Java für Frischlinge

Eine kleine Zusammenfassung

Roland Friebel, Christian Nitschke und Jonathan Hofinger

22. Februar 2016

Vorwort

Hallo Verzweifelter! Schön, dass du auf die Hilfe unserer Zusammenfassung bauen willst. Allerdings gibt es noch ein paar Dinge, die wir vorher klären sollten:
.

- Diese Zusammenfassung dient primär als Hilfe für die Tutorien „Informatik im Maschinenbau" am KIT (Ratlose aus anderen Veranstaltungen sind uns aber trotzdem willkommen =)) und als Unterstützung bei der Klausurvorbereitung. Dieses Kompendium ersetzt nicht den Besuch der Vorlesung oder die Vorlesungsfolien, da dort die Konzepte hinter der Programmierung sowie Programmiermuster erklärt werden.

- Es sei erwähnt, dass diese Zusammenfassung ursprünglich für die Jahrgänge WS09/10 & WS10/11 geschrieben wurde und sich an dem Stoff dieser Jahrgänge orientiert. Aus diesem Grund geht die Zusammenfassung an einigen Stellen auch deutlich über den aktuellen Stoff hinaus.

- Wer einen Fehler gefunden zu haben meint, soll uns bitte Bescheid geben (direkt oder indirekt über eine Mail an die Fachschaft: fachschaft@fmc.uni-karlsruhe.de)

- Wichtig wäre mir noch, dass diese Zusammenfassung nur über die Fachschafts-Homepage (in der Toolbox) verbreitet wird (Bitte nicht direkt rumschicken, sondern einfach auf die FMC-Homepage verweisen!)

- Ach ja, das Durchlesen dieser Zusammenfassung führt natürlich weder dazu, dass du programmieren kannst, noch, dass du sicher die Klausur bestehst. Sie kann aber zu beidem beitragen, besonders, wenn du die Beispiele nachvollziehst und mit dem Code herumspielst, um zu sehen, was für einen Effekt deine Änderungen haben.

Ansonsten: Viel Spaß & Erfolg - Java ist lang nicht so schwer, wie immer alle behaupten ;)

Ein Wort zum Typesetting

Dieses Dokument wurde in LaTeX gesetzt. Zur Darstellung von Quellcode wird das Paket `minted` verwendet.
Folgende Regeln ziehen sich durch das Dokument:

- **Fachbegriffe** werden fett gedruckt.
- `Quellcode` wird monospaced gedruckt.

- „Fallstricke“ für Anfänger werden mit einem Warnhinweis links neben den Text gekennzeichnet.

- Merkhilfen und Merksätze werden durch eine Glühbirne gekennzeichnet.
- Kommentare im Quellcode werden grün dargestellt: `//so zum Beispiel`.
- Exkurse, die dem Verständnis des Codes dienen, aber zum eigentlichen Programmieren und für die Klausur nicht zwingend vorausgesetzt werden, werden grau hinterlegt.

Grundsätzliche Java-Regeln

Wichtig für das Verständnis dieses Kompendiums ist, sich folgende Regeln einmal eingeprägt zu haben:

- Ein Befehl in Java wird immer durch ein Semikolon `;` beendet. Zwischen zwei Befehlen kann, muss aber keine neue Zeile angefangen werden. Ersteres macht den Code jedoch deutlich übersichtlicher.
- Leerzeilen und Tabulatoren werden vom Compiler ignoriert, außer sie befinden sich in einer durch Anführungszeichen definierten Zeichenkette oder anderen Literalen (z.B. Zahlen).
- Kommentare im Quellcode können auf zwei verschiedene Weisen erzeugt werden: `// so` (also durch zwei /-Zeichen) und `/* so */`. Der Unterschied ist, dass die erste Möglichkeit nur vom aktuellen Zeichen bis zum Ende der Zeile auskommentiert und die zweite Möglichkeit auch über mehrere Zeilen gehen kann.
  Kommentare werden vom Kompiler ignoriert.

Die meisten restlichen Zeichen, die in Java-Code auftauchen können, werden in diesem Dokument erwähnt.

Inhaltsverzeichnis

1 Die Zwischenablage: Variablen 6
1.1 Erstmal Vorstellen: Die Deklaration . . . 6
1.2 Das erste Mal: Initialisierung . . . 6
1.3 Die treuen Gefährten: Konstanten . . . 7
1.4 Die kleinen Unterschiede: Verschiedene Typen von Variablen . . . 7
1.4.1 Lokale Variablen . . . 8
1.5 Mehrere Variablen auf einem (ordentlichen) Haufen: „Arrays“ . . . 8

2 Datentypen 10
2.1 Primitive Datentypen . . . 10
2.1.1 Überlauf: Ein voller Eimer wird plötzlich leer . . . 12
2.2 Umwandlung von Datentypen . . . 12
2.2.1 Automatische (implizite) Typumwandlung . . . 13
2.2.2 Explizite Typumwandlung: Den Computer zu seinem Glück zwingen . . . 13
2.3 Zusammengesetzte Datentypen . . . 14

3 Fallunterscheidungen mit `if` und `else` 15
3.1 Bedingung abfragen mit `if` . . . 15
3.2 Alternativfälle mit `else` . . . 16
3.3 Kompliziertere Strukturen mit `else if` . . . 17
3.4 Fallunterscheidungen mit `switch` . . . 18

4 Und täglich grüßt das Murmeltier: Schleifen 19
4.1 Einfache Schleifen mit `while` . . . 19
4.2 Für die Langsamen: `do-while`-Schleifen . . . 20
4.3 Fortschritt ahoi: `for`-Schleifen . . . 20
4.4 Sprunganweisungen . . . 21
4.4.1 Das war’s: `break` . . . 21
4.4.2 So wird’s nix: `continue` . . . 22

5 Bedingungen verknüpfen mithilfe Boolescher Operatoren 22

6 Etwas erledigen lassen: Methoden 23
6.1 Sichtbarkeit . . . 24
6.2 Rückgabewert . . . 25
6.3 Methodennamen . . . 25
6.4 Methodenparameter . . . 25
6.5 Weitere Schlüsselwörter . . . 26
6.6 Der Ursprung: die main-Methode . . . 27
6.7 Überladen von Methoden . . . 28
6.8 Rekursion . . . 28

7 Objektorientiertes Programmieren 29
7.1 Objektorientierte Sprachen . . . 29
7.2 Objektorientierung . . . 30
7.3 Die Schablonen: Klassen . . . 31
7.3.1 Die Deklaration einer Klasse . . . 31

7.3.2 Jetzt geht's los: Instanzen . . . 31
7.3.3 Konstruktoren - Die Baumeister . . . 32
7.3.4 Instanzvariablen . . . 33
7.3.5 Klassenvariablen - die große Gemeinsamkeit . . . 34
7.3.6 Vorsicht Nebel - Verdeckung . . . 35
7.3.7 Zugriffe steuern und abfangen - get- und set-Methoden . . . 35
7.4 Evolutionstheorie bei Klassen: Vererbung . . . 36
7.4.1 Methoden überschreiben und `@Override` . . . 38
7.4.2 Gemeinsamkeiten erzwingen - abstrakte Klassen . . . 39
7.4.3 Java, wechsel dich: Polymorphie . . . 40
7.5 Interfaces . . . 42
7.5.1 Interface oder abstrakte Klasse? . . . 44

