht nur Designfehler, sondern alle anderen vorzüglich gestalteten, aber unerwünschten Teile der Basisklasse in die abgeleiteten Klassen in Form von Ballast übertragen: Die Nachkommen solcher übergewichtiger Klassen werden immer fetter und fetter. Daher sollten Sie Vererbung stets kritisch betrachten, sparsam einsetzen und wirklich nur dort verwenden, wo sie sinnvoll ist.

8.6. Umstrukturierung

Aber zurück zu den Designfehlern. Wie geht man mit Fehlern dieser Art um? Sie sind trotz der Vererbung heute kein so grosses Problem mehr wie noch vor ein paar Jahren. Es gibt mittlerweile moderne Softwareentwicklungswerkzeuge, mit denen es relativ leicht ist, die notwendige Umstrukturierung (Refactoring) vorzunehmen. Allerdings sollten Sie Software möglichst nur während der Analyse- und Designphase der Software umstrukturieren. Als Regel gilt: Je später Änderungen vorgenommen werden, desto höher ist der damit verbundene Aufwand.

8.7. Modellierung

Um solche Designfehler und damit kostspielige Umstrukturierungen zu vermeiden, ist es bei grösseren Projekten sinnvoll, ein Modell der Software zu entwerfen. Genauso wie man im Automobilbau vor jedem neu zu konstruierenden Automobil ein Modell entwickelt, ist es auch in der Softwareentwicklung sinnvoll, ein Modell zu konstruieren, bevor man mit der eigentlichen Umsetzung des Projekts beginnt. Ein Modell, das eine getreue Nachbildung eines kompletten Ausschnitts der Software darstellt, nennt sich Prototyp (Muster, Vorläufer).

8.8. Polymorphie

Wikipedia. Polymorphie oder Polymorphismus (griechisch für Vielgestaltigkeit) ist ein Konzept in der objektorientierten Programmierung, das ermöglicht, dass ein Bezeichner abhängig von seiner Verwendung unterschiedliche Datentypen annimmt.

8.8.0.1. Statische Polymorphie (Überladen) Stellen Sie sich vor, das Objekt *Adam* teilt dem Objekt *FauleSocke* mit, dass es laufen soll, und zwar mit der Geschwindigkeit 5 km/h. Was wird passieren? - *FauleSocke* wird sich mit dieser Geschwindigkeit fortbewegen. Offensichtlich ist die Richtung ebenso egal wie die Dauer. Was würde passieren, wenn *Adam* abermals *FauleSocke* mitteilt, er solle laufen, und zwar 10 Minuten? *FauleSocke* würde 10 Minuten lang mit 5 km/h laufen und danach stehen bleiben. Damit *FauleSocke* den etwas wirr klingenden Anweisungen seines Reiters Folge leisten kann, benötigt er Methoden unterschiedlicher Gestaltung. Er benötigt eine Methode, die auf den Parameter Geschwindigkeit reagiert, und eine Methode, die auf den Parameter Zeitdauer reagiert. obwohl die Methoden den gleichen Namen tragen, führen sie zu einer unterschiedlichen Verarbeitung durch das Objekt *FauleSocke* . Der Fachausdruck für diese Technik heisst Überladen.

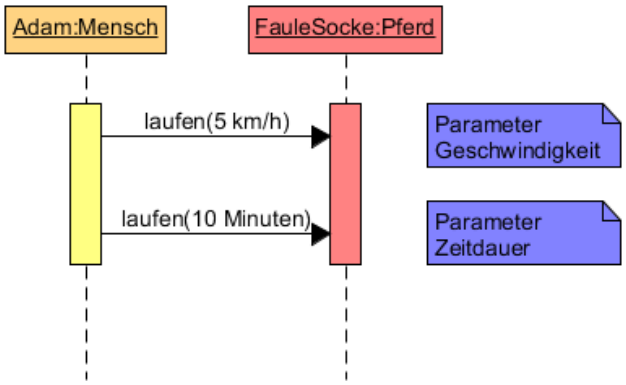


Abbildung 29: Verschieden gestaltete Methoden

8.8.0.2. Dynamische Polymorphie (Überschreiben) Anders als bei der Mehrfachvererbung sieht es aus, wenn man Eigenschaften der Basisklasse bei der Vererbung bewusst umgehen möchte. Dazu möchte ich nochmals auf das Beispiel der Basisklasse Säugetiere zurückgreifen. Angenommen, Sie möchten in der abgeleiteten Klasse Pferd bestimmen, auf welche Weise sich Pferde-Objekte verständigen. Dazu überschreiben Sie die Methode Verständigen und legen die Art und Weise des Wieherns in der Klasse Pferd für die abgeleiteten Objekte fest. Das Überschreiben von Methoden ist ein sehr mächtiges Mittel der objektorientierten Programmierung. Es erlaubt Ihnen, unerwünschte Erbinformationen teilweise oder ganz zu unterdrücken und damit eventuelle Designfehler - in Grenzen - auszugleichen beziehungsweise Lücken in der Basisklasse zu füllen. Dabei ist die Technik extrem simpel. Es reicht aus, eine identische Methode in der abgeleiteten Klasse Pferd zu beschreiben, damit sich Objekte wie FauleSocke plötzlich anders verhalten.