8 Wegweiser in Java: Referenzen **44**
8.1 Verweise ins Leere: Null-Referenz . . . 45
8.2 Postbotentalk: Referenzen vergleichen . . . 45

9 UML - Unified Modeling Language **46**
9.1 Das Klassendiagramm . . . 46
9.1.1 Darstellung von Klassen . . . 47
9.1.2 Verknüpfungen . . . 48
9.2 Vorsicht Kleinkram! . . . 50

10 Die Datenkraken: Collections **50**
10.1 Einführung . . . 50
10.2 Generische Datentypen . . . 51
10.3 Die Warteschlange: Queue . . . 52
10.4 Sardinen in der Dose: Stacks . . . 53

11 Mit Fehlern spielen: Exceptions **55**
11.1 Mit Fehlern um sich werfen: `throw` . . . 56
11.2 Fehlerketten: `throws` . . . 57
11.3 Alles wird gut: mit Fehlern weiterarbeiten . . . 58
11.3.1 Problemebereiche eingrenzen: `try` . . . 58
11.3.2 Der Torwart: Fehler auffangen mit `catch` . . . 58
11.3.3 Endstation: `finally` . . . 59

12 Nachwort **61**

13 Glossar **62**
13.1 Die wichtigsten Schlüsselwörter . . . 62
13.2 Operatoren . . . 63

1 Die Zwischenablage: Variablen

Prinzipiell hat ein Programm die Aufgabe, Daten entgegenzunehmen, zu verarbeiten und wieder auszugeben. Um eure Daten während des Programmablaufs „zwischenzuspeichern“, gibt es **Variablen** und **Konstanten**. Wir sprechen in den folgenden Abschnitten häufig von “speichern“, was aber eigentlich falsch ist und im Moment nur der Vereinfachung dient - später dazu mehr. Fangen wir mit dem einfacheren von beiden an: den Variablen. Hierbei unterscheidet man zwischen **Deklaration** und **Initialisierung**.

1.1 Erstmal Vorstellen: Die Deklaration

Bevor ihr eine Variable verwenden könnt, müsst ihr sie dem Compiler erst mal „vorstellen“. In der Informatik spricht man dann vom **Deklarieren** einer Variable. Bei der Deklaration wird der **Name**, der **Datentyp** und die **Sichtbarkeit** einer Variable festgelegt.

```
1 //Deklaration der Variable "istStudent":
2 private boolean istStudent;

3 //Deklaration der Variable "geburtstag":
4 protected Date geburtstag;
```

Dabei sind `private` und `protected` die jeweilige Sichtbarkeit (siehe Tabelle in Kapitel 6.1) der Variable, `boolean` und `Date` der jeweilige Datentyp und `istStudent` bzw. `geburtstag` der Name der Variable.

Die Sichtbarkeit legt fest, welche Objekte auf die Variable zugreifen dürfen. Der Datentyp legt fest, was für Daten in der Variable „gespeichert” werden dürfen (nämlich nur der angegebene Datentyp und alle von diesem abgeleiteten Datentypen). Der Variablenname schließlich gleicht dem Namen einer Person: eine Variable wird im späteren Programmfluss durch ihren Namen

eindeutig identifiziert, folglich darf ein Variablenname innerhalb eines Sichtbarkeitsbereiches nur einmal existieren! Eine Besonderheit dabei ist die sogenannte **Verdeckung** (siehe Kapitel 7.3.6).

Die Eindeutigkeit von Variablennamen hat auch folgende Konsequenz: Variablen dürfen nicht wie Klassen oder Schlüsselwörter heißen (also z.B. `int String;` oder `boolean continue`). Auch wenn wir hier nicht alle Schlüsselwörter vorstellen, markiert ein Editor mit Syntax Highlighting, wie der EJE, das automatisch.

Nicht alle Datentypen sind gleich, es gibt primitive Datentypen (Kapitel 2) und Referenzdatentypen (Kapitel 8). Sie verhalten sich unterschiedlich!

1.2 Das erste Mal: Initialisierung

Solange eine Variable nur deklariert ist, ist sie dem Compiler zwar bekannt, existiert innerhalb eures Programms aber noch nicht. Damit eine Variable auch tatsächlich existiert, fehlt noch die sogenannte **Initialisierung**: Dies geschieht automatisch, sobald ihr einer Variable einen Wert zuweist. Deshalb bezeichnet man das erstmalige Zuweisen auch als Initialisierung einer Variable.

Wenn ihr später der Variable einen neuen Wert zuweist, ist das keine Initialisierung, sondern nur eine neue Zuweisung![1]

Nehmen wir die bereits oben deklarierten Variablen, sieht das Ganze dann so aus:

[1] Der Unterschied wird besonders bei Referenzdatentypen deutlich.

```
istStudent = false; //Initialisierung mit dem Wert
istStudent = true; //neue Wertzuweisung, aber keine Initialisierung mehr

//Initialisierung mit dem aktuellen Datum:
geburtstag = new Date();

//neue Wertzuweisung, aber keine Initialisierung mehr
geburtstag = new Date(1988,9,28);

//Deklaration & Initialisierung lassen sich auch zusammen ziehen:
private String s = "Hallo Welt!";
```

1.3 Die treuen Gefährten: Konstanten

Konstanten sind nichts anderes als Variablen, deren Wert nach der Initialisierung nicht mehr geändert werden kann. Folglich werden Konstanten auch fast genauso wie Variablen deklariert, der einzige Unterschied ist das zusätzliche Schlüsselwort **final**.

```
public final String vorname; //Deklarieren der Konstanten "Vorname"
vorname = "Roland";    //Initialisierung mit dem String "Roland"

vorname = "Max";
//würde einen Fehler verursachen, da bereits ein Wert zugewiesen wurde:
//Uncompilable source code. Cannot assign a value to final variable
↪ vorname
```

1.4 Die kleinen Unterschiede: Verschiedene Typen von Variablen

Prinzipiell unterscheidet man in Java drei Typen von Variablen:

- **Objektvariable.** Das sind Variablen, die irgendwo direkt innerhalb der Klasse (und nicht innerhalb einer Methode) deklariert und manchmal auch initialisiert werden. Jede Instanz der Klasse (= jedes Objekt mit dem Datentyp der Klasse) besitzt seine eigene, unabhängige Variable.
- **Klassenvariablen.** Ähnlich den Objektvariablen, allerdings besitzen alle Instanzen dieser Klasse zusammen nur eine Variable. Wenn also eine Instanz den Wert einer Variable ändert, ändert er sich auch für alle anderen Instanzen! Zum „Erstellen“ einer Klassenvariable (oder auch **statische Variable**) dient das Schlüsselwort `static`, das einfach bei der Deklaration dazugeschrieben wird.
- **Lokale Variablen.** Hierbei handelt es sich um Variablen, die innerhalb einer Methode deklariert sind. Sie existieren nur innerhalb des sie umschließenden Blocks und werden von Java nach dem Beenden des Blocks aus dem Speicher gelöscht. Bei **lokalen Variablen** darf es zu Verdeckung kommen.

Diese Liste gilt äquivalent für Konstanten.

1.4.1 Lokale Variablen

Es ist an der Zeit, dass ihr euer Wissen über lokale Variablen etwas ausbaut. Dabei handelt es sich um Variablen, die innerhalb von Methoden deklariert werden und dort eventuell nur in speziellen Blöcken vorhanden sind, sprich: wird der Block verlassen, fällt die Variable der Garbage-Collection zum Opfer und ist nicht weiter existent.

```
1  class Beispiel {
2      public void beispielMethode() {
3          int a;    //Deklaration von a im Block "Methode"
4          a = 3;    //Initialisierung von a mit dem Wert 3
5          int b = 4;    //Deklaration und Initialisierung von b

6          for(int i=a; i>1; i--) {
7              b = 4*i;
8              int c = 9;    //Deklaration von c im Sub-Block "Schleife"
9              System.out.println(c);    //Ausgabe wäre 9
10         }

11         System.out.println(a);    //Ausgabe wäre 3
12         System.out.println(b);    //Ausgabe wäre 8
13         System.out.println(c);  /*Hier würde sich der Compiler
14           * beschweren, dass es die Variable c nicht gibt! */
15     }
16 }
```

Wie ihr in dem Beispiel gut erkennen könnt, ist die Variable c, die im Block der `for`-Schleife deklariert wurde, auch nur innerhalb dieses Blocks vorhanden. Das gilt natürlich genauso für andere Typen von Blöcken, wie `while`, `if`, `else`, `try` oder `catch`, auf die wir später noch eingehen werden.

Lokale Variablen sind also nur eine Art „Zwischenspeicher“ innerhalb eines Blockes. Wer mehr braucht, sollte sich die anderen beiden Typen von Variablen anschauen (also Kapitel 1.4)

1.5 Mehrere Variablen auf einem (ordentlichen) Haufen: „Arrays“

Häufig habt ihr den Fall, dass ihr euch mit vielen Objekten des gleichen Typs herumschlagen müsst. Nun wäre es sehr umständlich, wenn wir für jedes dieser Objekte eine eigene Variable deklarieren müssten. Deshalb existieren in Java verschiedene Klassen (die sogenannten **Collections**) und Datenstrukturen, die Objekte sammeln können. Wir möchten euch nun die einfachste dieser Techniken - die **Arrays** - vorstellen.

Wenn ihr schon ein eigenes Java-Programm zum Laufen gebracht habt, habt ihr bereits einen Array verwendet. Erinnert ihr euch an das `String[] args` in der Parameterliste der `main`-Methode? Das ist ein Array[2]! Ein Array wird in Java eigentlich recht simpel angelegt: Bei der Deklaration schreibt ihr hinter den Datentyp einfach *eckige* Klammern (also z.B.: `Date[] alleMeineTermine`), während ihr bei der Initialisierung einfach wieder `[ ]` hinter den Datentyp schreibt. Wichtig ist nur, dass bei der Initialisierung die Größe des Arrays in den eckigen Klammern angegeben wird (also z.B.: `new int[20]`). Die runden Klammern eines Konstruktoraufrufs

[2] Wenn wir ein Java-Programm über die Konsole aufrufen, können wir Parameter übergeben. Diese werden an den Leerzeichen aufgetrennt und der Main-Methode als String-Array übergeben. Siehe 6.6.

© R. Friebel, C. Nitschke, J. Hofinger

machen bei Arrays keinen Sinn, weil ein Array nur eine Sammlung gewisser Elemente darstellt, sprich selber gar keine Attribute (außer seiner Größe und seinem Datentyp) hat. In einem Array werden, analog zu Variablen, also nur die Referenzen (siehe Kapitel 7.4) auf die Objekte gespeichert.

```
int[] array; // Deklaration eines Arrays
array = new int[15]; // Initialisierung mit Größe 15
char[] alphabet = new char[26]; //Kurzversion
```

Ein weiteres Analogon zu Variablen ist auch, dass in Arrays primitiven Datentyps tatsächlich an jeder Position ein Wert steht und bei der Initialisierung auch sofort mit einem Standardwert (also z.B. 0 bei einem `int`-Array) eingerichtet wird.

Einzelne Objekte legt ihr in den Array, indem ihr direkt den **Index** angebt.

```
int[] abc = new int[4]; //einen int-Array deklarieren und initialisieren
abc[0]=4;     //Zahl '4' an 1. Stelle einfügen
abc[1]=11;     //Zahl '11' an 2. Stelle einfügen
abc[3]=2;     //Zahl '2' an 4. Stelle einfügen

abc[4]=11;     //Fehler: "ArrayIndexOutOfBoundsException"

String[] args = new String[5]; //einen String-Array der Größe 5 anlegen
args[0] = new String("test"); //ähnlich bei allen komplexen Datentypen
args[1] = "test";     //Sonderfall bei Strings
```

Ihr könnt jederzeit die Länge eines Arrays ausgeben. Dies geht mit dem Befehl `name.length`. Übrigens handelt es sich hier um einen Integer (siehe Kapitel 2), wodurch sich die (theoretische) maximale Länge eines Arrays zu 2.147.483.647 Werten ergibt. In der Praxis wird man diese Grenze kaum ausreizen, da beispielsweise ein `int`-Array dieser Größe 8,6 GB Arbeitsspeicher belegen würde.

⚠ Wichtig ist: Der Computer fängt stets bei 0 an zu zählen. Somit hat ein Array der Größe 4 nur die Indizes 0, 1, 2 und 3. Der höchste Index ist also `array.length - 1`.

⚠ Wenn ihr auf einen Index außerhalb des Arrays zugreift (also z.B. `abc[-1]` oder `abc[4]`), stürzt euer Programm ab, da die Java Virtual Machine das nicht verarbeiten kann. Genauer gesagt wird eine sogenannte **Exception** geworfen, auf die wir in Kapitel 11 genauer eingehen werden. Der Exception-Typ ist in diesem Fall `ArrayIndexOutOfBoundsException`.

⚠ Wenn ihr auf Elemente eines Arrays zugreift, bevor er initialisiert wurde, wirft die JVM auch eine Exception: eine `NullPointerException`. Zwar hat Java einen Warnmechanismus, der euch schon beim Kompilieren darauf hinweist, aber der funktioniert nur bei lokalen Array-Variablen.

Eine weitere Option, einen Array zu initialisieren, zeigt folgendes Beispiel:

```
int[] antwort = new int[] {6, 9, 42};
//Kurzform:
int[] primzahlen = {2, 3, 5, 7, 11};
```

Der wohl größte Nachteil von Arrays ist, dass sie nicht erweiterbar sind. Ist ein solches Objekt erst einmal mit einer gegebenen Größe initialisiert, lässt sich diese nicht mehr verändern. Also: Bevor sie verwendet werden können, muss man wissen, wie viele Elemente das Array später enthalten soll.

2 Datentypen

Wir haben bis jetzt schon häufig Datentypen verwendet, z.B. `int`, `boolean`, `String`, `Date` oder Arrays - Zeit also, klarzustellen, was ein Datentyp überhaupt ist und wie man ihn verwendet. Streng genommen definiert ein Datentyp die Zusammenfassung einer Objektmenge sowie darauf definierte Methoden.

Man unterscheidet grundsätzlich zwischen den sogenannten **primitiven** und **zusammengesetzten** Datentypen[3].

2.1 Primitive Datentypen

Folgende Datentypen haben wir in Java zur Verfügung:

- `int` (von engl. integer) stellt eine ganze Zahl zur Verfügung, in der maximal 2^{32} Werte gespeichert werden können, also von $-2^{31} = -2.147.483.648$ bis $2^{31} - 1 = 2.147.483.647$. Der Grund für die genannten Werte liegt darin, dass dann das erste Bit der Binärdarstellung dem Vorzeichen entspricht.