8.9. Abstract

Wikipedia. Eine abstrakte Klasse bezeichnet in der objektorientierten Programmierung eine spezielle Klasse, welche sich per Definition nicht instanziieren lässt (d.h. es lassen sich keine Objekte von ihr erzeugen) und die somit lediglich als Strukturelement innerhalb einer Klassenhierarchie dient. Innerhalb von abstrakten Klassen besteht die Möglichkeit abstrakte Methoden (also Methoden ohne „Rumpf“ (Implementierung) nur mit der Signatur) zu deklarieren. Der Modifikator *abstract* kann bei einer Methode, Klasse oder einem Interface verwendet werden. Dies bedeutet, dass das mit *abstract* gekennzeichnete Element noch nicht vollständig implementiert worden ist. Ist eine Methode oder ein Interface mit dem Mofikator *abstract* versehen, so muss die implementierende Klasse ebenfalls als *abstract* deklariert werden. Wenn eine Klasse mit *abstract* gekennzeichnet wird, so kann von dieser kein Objekt erzeugt werden. Eine von einer abstrakten Klasse abgeleitete Subklasse muss alle abstrakten Methoden der Elternklasse vollständig implementieren, anderenfalls muss sie ebenfalls als *abstract* deklariert werden. Im Gegensatz zu einem Interface kann eine abstrakte Klasse auch implementierte Methoden und Attribute (die nicht als *static* oder *final* sind) wurden, beinhalten. Findet bei einer abstrakten Klasse also keinerlei Implementation statt, ist es sinnvoller diese als Interface zu deklarieren.

8.10. Interfaces (Schnittstellen)

In Java existieren neben Klassen auch Interfaces. Diese werden anstatt dem Schlüsselwort *class* mit dem Schlüsselwort *interface* eingeleitet. Interfaces sollten ebenso wie Klassen auch jeweils in einer separate Java-Datei gespeichert werden. Vom Aufbau her sind Interfaces einer Klasse sehr ähnlich, sie sind nahezu vergleichbar abstrakten Klassen, welche nur Methodendeklarationen enthalten. Der einzige Unterschied zu einer Klasse besteht darin, dass ein Interface keine Implementierungen besitzt, sondern nur Methodenköpfe und Konstanten. Schauen wir uns einmal den Aufbau eines Interfaces an.

```
Modifikator interface MyInterface {
  final Modifikator Datentyp variable = Wert;
  Modifikator Datentyp methode();
}
```

Der Kopf eines Interfaces ist aufgebaut wie der einer Klasse, nur dass das Schlüsselwort *class* durch *interface* ersetzt wird. In dem Rumpf des Interfaces können sich nur Konstanten und Methodennamen befinden. Interfaces werden unter anderem dazu verwendet, um die Spezifikation von Klassen von deren Implementierung zu trennen. Man kann aber auch durch Interfaces die eigentliche Implementierung vor Dritten schützen. Daher kann man auch Interfaces und deren enthaltene Methoden als Kommunikationsschnittstelle benutzen, da durch den Methodenkopf die zu erwartenden Übergabeparameter und der Rückgabewert festgelegt sind. Ein Interface kann auch, wie Klassen, eine Vererbungshierarchie besitzen. Dort gelten aber die selben Regeln wie bei Klassen (Einfachvererbung, Sichtbarkeit etc.). Da wir aber auch Interfaces verwenden wollen, müssen wir Sie irgendwie in eine Klasse einbinden. Dies geschieht über das Schlüsselwort *implements*. Das nachfolgende Beispiel zeigt, wie das obige Interface MyInterface in eine Klasse eingebunden wird.

```
Modifikator class Klasse extends Superklasse implements MyInterface {
  // Anweisungen
}
```

Wir können den normalen Aufbau einer Klasse einfach mit dem neuen Interface erweitern, indem wir vor dem Klassenrumpf mit dem Schlüsselwort *implements* unser Interface MyInterface einbinden. Das Einbinden eines Interfaces kann parallel zur Vererbung stattfinden. Man kann beliebig viele Interfaces in eine Klasse einbinden, die Interfacenamen werden dann im Klassenkopf einfach durch ein Komma getrennt. Dies macht man sich häufig zunutze, um in Java eine Art Mehrfachvererbung zu simulieren, da dies ansonsten nicht direkt möglich ist. Schauen wir uns direkt ein konkretes Beispiel an.

```
interface PunktSchnittstelle {
  /* Methoden, die von einem Punkt implementiert werden sollen */
```

```
  public int getX();public void setX(int i);
}
/* Unsere Klasse Punkt,
 * die das Interface PunktSchnittstelle einbindet */
public class Punkt implements PunktSchnittstelle {
  private int x;
  /* Implementierung der Methode getX
   * aus dem Interface PunktSchnittstelle */
  public int getX() {
    return x;
  }
  /* Implementierung der Methode setX
   * aus dem Interface PunktSchnittstelle */
  public void setX(int i) {
    x = i;
  }
}
```

In dem obigen Beispiel haben wir ein Interface mit dem Namen *PunktSchnittstelle*, welche in die Klasse *Punkt* eingebunden wird, erstellt. Die Klasse *Punkt* muss anschliessend die Methodendeklarationen aus dem Interface implementieren.

8.10.1. Nutzung mehrerer Interfaces in einer Klasse

Da wir in Java nur eine indirekte Mehrfachvererbung über das Einbinden mehrerer Interfaces simulieren können, wollen wir uns jetzt etwas näher mit diesem Thema befassen. Das Einbinden von mehreren Interfaces ist nicht immer möglich. Es kann z.B. sein, dass die gleiche Methode in zwei Interfaces deklariert ist und die Methodendeklarationen sich nur durch den Rückgabewert unterscheiden. In diesem Fall ist die Einbindung von diesen beiden Interfaces gleichzeitig nicht möglich. Sollten die beiden Methodendeklarationen gleich sein, so ist das Einbinden der beiden Interfaces möglich, die Methode ist dann allerdings auch nur einmal in der Klasse vorhanden. Konstanten von unterschiedlichen Interfaces können zwar denselben Namen, aber einen anderen Wert haben. Dies sollte allerdings vermieden werden. Man kann dann allerdings mit der Angabe des Interface namens auf die jeweilige Konstante zugreifen. Schauen wir uns dazu direkt einmal ein Beispiel an.

```
/**
*  Interface Schnitt1
*/
interface Schnitt1 {
/* Konstanten var1 und var2 */
public static final int var1 = 1;
public static final int var2 = 2;

/** Methode setWert mit einem Übergabeparameter vom Datentyp int */
public void setWert(int wert);
}

/**
*  Interface Schnitt2
*/
interface Schnitt2 {
/**
* Konstanten var1 und var3(var1 ist schon im Interface Schnitt1 deklariert)
*/
public static final int var1 = 3;
public static final int var3 = 4;

/**
* Methode setWert mit einem Übergabeparameter vom Datentyp int (identisch
* mit der Methode aus Schnitt1)
*/
public void setWert(int wert);
}

/**
*  Startklasse Test, die unsere Interfaces Schnitt1 u. Schnit2 einbindet
*/
public class Test implements Schnitt1, Schnitt2 {
/** lokal definierte Konstante var1 */
private static final int var1 = 10;
```

```
/** implementierte Methode aus dem Interface Schnitt1 und Schnitt2 */
public void setWert(int wert) {
/* Anweisungen */
}

/**
* main-Methode
* @param args
*/
public static void main(String[] args) {
/* Ausgabe der Konstanten, wie lautet hier die Ausgabe? */
System.out.println(var1);
System.out.println(var2);
System.out.println(var3);
System.out.println(Schnitt1.var1);
System.out.println(Schnitt2.var1);
}
}
```

In dem obigen Beispiel haben wir zwei Interfaces *Schnitt1* und *Schnitt2*. Beide besitzen jeweils zwei Konstanten und eine Methode. Anschliessend folgt die Startklasse *Test*, welche unsere beiden Interfaces *Schnitt1* und *Schnitt2* einbindet. Die Methode aus den Interfaces *Schnitt1* und *Schnitt2* ist unkritisch, da sie in beiden Klassen gleich deklariert ist. Zusätzlich besitzen die Interfaces jeweils eine Konstante mit gleichem Namen *var1*, aber unterschiedlichem Wert und jeweils eine komplett unterschiedliche Konstante, *var2* und *var3*. Unsere Startklasse *Test* implementiert nun beide Interfaces. Die Klasse *Test* besitzt noch eine eigene Konstante mit dem Namen *var1*. Insgesamt gibt es die Konstante *var1* also dreimal mit jeweils einem anderen Wert. Die Methode *setWert* wird in der Klasse *Test* implementiert, d.h. wir geben ihr an dieser Stelle einen Methodenrumpf.

In der main-Methode geben wir nun die Konstanten aus. Wie lautet hier die Ausgabe?

8.10.2. Interface als Datentyp

Interfaces können in Java auch als Datentyp verwendet werden. Dabei ist allerdings zu beachten, dass von einem Interface kein Objekt mit dem new-Operator erzeugt werden kann. Eine Instanz eines Interfaces kann nur über Zuweisungen erstellt werden, indem ein Objekt einer das Interface implementierenden Klasse einer Referenzvariablen, die das Interface zum Datentyp hat, zugewiesen wird. Dadurch wird es einem, wie im nachfolgenden Beispiel gezeigt wird, auch möglich, Objekte unterschiedlicher Klassen in einem Array zu speichern.