- `short` stellt ebenfalls eine ganze Zahl zur Verfügung, die allerdings weniger Speicherplatz braucht. Der mögliche Wertebereich ist -2^{15} bis $2^{15} - 1$, also insgesamt 65.563 Werte.

- `byte` ist ebenfalls eine ganze Zahl, braucht allerdings noch weniger Speicherplatz, nämlich genau ein Byte, also 8 Bit. Somit lassen sich 256 verschiedene Werte speichern.

- `long` ist der vierte Datentyp für ganze Zahlen. Dieser stellt jedoch doppelt so viel Speicherplatz bereit wie ein `int`, nämlich -2^{63} bis $2^{63} - 1$.

- `char` (von engl. character) speichert ein Unicode-kodiertes Zeichen. Wie ein Short sind 2^{16} Werte möglich, der kleinste Wert ist allerdings 0. Variablen diesen Datentyps werden folgendermaßen initialisiert:

```
1 char c;
2 c = 'A';
3 //Gleiches Ergebnis wie Zeile 2:
4 c = 0x41; // 'A' ist in Unicode mit 41 (hex) kodiert
5 // oder:
6 c = 65; // 41 im Hexadezimalsystem ist 65 im Dezimalsystem
```

- `float` (von engl. floating point number) speichert eine **Gleitkommazahl** mit den drei Werten Vorzeichen, Exponent (zur Basis 2) und Mantisse[4]. Wie genau das funktioniert, wird in der Norm IEEE 754 definiert und ist etwas kompliziert. Glücklicherweise macht Java das für uns, intern handelt es sich um eine 32-Bit-Zahl. Der Wertebereich einer solchen Zahl liegt bei $\pm 1,4 \cdot 10^{-45} ... \pm 3,4 \cdot 10^{38}$. Wenn ein Programm sehr genau rechnen muss, sollte man diesen Datentyp nicht verwenden.

  Wollen wir Java dazu zwingen, einen Wert in einer Rechnung als `float` zu interpretieren, benutzt man dafür den Buchstaben `f`. Ein Beispiel dafür wäre `System.out.println(5 / 7f)`. Wieso, erfährst du im nächsten Kapitel, nur so viel: Ohne das `f` würde als Ergebnis 0 ausgegeben werden, da Ganzzahlliterale ohne weitere Angabe als `int` interpretiert werden.

[3] Eigentlich gibt es auch noch Zeigerdatentypen. Aber das lassen wir lieber, dann wirds nämlich wirklich kompliziert.

[4] Die Zahl ist dann: (Vorzeichen Mantisse$\cdot 2^{\text{Exponent}}$).

- `double` speichert eine Gleitkommazahl mit doppelter Genauigkeit, also 64 Bit. Das erweitert den Wertebereich auf $\pm 4,9 \cdot 10^{-324} ... \pm 1,7 \cdot 10^{308}$, was für die meisten Anwendungen reichen sollte. Das Anhängsel für Literale ist ein `d`.

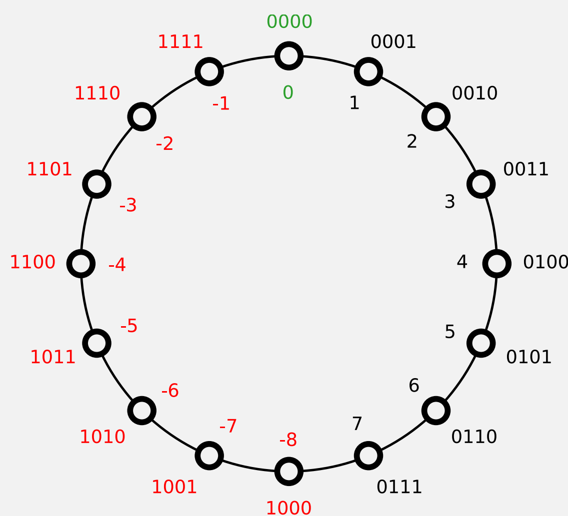
  Trotzdem: Beliebig genau ist auch ein `double` nicht. Das zeigt sich vor allem bei wiederholten Operationen. Addiert man beispielsweise 100 `double`s mit dem Wert 0.01 auf, erhält man nicht - wie zu erwarten wäre - 1.0, sondern 1.0000000000000007. Bei einer Million `double`s erhält man schon 10000.000000171856.

- `boolean` tanzt ein wenig aus der Reihe. Je nach Implementierung der Java Virtual Machine braucht dieser unterschiedlich viel Speicherplatz, er speichert aber einfach einen Wahrheitswert. Benannt ist er nach George Boole, dem Pionier der Wahrheitswertrechnung. Ein `boolean` kann genau zwei verschiedene Werte annehmen: `true` oder `false`, standardmäßig, d.h. wenn ein `boolean`-Array initialisiert wird, ist er auf `false` gesetzt.

Das Zweierkomplement

In der Informatik gibt es mehrere Methoden, negative Zahlen darzustellen. Die Idee, ein Bit einer Zahl als Vorzeichen zu verwenden, ist gut, aber wie legt man dann die anderen Zahlen fest? Heutzutage hat sich eine Darstellung, die Zweierkomplement-Darstellung, durchgesetzt.

Betrachten wir mal eine 4-Bit-Zahl, die 16 verschiedene Werte darstellen kann:



Damit wir gleich viele positive wie negative Zahlen darstellen können, ordnen wir die 1000 den negativen Zahlen zu - da 0000 ja per Definition kein Vorzeichen hat. Dann haben alle negative Zahlen das Vorzeichen 1, woran man sie leicht erkennen kann.

Die Definition hat noch einen Vorteil: Addieren von positiven und negativen Zahlen macht keine Probleme. Rechnet man $-2+3$, addiert man also binär $1110+0011 = 10001$, die führende 1 wird abgeschnitten und so erhält man korrekterweise 1.

So erklärt sich, wieso der Wertebereich der primitiven Datentypen scheinbar mehr negative als positive Zahlen umfasst.

Das Umrechnen zwischen Dezimaldarstellung und Zweierkomplement ist einfach: Für eine negative Zahl (z.B. 1101) alle Bits invertieren (also 0010) und 1 addieren (0011). Das entspricht 3, was korrekt ist.

2.1.1 Überlauf: Ein voller Eimer wird plötzlich leer

Folgender Code wird erst gar nicht kompiliert:

```
byte b = 500;
```

Klar, ein Byte fasst nur Werte bis 127. Aber folgender Code wird sogar fehlerfrei ausgeführt:

```
byte b = 125;
byte ergebnis += 5; // Kurzform für "ergebnis um 5 erhöhen"
```

Das Ergebnis verwundert allerdings: Nach dem Ausführen ist in `ergebnis` der Wert -126 gespeichert. Was ist passiert?

Erklären lässt sich das mit dem sogenannten Y2K-Problem: Grob gesagt haben viele IT-System vor dem Jahr 2000 Jahreszahlen zweistellig gespeichert. Beim Umspringen von 1999 auf 2000 wird in diese Variablen also der Wert 00 gespeichert. Das ist aber etwas unschön, weil das System diese Jahreszahl immer noch mit 19 am Anfang interpretiert. Also wird der 1.1.2000 als 1.1.1900 angezeigt. Es wurde sogar befürchtet, dass dieser Fehler in so vielen Programmen enthalten ist, dass am 1. Januar 2000 die weltweite digitale Infrastruktur zusammenbricht. Natürlich war das übertrieben. Aber darauf aufpassen sollte man trotzdem!