8.10.3. Spezielle Interfaces

In Java gibt es schon einige vordefinierte Interfaces, die man für bestimmte Aufgaben einbinden kann. Wir wollen in den hier folgenden Unterkapiteln einige speziellere Interfaces betrachten. Oft werden Interfaces zur Abhandlung bzw. Reaktion auf spezielle Ereignisse verwendet. Ein großer Teil von Interfaces wird im Bereich der GUI (Graphical User Interface)-Entwicklung verwendet. Grafische Oberflächen müssen z.B. auf Eingaben reagieren, dafür gibt es spezielle Interfaces. Auf diese speziellen Interfaces gehen wir aber später noch einmal bei (Event Handling) näher ein.

8.10.4. Marker Interfaces

Eine Marker-Schnittstelle in Java ist ein Interface ohne Feld oder Methoden oder in einfachen Worten ein leeres Interface. Beispiel für Marker Interface sind die *Serializable*, *Clonnable* und *Remote*-Schnittstelle. Nun, wenn Marker-Interface keine Felder und keine Methode und auch kein Verhalten implementieren, wozu sollen Sie den gut sein?

8.10.5. Warum Klassen mit Marker Interfaces markiert werden

Wenn wir die Interfaces *Serializable*, *Clonnable* und *Remote* genauer betrachten, stellen wir fest, dass diese benötigt werden um irgendwas spezielles an der Klasse zu tun. Wenn also die JVM eine solche

Klasse 'sieht' so erledigt die JVM etwas im voraus abgemachtes spezielles mit der Instanz einer solche markierten Klasse. Beim Interfaces *Clonnable* zum Beispiel kann die JVM den Speicherbereich weit effizienter kopieren als dies mit reinen Java Möglichkeiten gegeben wäre.

Allerdings muss man sich bei einem solchen speziellen Konstrukt auch die Frage gefallen lassen warum dies nicht auch per speziellem Attribut innerhalb der Klasse oder einer Annotation erledigt werden kann? Dies wäre durchaus möglich mit Hilfe eines *boolean* Attributes oder auch einem *String* aber wenn ein Interface benutzt wird ist dies besser lesbar und verbessert die Benutzung von Polymorphismus.

8.10.6. Where Should I use Marker interface

in Java Apart from using built in marker interface for making a class Serializable or Clonnable. One can also develop his own marker interface. Marker interface is a good way to classify code. You can create marker interface to logically divide your code and if you have your own tool than you can perform some pre-processing operation on those classes. Particularly useful for developing API and framework like Spring or Struts. After introduction of Annotation on Java5, Annotation is better choice than marker interface and JUnit is a perfect example of using Annotation e.g. @Test for specifying a Test Class. Same can also be achieved by using Test marker interface.

Another use of marker interface in Java

One more use of marker interface in Java can be commenting. a marker interface called Thread Safe can be used to communicate other developers that classes implementing this marker interface gives thread-safe guarantee and any modification should not violate that. Marker interface can also help code coverage or code review tool to find bugs based on specified behavior of marker interfaces. Again Annotations are better choice @ThreadSafe looks lot better than implementing ThraedSafe marker interface.

In summary marker interface in Java is used to indicate something to compiler, JVM or any other tool but Annotation is better way of doing same thing.

Read more: http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html#ixzz41w4ptObP

8.10.7. Interface Cloneable

Ein spezielles Interface ist das Interface *Cloneable*. Dies dient dazu, eine exakte Kopie von einem Objekt zu erstellen. Normalerweise würde man denken, dass bei einer Zuweisung ebenfalls eine exakte Kopie eines Objektes entsteht. Dies ist aber nur bei einfachen Datentypen der Fall. Bei Objekten allerdings findet bei einer Zuweisung lediglich eine Referenzierung auf den Speicherbereich, in dem das Objekt liegt, statt (siehe auch 7.0.0.2 Zuweisungen zwischen Objekt variablen).

8.11. Pakete

Ein Paket ist eine Bündelung von zusammengehörigen Klassen. Man könnte auch sagen, dass alle Klassen in einem Paket dasselbe Thema behandeln. Ein Beispiel für eine Bündelung von Klassen ist z.B. das Paket javax.swing aus der Java Standard Edition (JSE). Das Swing-Paket enthält Klassen, mit denen man grafische Benutzeroberflächen programmieren kann. Ein Paket ist physikalisch gesehen auf der Festplatte nichts anderes als ein Verzeichnis. Die Pakete können eine beliebig tiefe Schachtelung haben. Der Punkt wie bei javax.swing ist vergleichbar mit einem Slash bei Pfadangaben, um Über- und Unterordner zu trennen. In Java werden Pakete in Archiven mit der Endung .jar komprimiert. Pakete dienen der einfachen Wiederverwendung von immer wiederkehrendem Quellcode. So kann ein schon vorhandenes Paket einfach in ein Projekt eingebunden werden und dort direkt verwendet werden ohne den Quellcode neu schreiben zu müssen. So kann man sich im Laufe der Zeit auch eigene nützliche Bibliotheken anlegen, die man für seine Projekte wiederverwendet. Schauen wir uns mal ein kleines Beispiel an:

```
// Unsere Klasse Beispielklasse befindet sich in dem Paket beispielpaket
package beispielpaket;
// Hier werden alle Klassen des Paketes io importiert
import java.io.*;
public class Beispielklasse{
  public void main (String [] args) {
    System.out.println("Hello World!");
  }
}
```