Also, was ist in unserem Beispiel passiert? Führen wir die Addition einmal binär aus:

$$
\begin{array}{r|cccccccc}
 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & {\scriptstyle 1} & \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Soweit ist noch alles richtig. Aber wir haben ja gesagt, dass das erste Bit eines `byte` das Vorzeichen ist - hier steht also die zweite negative Zahl, was bei einem `char` nun mal -126 ist. Also: Aufpassen!

Überlauf in hardwarenahen Sprachen

Wie geht ein Prozessor mit einem Überlauf um? Die heutigen RISC-Prozessoren haben in ihrem Akkumulator (der Recheneinheit) sogenannte Flags. Diese indizieren z.B., ob das Ergebnis 0 ist (das `zero`-Flag), was der Übertrag der aktuellen Operation ist (das `carry`-Flag) und ob ein Überlauf stattgefunden (das `overflow`-Flag).

Auf Assemblerebene kann man so sehr einfach überprüfen, ob das Bit gesetzt wurde, da der Prozessor entsprechende Befehle für Flag-Vergleiche bereithält.

In höheren Programmiersprachen wie Java sind arithmetische Operationen so definiert, dass dabei per Definition kein Überlauf entstehen kann. Daher muss man sich darum selbst kümmern.

2.2 Umwandlung von Datentypen

Betrachten wir folgenden Code-Schnipsel:

```
int i = 2;
double d = 2.5d;
System.out.println(i + d);
```

Wir addieren hier zwei verschiedene Datentypen, wobei einer 32 Bit lang ist und der andere 64 Bit. Erinnerung: Ein `double` besteht eigentlich aus zwei Zahlen, Mantisse und Exponent. Würden wir beide Zahlen einfach binär addieren, würde ziemlicher Unfug herauskommen - gut, dass Java das vermeidet.

In manchen Fällen erledigt Java die notwendige **Typumwandlung** selbst, manchmal musst du dich auch selbst drum kümmern. [5]

2.2.1 Automatische (implizite) Typumwandlung

Wenn bei einer Operation ein Objekt falschen Datentyps angegeben wird, versucht Java zunächst, diesen selbstständig umzuwandeln.

Das geschieht immer dann, wenn dabei keine Genauigkeit verloren geht. Man spricht in diesem Fall von sog. **werterhaltender Typumwandlung**.

Also: Ein `short` wird immer automatisch in Datentypen höherer Genauigkeit, wie `int` oder `double`, umgewandelt. Ein `double` wird dagegen nie automatisch umgewandelt. Wenden wir das auf das folgende Beispiel an:

```
1 byte b = 2;
2 int i = 1000;
3 double ergebnis = b * i;
```

Zunächst wird `b * i` ausgeführt. Der Operator `*` erwartet zwei Objekte gleichen Datentyps, daher wird `b` automatisch in einen `int` konvertiert, da `int` genauer als `byte` ist. Das Ergebnis dieser Rechnung ist also ein `int`. Dieses soll nun in einem `double` gespeichert werden. Auch diese Umwandlung geschieht automatisch.

Folgendes Beispiel würde hingegen als syntaktisch falsch erkannt werden:

```
1 double b = 2;
2 int i = 1000;
3 int ergebnis = i / b;
```

Intuitiv erscheint dies unlogisch, ist doch das Ergebnis (500) eine ganze Zahl. Doch genauer betrachtet ist das Ergebnis von i/b ein `double`, der in einem `int` gespeichert werden soll - aber wie gesagt, Java „verschenkt“ keine Genauigkeit, außer im Code ist etwas anderes angegeben. Das bringt uns zur expliziten Typumwandlung.

2.2.2 Explizite Typumwandlung: Den Computer zu seinem Glück zwingen

Hierfür führen wir die sogenannten **type casts** ein.

Ein Typecast - kurz cast - zwingt ein Objekt in die in Klammern angegebene Form.

Wie sieht so was aus? Betrachten wir das vorherige Beispiel, diesmal korrekt:

```
1 double b = 2;
2 int i = 1000;
3 int ergebnis = (int)(i / b);
```

[5] Hinweis: Die folgenden beiden Unterkapitel beziehen sich lediglich auf primitive Datentypen, für zusammengesetzte Datentypen gilt es aber in analoger Weise.

Der Typecast `(int)` wandelt das Ergebnis - unabhängig von dessen Datentyp - in einen `int` um[6]. Dann wird das umgewandelte Ergebnis in einen `int` gespeichert - also passt alles.

Eine kleine Änderung:

```
double b = 2;
int i = 1000;
int ergebnis = (float)(i / b);
```

Dieses Beispiel funktioniert natürlich nicht. Das Ergebnis wird in einen `float` umgewandelt und soll in einen `int` gespeichert werden - da weigert sich Java also zurecht. Daher nennt man die Typumwandlung in Java auch **checked** - bevor umgewandelt wird, wird überprüft, ob der umgewandelte Datentyp überhaupt verarbeitet werden kann. In manchen Programmiersprachen ist das anders.

Bei Casts verliert man meistens Genauigkeit. Folgende Beispiele sollen das illustrieren:

- Bei `int -> byte` werden die überschüssigen Bits abgeschnitten. Nach dem Aufruf von `byte b = (byte)256;` ist in `b` der Wert 0 gespeichert, da die letzten 8 Ziffern von 256 in Binärdarstellung allesamt 0 sind.
- Bei `double -> int` werden die Nachkommastellen abgeschnitten. Es wird nicht gerundet!
- Bei `double -> float` wird die Zahl auf `float`-Genauigkeit gerundet.

2.3 Zusammengesetzte Datentypen

Zwei zusammengesetzte Datentypen haben wir schon kennengelernt: Arrays und Zeichenketten (`String`s). Das klingt erst mal seltsam, denn wieso sollte ein Zeichenkette aus mehreren anderen Objekten bestehen? Folgender Satz ist sogar noch verwirrender: Eine Zeichenkette ist auch ein Array.

- `String` Eine Zeichenkette besteht zunächst einmal aus einer endlichen Anzahl `char`s. Außerdem kann man auf die Länge eines Strings mit `mystring.length()` zugreifen. Das legt nahe, dass ein `String` eigentlich so etwas ist: `char[]`. Tatsächlich kann man sich das so vorstellen (im Hintergrund passiert noch ein wenig mehr), erklärt aber auch, wieso es nicht ganz trivial ist, zwei Zeichenketten zu verbinden (Erinnerung: Arrays haben eine feste Größe).

  Daraus ergibt sich auch, dass der maximale Wert von `length` gleich dem Maximum eines `int` ist. Das sollte im Normalfall reichen, denn ein String mit 2^{31} Zeichen wäre auf DIN A4 gedruckt ein etwa 25 Meter hoher Papierstapel.