Das Paket wird über das Schlüsselwort *package* definiert. Anschließend folgt der Name des Paketes. Ebenso wie bei Variablen werden Paketnamen grundsätzlich klein geschrieben. Die Paketdefinition ist immer der erste Eintrag in einer Java-Datei. Externe Klassen können über eine *import* Anweisung eingebunden werden. In dem obigen Beispiel werden z.B. alle Klassen (gekennzeichnet durch den Stern) aus dem Paket io importiert. Das io-Paket liegt wiederum in dem Paket java. Ineinander geschachtelte Pakete werden über die Punktnotation miteinander verknüpft. Als Verzeichnisstruktur veranschaulicht, würde der Pfad so aussehen:

.../java/io/

Die *import* Anweisungen befinden sich in Java immer vor der eigentlichen Klassen- und nach der Paket-Deklaration. Methoden, die innerhalb einer Klasse mit dem Modifikator *protected* versehen sind, sind auch nur in diesem Paket zugänglich. Die Klasse selbst kann zwar in einem anderen Paket verwendet werden, aber nicht die geschützte Methode. Subklassen müssen nicht demselben Paket angehören wie ihre Superklasse.

8.12. Designregeln

Auch wenn die objektorientierte Softwareentwicklung im Vergleich zur konventionellen Programmierung gutes Softwaredesign besser unterstützt, ist sie auf keinen Fall eine Garantie für sauber strukturierte und logisch aufgebaute Programme. Die objektorientierte Programmierung erleichtert zwar gutes Softwaredesign, sie erzwingt es jedoch nicht. Da man trotz Objektorientierung schlechte Programme entwickeln kann, sollten Sie einige Grundregeln beachten:

- Setzen Sie die Vererbung bewusst und sparsam ein.
- Reduzieren Sie die Anforderungen auf das Wesentliche.
- Kapseln Sie alle Attribute und Methoden, die nicht sichtbar sein müssen.
- Arbeiten Sie bei grossen Projekten mit einem Modell.
- Verwenden Sie einen Prototyp.

8.13. Zusammenfassung

Die objektorientierte Programmierung war eine Antwort auf die Softwarekrise in der Mitte der 60er-Jahre des letzten Jahrhunderts. Durch Objektorientierung lässt sich die natürliche Welt leichter in Computerprogrammen umsetzen. Diese objektorientierten Computerprogramme bestehen aus einer Sammlung eines oder mehrerer Objekte. Ein Objekt lässt sich mit einem natürlichen Lebewesen vergleichen und verfügt über eine Gestalt und Fähigkeiten. Die Gestalt prägen Attribute, während die Fähigkeiten von Methoden bestimmt sind. Beide Bestandteile eines Objekts sind in der Klasse festgelegt, von der ein Objekt abstammt. sie liefert den Bauplan für gleichartige Objekte. Objektorientierte Programmierung ist kein Allheilmittel. Sie unterstützt gutes Design, ohne es zu erzwingen. Es ist deshalb notwendig, auf sauberes Design zu achten, wenn man mit objektorientierter Programmierung erfolgreich sein will.

8.14. Aufgaben

8.14.0.1. Allgemeine Fragen

1. Worin unterscheiden sich Klassen von Objekten?

2. Wie unterscheiden sich Objekte der gleichen Klasse voneinander?
3. Was bedeutet der Begriff Basisklasse?
4. Was bedeutet der Begriff, abgeleitete Klassen?
5. Wie verständigen sich Objekte untereinander?
6. Welche Arten von Beziehungen gibt es, und wie unterscheiden sie sich?
7. Worin liegt die Gefahr bei Vererbungsbeziehungen?

9. Persistenz

10. Testing

11. Dokumentation

12. Build

Teil III.
Vom Text zum ausführbaren Programm

13. Der lange Weg...

13.1. Der Quell-Code

Java-Code kann mit Hilfe eines einfachen Texteditors geschrieben werden. Dabei wird üblicherweise eine Java-Klasse in eine Datei geschrieben. Dabei ist streng darauf zu achten, dass die Datei den Namen der Klasse mit der Erweiterung .java erhält.

Beispiel:

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

Der Text wird in die Datei Hello.java abgespeichert.

13.2. Das Class-File

Das *.class ('ausführbarer Code') wird mithilfe des Compilers erstellt.

Im selben Ordner wie die Datei Hello.java wird eine Kommando Zeile gestartet und mit dem Befehl

```
javac Hello.java
```

wird die Datei kompiliert und die neue Datei *Hello.class* wird erzeugt.

13.3. Starten des Programms

Die Java Virtuelle Maschine kann *.class Dateien ausführen

```
java Hello
```

Die Ausgabe

```
Ich kann Programmieren!
```

zeigt, dass der Vorgang erfolgreich war :-)

13.3.1. Aufgabe:

13.3.1.1. Übergabeparameter Ergänzen sie das Programm derart, dass die Übergabeparameter vom Programm ausgegeben werden.

Beispiel, als Übergabeparameter werden hier die Worte 'parameter1 und noch ein Parameter' verwendet.

```
java Hello parameter1 und noch ein Parameter
```

finden sie durch Überlegung und versuchen heraus welche Ausgabe von ihrem Programm erzeugt wird für folgende Aufrufe.

```
java Hello "parameter1␣und␣noch␣ein␣Parameter"
```

```
java "Hello␣parameter1␣und␣noch␣ein␣Parameter"
```

```
"java␣Hello␣parameter1␣und␣noch␣ein␣Parameter"
```

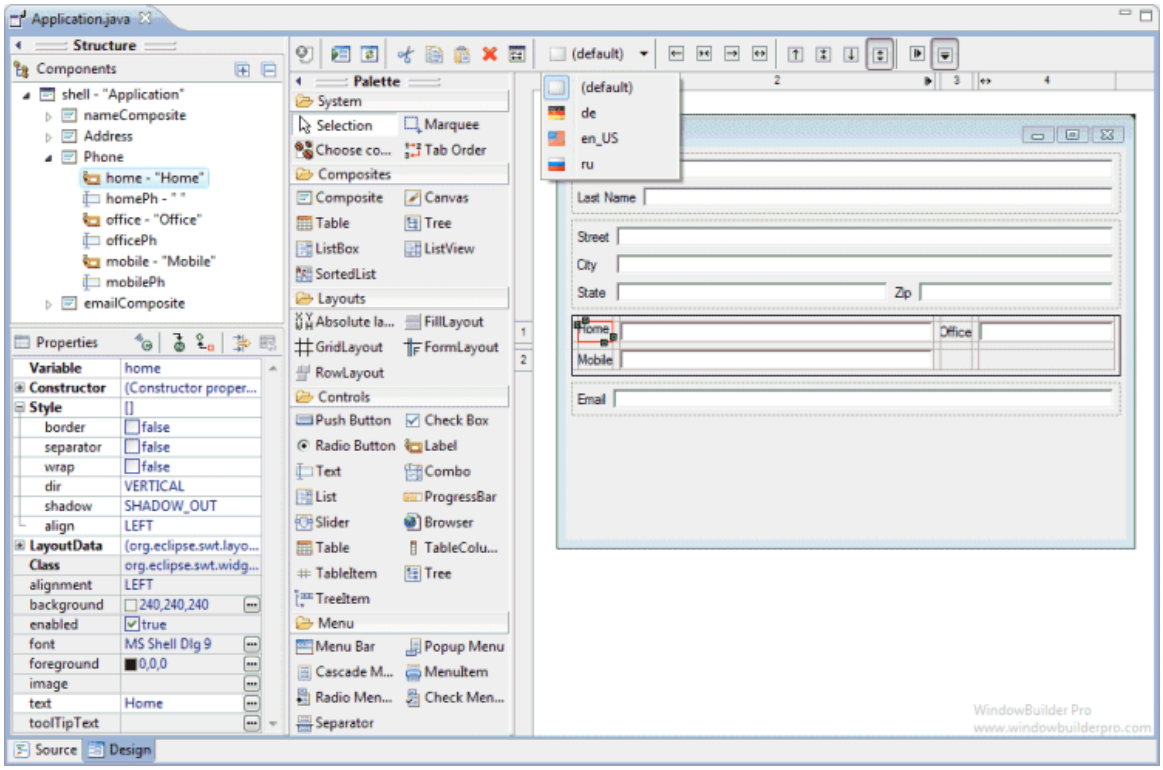
13.3.1.2. Elektrischer Widerstand Es soll ein Programm erstellt werden welches der elektrische Widerstand aus n seriellen Widerständen berechnet. In dem Programm soll die Übergabe Möglichkeit von Parametern genutzt werden.