  Stringliterale werden durch Anführungszeichen gekennzeichnet. Also beispielsweise:

  ```
  String s = "Hallo Welt!";
  ```

  Bis jetzt haben wir nur Eigenschaften von `String`s betrachtet, die dafür sprechen, dass ein String nicht primitiv ist. Es gibt jedoch auch Gründe, die dagegen sprechen:

  – Man kann auf Strings binäre Operatoren wie `+` anwenden. So kann man primitive Datentypen recht einfach in Strings konvertieren: durch **Konkatenation** (= Verbinden zweier Zeichenketten, der komplizierte Name für die `+`-Operation im `String`-Raum) mit dem leeren Stringliteral `""`.

[6] Natürlich nur, wenn Java weiß, wie das geht. Bei eigenen Klassen müsste man eine solche Konversion selbst implementieren.

```
1 int i = 15;
2 String s = "" + i;
```

– Weist man eine `String`-Variable eine andere zu, wird keine Referenz kopiert, sondern eine echte Kopie angelegt.

Ein String ist somit irgendwo zwischen beiden Kategorien.

- Arrays. Eindimensionale Arrays haben wir schon in Kapitel 1.5 behandelt, mehrdimensionale Arrays sind natürlich auch zusammengesetzt.
- Klassen. Die zweite Hälfte dieses Kompendiums widmet sich diesem Thema. Daher werden wir hier nicht vorgreifen. Ein Beispiel hatten wir hier schon: `Date` speichert u.a. die Ganzzahlen Jahr, Monat und Tag.

3 Fallunterscheidungen mit `if` und `else`

Ihr kennt das sicherlich schon: Ihr seid auf dem Weg zu einem Hörsaal und steht plötzlich an einer Weggabelung und müsst euch entscheiden, wo ihr jetzt lang geht. So etwas gibt es natürlich in einem Programmfluss auch und es liegt an euch, diese „Gabelungen“ für den Computer lesbar in

euren Code einzubauen. Das Ganze geschieht mit `if`, `else` und `else if`. Fallunterscheidungen sind keine Schleifen - sie sehen sich nur recht ähnlich.

3.1 Bedingung abfragen mit `if`

`if` ist eines der am leichtesten zu verstehenden Kommandos in einer Programmiersprache. Euer Englisch sollte gut genug sein, um zu wissen, dass „if“ auf deutsch „falls“ bedeutet und genau so wird es in Java (und in vielen anderen Programmiersprachen) auch verwendet:

Falls etwas zutrifft, tue etwas Bestimmtes.

In Java funktioniert das ähnlich wie in der gesprochenen Sprache, nur die Syntax unterscheidet sich etwas:

```
1 if (true) {
2     System.out.println("Hello world!");
3 }
```

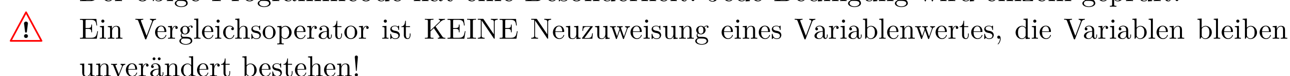
Wie ihr sehen könnt, ist der Aufbau recht einfach: `if` wird von einer Klammer gefolgt, in der die Bedingung steht. Diese Bedingung **muss** ein boolescher Wert, also `true` oder `false`, sein! Ihr könnt also z.B. eine Variable vom Typ `boolean` oder eine Methode, die einen booleschen Wert zurück gibt, in die Bedingung schreiben. [7]
Wenn ihr Variablen oder Rückgabewerte von Funktionen anderer **primitiver** Datentypen habt, helfen euch so genannte Vergleichsoperatoren weiter (`==` (ist gleich), `!=` (ungleich), `<` (kleiner), `>` (größer), `<=` (kleiner gleich), `>=` (größer gleich)).

Der Vergleichsoperator `==` ist nicht mit dem Zuweisungsoperator `=` zu verwechseln!

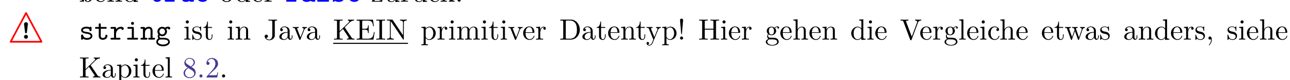
[7] Direkt `true`/`false` in die Bedingung zu schreiben macht nur dann Sinn, wenn man grade auf Fehlersuche in seinem Programmcode ist und einen bestimmten Block weglassen oder ausführen will, um zu sehen, wie das Programm dann weiter läuft.

```java
int a = 5;
int b = 7;

if(a == b) {
    // 5 == 7 ist eine unwahre Aussage,
    //also wird dieser Block nicht ausgeführt
}

if(a != b) {
    //5 und 7 sind versch. Zahlen, also wird dieser Block ausgeführt
}

if(a < 8) {
    // 5 < 8, also wird dieser Block ausgeführt
}
```

Der obige Programmcode hat eine Besonderheit: Jede Bedingung wird einzeln geprüft!

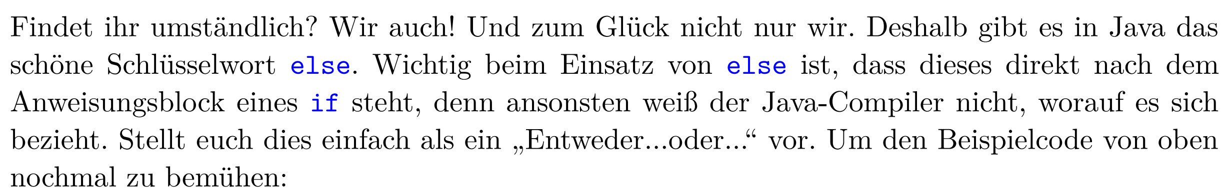

Ein Vergleichsoperator ist KEINE Neuzuweisung eines Variablenwertes, die Variablen bleiben unverändert bestehen!

Vergleichsoperatoren, führen, wie der Name schon sagt, einen Vergleich aus und liefern anschließend `true` oder `false` zurück.


`string` ist in Java KEIN primitiver Datentyp! Hier gehen die Vergleiche etwas anders, siehe Kapitel 8.2.

3.2 Alternativfälle mit `else`

```java
int x = 5;
if (x > 6) {
    System.out.println("Der Wert ist größer als 6!");
}
if (x <= 6) {
    System.out.println("Der Wert ist kleiner oder gleich 6!");
}
```


Findet ihr umständlich? Wir auch! Und zum Glück nicht nur wir. Deshalb gibt es in Java das schöne Schlüsselwort `else`. Wichtig beim Einsatz von `else` ist, dass dieses direkt nach dem Anweisungsblock eines `if` steht, denn ansonsten weiß der Java-Compiler nicht, worauf es sich bezieht. Stellt euch dies einfach als ein „Entweder...oder...“ vor. Um den Beispielcode von oben nochmal zu bemühen:

```java
int x = 5;
if (x > 6) {
    System.out.println("Der Wert ist größer als 6!");
}
else {
    System.out.println("Der Wert ist kleiner oder gleich 6!");
}
```

Schon etwas einfacher, oder?

Also ganz einfach gesagt: `else` wird immer dann ausgeführt, wenn das `if` davor nicht ausge-

führt wurde. So erspart euch `else`, euch die Gegenbedingung zu überlegen und somit auch eine potentielle Fehlerquelle.