Teil IV.
Anhang

A. Eclipse

A.1. Installation nützlicher Plugins

A.1.1. Window Builder



folge: WindowBuilder download
oder füge das repository zu den verfügbaren Software Seiten hinzu.
Name: WindowBuilder
Update Sites

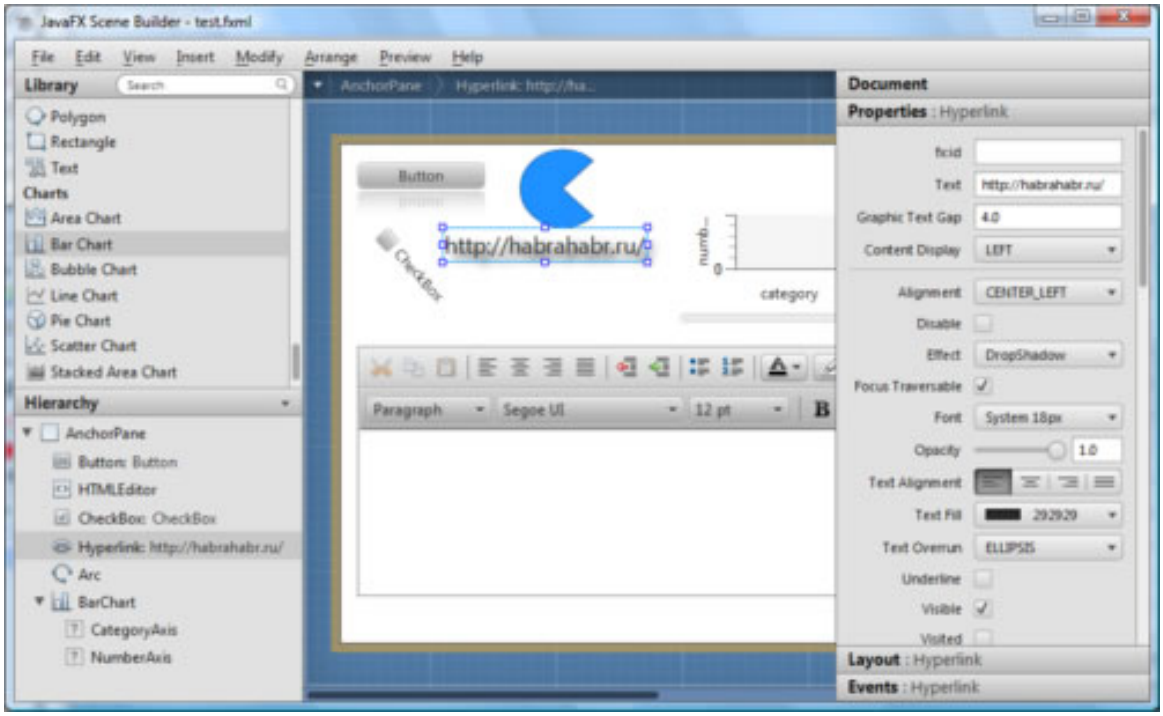
4.5 Mars	http://download.eclipse.org/windowbuilder/WB/release/R201506241200-1/4.5/
4.4 Luna	http://download.eclipse.org/windowbuilder/WB/release/R201506241200-1/4.4/

A.1.2. JavaFx

folge: install e(fx)clipse
Oder füge das repository zu den verfügbaren Software Seiten hinzu.
Name: e(fx)clipse
Update Site

Alle Versionen	http://download.eclipse.org/efxclipse/updates-released/2.3.0/site
JavaFX Tutorial	http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

A.1.3. JavaFx SceneBuilder



Oracle führt den SceneBuilder im Moment nicht mehr weiter. Auf verschiedenen Webseiten habe ich gelesen, dass Oracle mit Java 1.9 den SceneBuilder fix in das JDK einbauen will. Auf anderen habe ich gelesen, dass der SceneBuilder Opensource sei und dieser von der Opensource Gemeinde weiter gepflegt wird. Wie auch immer.... Der aktuelle Download ist ab hier möglich.
http://gluonhq.com/open-source/scene-builder/

A.1.4. PHP Untersützung

Öffne den Eclipse Marktplatz 'Eclipse → Help → Eclipse Marketplace...' Im aufspringenden Dialog nach 'PHP Development Tools' suchen.

A.1.5. Python Untersützung

Öffne den Eclipse Marktplatz 'Eclipse → Help → Eclipse Marketplace...' Im aufspringenden Dialog nach 'pyDev' suchen.

A.1.6. Node.js Untersützung

Öffne den Eclipse Marktplatz 'Eclipse → Help → Eclipse Marketplace...' Im aufspringenden Dialog nach 'nodeclipse' suchen.

A.1.7. Remote System Explorer (Beispiel RaspBerry)

Dieses Plugin gehört zur Standard Installation und muss daher NICHT installiert werden! Falls es dennoch notwendig wird: Öffne den Eclipse Marktplatz 'Eclipse → Help → Eclipse Marketplace...' Im aufspringenden Dialog nach 'Remote System Explorer' suchen.

Abbildungsverzeichnis

Tabellenverzeichnis

List of Codes

Literatur

[1] © Alex Abrecht. PROGRAM DESIGN, Die strukturierte Programmierung,nach Nassi - Schneidermann, 2013 5

[2] Daniel Sterchi. Script für die Java Vorlesung an der TSBE für Telematiker. Version 2.3.1, 28 April 2005.

[ZUM] Zentrale für Unterrichtsmedien im Internet http://www.zum.de

[3] Henning/Vogelsang. Taschenbuch Programmier-sprachen, Fachbuchverlag Leipzig im Carl Hanser Verlag, 2004

[4] Hanspeter Mössenböck. Sprechen Sie Java? Eine Einführung in das systematische Programmieren. dpunkt.verlag

Abkürzungsverzeichnis

Allozieren Unter allozieren (die Allokation) versteht man das dynamische belegen von Speicher.

Anweisungsblock Wikipedia
(engl. Statement Block) Ein Anweisungsblock dient dazu, mehrere Anweisungen zu gruppieren. Dadurch können mehrere Anweisungen wie eine einzelne Anweisung interpretiert werden, was von einigen Sprachkonstrukten vorausgesetzt wird – wie etwa von den Kontrollstrukturen.

Assoziativ Wikipedia
Das Assoziativgesetz (lat. associare „vereinigen, verbinden, verknüpfen, vernetzen“), auf Deutsch Verknüpfungsgesetz oder auch Verbindungsgesetz, ist eine Regel aus der Mathematik. Eine (zweistellige) Verknüpfung ist assoziativ, wenn die Reihenfolge der Ausführung keine Rolle spielt. Anders gesagt: Die Klammerung mehrerer assoziativer Verknüpfungen ist beliebig. Deshalb kann man es anschaulich auch „Klammergesetz“ nennen.

Neben dem Assoziativgesetz sind Distributivgesetz und Kommutativgesetz von elementarer Bedeutung in der Mathematik.

Ausdruck Wikipedia
(engl. Expression) Unter einem Ausdruck wird ein beliebig komplexes Sprachkonstrukt verstanden, dessen Auswertung einen einzigen wohl definierten Wert ergibt. Im einfachsten Fall ist ein Ausdruck eine Konstante, wie z.B. das Schlüsselwort true oder auch eine Zahl (z.B. 1 oder 0x7fff). Komplexere Ausdrücke sind

	Vergleiche oder Berechnungen mit mehreren Variablen und Konstanten. Innerhalb der Grammatikbeschreibungen wird hier häufig noch nach verschiedenen Arten von Ausdrücken (z.B. numerisch, literal, etc.) unterschieden.
Deklaration	Mit der Deklaration einer Variable wird diese bekannt gemacht, aber noch kein Wert zugewiesen. Siehe auch Initialisierung.
Distributiv	Wikipedia Die Distributivgesetze (lat. distribuere „verteilen“) sind mathematische Regeln, die angeben, wie sich zwei zweistellige Verknüpfungen bei der Auflösung von Klammern zueinander verhalten, nämlich dass die eine Verknüpfung in einer bestimmten Weise verträglich ist mit der anderen Verknüpfung. Insbesondere in der Schulmathematik bezeichnet man die Verwendung des Distributivgesetzes zur Umwandlung einer Summe in ein Produkt als Ausklammern oder Herausheben. Das Auflösen von Klammern durch Anwenden des Distributivgesetzes wird als Ausmultiplizieren bezeichnet. Das Distributivgesetz bildet mit dem Assoziativgesetz und dem Kommutativgesetz grundlegende Regeln der Algebra.
Initialisierung	Bei der Initialisierung einer Variable wird dieser ein Wert zugewiesen. (sprich Wertezuweisung).
Kommutativ	siehe auch Wikipedia Das Kommutativgesetz (lat. commutare „vertauschen“), auf Deutsch Vertauschungsgesetz, ist eine Regel aus der Mathematik. Wenn sie gilt, können die Argumente einer Operation vertauscht werden, ohne dass sich am Ergebnis etwas ändert. Mathematische Operationen, die dem Kommutativgesetz unterliegen, nennt man kommutativ. Das Kommutativgesetz bildet mit dem Assoziativgesetz und Distributivgesetz grundlegende Regeln der Algebra.
Literal	siehe auch Wikipedia Ein Literal ist ein konstanter Ausdruck. Es gibt verschiedene Typen von Literalen: die Wahrheitswerte true und false integrale Literale für Zahlen, etwa 123 Zeichenliterale, etwa ’J’ oder ’n’ Fliesskommaliterale, etwa 123.456789 oder 9.999E-2 Stringliterale für Zeichenketten, wie Java ist auch eine Insel null steht für einen besonderen Referenztyp.
promiscuous	Der promiscuous mode (engl. etwa freizügiger Modus) bezeichnet einen bestimmten Empfangsmodus für netzwerktechnische Geräte. In diesem Modus liest das Gerät den gesamten ankommenden Datenverkehr an die in diesen Modus geschaltete Netzwerkschnittstelle mit und gibt die Daten zur Verarbeitung an das Betriebssystem weiter. Bei Wireless LANs werden im promiscuous mode nur die Pakete des Netzwerks (Accesspoints) weitergeleitet, mit dem der Client gerade verbunden ist. Da das Herstellen einer Verbindung mit dem Netzwerk normalerweise mit einer Authentifizierung einher geht, ist der promiscuous mode nicht geeignet, um Pakete eines Netzwerks aufzufangen, zu dem man keinen direkten Zugang hat.