3.3 Kompliziertere Strukturen mit `else if`

Manchmal habt ihr Fälle, bei denen ihr nicht nur eine reine „Entweder...oder...“ - Situation habt. Sprich: Nur wenn eine Bedingung falsch ist, soll die nächste geprüft werden. Natürlich könntet ihr lauter `if` untereinander schreiben, aber dann müsst ihr bei den Bedingungen höllisch gut aufpassen, aber das wird bei komplizierten Bedingungen schnell noch komplizierter (und dabei unterstützt euch keine IDE[8] der Welt!). Folglich ist es logisch und etwas effizienter, so etwas zu schreiben:

```
1  int x = 5;
2  if (x > 6) {
3      System.out.println("Der Wert ist größer als 6!");
4  }
5  else {
6      if (x < 6) {
7          System.out.println("Der Wert ist kleiner als 6!");
8      }
9      else {
10         System.out.println("Der Wert ist gleich 6!");
11     }
12 }
```

Das ist nach immer noch etwas arg umständlich zu schreiben und deshalb existiert in Java eine Kurzform, die uns das Leben ein wenig erleichtert: `else if`.

```
1  int x = 5;
2  if (x > 6) {
3      System.out.println("Der Wert ist größer als 6!");
4  }
5  else if (x < 6){
6      System.out.println("Der Wert ist kleiner als 6!");
7  }
8  else {
9      System.out.println("Der Wert ist gleich 6!");
10 }
```

Schon übersichtlicher, oder?

`else if` benötigt also auch eine Bedingung vom Typ `boolean`, also helfen wieder die Vergleichsoperatoren!

Kurzversion: Der ternäre Operator `?:`

Ternär ist eine Operation dann, wenn sie 3 Eingabeparameter hat. Da diese Art der Fallunterscheidung die einzige solche Operation in Java ist, spricht man oft auch von

[8] *IDE* steht für „*integrated development environment*“

„dem ternären Operator“.

Schauen wir uns mal ein Beispiel an:

```
int x = 0; // tue irgendetwas mit x
int y;
if (x < 5) {
    y = 17;
}
else {
    y = -12;
}
```

Die Kurzversion mit dem ternären Operator geht so:

```
int x = 0; // tue irgendetwas mit x
int y = (x < 5) ? 17 : -12;
```

Die allgemeine Form sieht so aus:

```
Bedingung ? Ausdruck1 : Ausdruck2
```

Wenn die `Bedingung` wahr ist, wird `Ausdruck1` zurückgegeben, sonst `Ausdruck2`.

Ist also gar nicht so kompliziert...

Übrigens kann man diese Operatoren auch verschachteln:

```
int x = 0; // tue irgendetwas mit x
int y = (x < 5) ? (x < 2 ? 0 : 1) : (x < 7 ? 2 : 3);
```

Ist $x < 2$, wird in y der Wert 0 gespeichert. Für $x \in [3, 5]$ wird $y := 1$ gesetzt. Für $x = 6$ wird $y := 1$ gesetzt und für alle größeren Werte als 6 wird $y := 3$. Aber mal ehrlich: Das wird schnell kompliziert. Für so etwas bietet sich dann eher folgendes Schlüsselwort an.

3.4 Fallunterscheidungen mit `switch`

Schauen wir uns folgendes Beispiel an:

```
int x = 0; // tue irgendetwas mit x
String name;
if (x == 1) {
    name = "Eins";
}
else if (x == 2) {
    name = "Zwei";
}
else if (x == 3) {
    name = "Drei";
}
else {
```

```
13      name = "Unbekannt";
14  }
```

Für derartige Aufgaben bietet Java einen sogenannten `switch` (von engl. Schalter).

Der Aufbau ist denkbar einfach: Zunächst geben wir eine Variable (oder allgemein Ausdruck) an, anhand der „geschaltet“ werden soll. Diese muss ein primitiver, ganzzahliger Datentyp sein[9]. Dann folgt der Rumpf, in dem die einzelnen Fälle definiert sind: mit dem Schlüsselwort `case` gefolgt von weitere Anweisungen.

⚠ Die JVM sucht sich nun den passenden Fall und springt dorthin, führt aber alle folgenden Fälle ebenfalls aus![10]

Obiges Beispiel als `switch`:

```
1   int x = 0; // tue irgendetwas mit x
2   String name;
3   switch(x) {
4       case 1: name = "Eins";
5               break;
6       case 2: name = "Zwei";
7               break;
8       case 3: name = "Drei";
9               break;
10      default: name = "Unbekannt";
11  }
```

Ohne die `break`s wäre `name` danach in jedem Fall `"Unbekannt"`. Daher wird mit `break;` der aktuelle Block (also der `switch`) verlassen!

Eine Besonderheit ist das letzte Schlüsselwort, `default`: Sollte keiner der davor genannten Fälle zutreffen, springt das Programm zu dieser Stelle.

4 Und täglich grüßt das Murmeltier: Schleifen

Bei der Entwicklung von Programmen steht man oft vor dem Problem, dass eine bestimmte Anweisung wiederholt ausgeführt werden muss.[11] Und um dem Ganzen die richtige Würze zu geben: Meist weiß man zur Compilezeit noch nicht einmal, wie oft überhaupt.

Doch für dieses Problem gibt es in der Informatik eine recht einfache Lösung: Schleifen. Diese bestehen prinzipiell aus Bedingung und Anweisungsblock.

Es gibt drei verschiedene Typen von Schleifen, auf die ich im Folgenden kurz eingehen will.

⚠ Aber bei allen drei Schleifentypen gilt: Vorsicht vor Endlosschleifen!

4.1 Einfache Schleifen mit `while`

Die **while**-Schleife ist wohl die simpelste und am einfachsten zu verstehende Schleife. Sie besteht aus einer einfachen Bedingung und dem Anweisungsblock, der so lange abgearbeitet wird, wie die

[9] Seit Java / sind auch `String`s erlaubt. Diese Neuerung brauchte ganze 16 Jahre, bis sie umgesetzt war...
[10] In manchen Fällen kann das ganz praktisch sein, z.B. wenn man Listen mit Werten befüllen will.
[11] Ihr könnt eine Schleife auch so schreiben, dass sich die Anweisungen mit jedem Durchlauf leicht ändern.

Bedingung noch erfüllt ist. Es gilt die Reihenfolge: Bedingung prüfen → Anweisungen ausführen → Bedingung prüfen usw. Daher nennt man diesen Typ einer Schleife auch **kopfgesteuerte Schleife**.

Hinweis: `i++` ist übrigens eine Kurzform von `i=i+1`.[12]

```
int maxzahl=10;
int i = 1;
while (i <= maxzahl) {
    System.out.print("Die Zahl " + i + " ist ");
    if(i % 2 == 0) {
        System.out.println("gerade");
    }
    else if (i % 2 != 0) {
        System.out.println("ungerade");
    }
    i++;    //i um 1 erhöhen
}
```

Diese Schleife wird insgesamt 10 mal ausgeführt.

`%` ist der sogenannte **Modulo-Operator** und macht nichts anderes als den Rest einer Division zurückzugeben. Geschrieben wird das Ganze `a % b`, wobei `a` durch `b` geteilt würde. Sei `a = 15` und `b = 6`, wäre der Ausdruck `a % b` gleich 3 (da $15/6 = 2$ Rest 3).

4.2 Für die Langsamen: `do-while`-Schleifen

Die `do-while`-Schleife ist ein ziemlicher Exot, da sich meist das gleiche Ergebnis mit einer `while`-Schleife erreichen lässt, wenn man die Bedingung leicht anpasst. Im Prinzip macht sie das Gleiche wie eine gewöhnliche `while`-Schleife. Der Unterschied liegt nur in der Reihenfolge: Die `do-while`-Schleife arbeitet erst den Anweisungsblock ab und prüft dann, ob die Bedingung noch erfüllt ist. Also: Anweisungen ausführen → Bedingungen prüfen → Anweisungen ausführen usw. Daher spricht man auch von einer **fußgesteuerten Schleife**

Das heißt aber auch, dass die `do-while`-Schleife auf jeden Fall einmal den Anweisungsblock ausführt!

Es gibt natürlich Situationen, in denen eine solche Schleife programmiertechnisch recht praktisch sein kann:

```
int gebjahr;
do {
    gebjahr = IOTools.readInt("Bitte geben Sie ihr Geburtsjahr ein: ");
} while (gebjahr > 2015);
```

4.3 Fortschritt ahoi: `for`-Schleifen

Ihr habt es sicherlich schon bemerkt: Es ist ein wenig nervig, erst eine Zählvariable zu deklarieren und initialisieren und dann die Schleife zu schreiben. Die Variable hochzuzählen darf man natürlich auch nicht vergessen, sonst hätte man eine Endlosschleife. Und übersichtlich ist es

[12] Wo genau der Unterschied zwischen `i++` und `++i` liegt, ist für euch erstmal nicht relevant.

sowieso nicht. Deshalb kann man in das Argument der `for`-Schleife die Zählvariable gleich mit zu der Bedingung dazuschreiben:

```
int[] array = {2, 3, 5, 7, 11};
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

Es werden in diesem Beispiel alle Werte des Array ausgegeben. Warum? Eine `for`-Schleife wird wie folgt abgearbeitet: Erste Anweisung im Argument ausführen (hier also eine `int`-Variable deklarieren und initialisieren) → Prüfen, ob die Bedingung erfüllt ist → Block ausführen → Zählvariable verändern (oder allgemein die dritte Anweisung ausführen) → Erneut Bedingung prüfen. Da die Indizes in einem Array von 0 bis `array.length - 1` gehen, werden alle diese Elemente abgerufen.

Und noch eine: die „for-each"-Schleife

Hierbei handelt es sich um eine spezielle Version der `for`-Schleife, die sich in Kombination mit Arrays oder **Collections** anbietet. Was Collections sind und wofür man die braucht, lernt ihr in Kapitel 10. Das folgende Beispiel ist deshalb auch nur der Vollständigkeit halber in diesem Kapitel dabei. Also nicht wundern, wenn ihr noch nicht alles versteht!

```
// Erzeuge eine Collection aus Integern
List<Integer> sammlung = new ArrayList<Integer>();

// Befülle diese Liste:
sammlung.add(5);

System.out.println("Die Collection enthält folgende Werte");
// die for-each-Schleife
for (int aktwert : sammlung) {
    System.out.println(aktwert);
}
```

Diese Schleife würde jetzt *nacheinander* alle Zahlen in der Liste `sammlung` durchgehen und ausgeben. Wir müssen uns also nicht um Zählvariablen kümmern, und somit auch keine Sorgen machen, dass wir auf Listenelemente zugreifen, die nicht existieren.

4.4 Sprunganweisungen

4.4.1 Das war's: `break`

Möchte man eine Schleife - oder allgemein einen Block - vorzeitig verlassen, kann man das Schlüsselwort `break` verwenden.

Dieser Code beschreibt einen (sehr ineffizienten) Primzahltest:

```
int aktZahl = 57;
boolean istPrim = true;
for (int teiler = 2; teiler < aktZahl; teiler++) {
```

```
4       if (aktZahl % teiler == 0) {
5           istPrim = false;
6           break;
7       }
8   }
9   System.out.println(istPrim);
```

Unsere Eingabe `aktZahl` wird durch jede Zahl zwischen 1 und sich selbst geteilt. Wenn dabei der Rest 0 herauskommt, wird das `flag` auf `false` gesetzt (da die Zahl dann keine Primzahl ist). Ohne das Schlüsselwort `break` würde die Schleife trotzdem fertig laufen - das ist aber nicht gewünscht, da wir bereits einen Teiler der Zahl gefunden haben und sie somit nicht mehr prim sein kann.

4.4.2 So wird's nix: `continue`

Mit dem Schlüsselwort `continue` kann man zur nächsten Iteration der Schleife fortfahren. Folgender Code gibt beispielsweise alle geraden Zahlen kleiner 10 aus:

```
1 for (int i = 1; i < 10; i++) {
2     if (i % 2 != 0) continue;
3     System.out.println(i);
4 }
```

`continue` springt also zur nächsten Iteration der Schleife. Das spart Rechenzeit, wenn man z.B. eine komplizierte Rechnung hat, die von einer Variablen abhängt, und schon früh sieht, dass für diesen Wert nicht das gewünschte Ergebnis herauskommt.

Aber: Eigentlich sind sowohl `continue` als auch `break` nicht ganz sauberer Stil. Denn Sprunganweisungen machen den Code eigentlich immer komplizierter und schwerer zu lesen, auch wenn es vielleicht kürzer ist. Aber selbst das ist nicht immer der Fall, denn das obige Beispiel lässt sich z.B. so umformulieren:

```
1 for (int i = 1; i < 10; i += 2) {
2     System.out.println(i);
3 }
```

Also: Vermeide Sprunganweisungen, wo immer möglich!

5 Bedingungen verknüpfen mithilfe Boolescher Operatoren

Manchmal habt ihr den Fall, dass mehrere Bedingungen zutreffen müssen oder können, damit etwas ausgeführt werden soll. Dafür gibt es die sogenannten booleschen Operatoren:

- `&&` - Beide Bedingungen müssen zutreffen (logisches **und**)
- `||` - Mindestens eine der Bedingungen muss erfüllt sein (logisches **oder**)
- `^` - Genau eine Bedingung muss erfüllt sein. Entspricht dem umgangssprachlichen „entweder ... oder“ und heißt auch **Exklusives Oder**.
- `!` - negiert einen Wahrheitswert

Ein Beispiel als Code:

```
int i = 5, j = 17;

boolean b1 = (i < j && i > 0);  // true, da 5 < 17 und 5 > 0
boolean b2 = (i < j && j == 0); // false, da 17 nicht 0 ist

boolean b3 = (i < j || j == 0); // true, da 5 < 17
boolean b4 = (i < j || j > 5);  // true, da beide Bed. erfüllt
boolean b5 = (j < 8 || i == 0); // false, da keine Bed. erfüllt

boolean b6 = !(i < j);          // false, da 5 < 17 true ist
boolean b7 = (!b3 || b4);       // true, da b4 true ist
```

Die Klammern kann man übrigens weglassen (außer im 6. Beispiel), die sind nur zur Veranschaulichung da.

Diese Operatoren kann man gut bei `if`-Statements einsetzen:

```
if (5 < 7 && 9 == 8 + 1) {
    // Bedingung ist wahr, wird also ausgeführt
}
```

Das war einfach, oder?

6 Etwas erledigen lassen: Methoden

Jetzt habt ihr gelernt, wie ihr Bedingungen und Schleifen schreibt, aber noch nicht, *wo* ihr sie überhaupt einbauen könnt. Das geht nämlich nur innerhalb bestimmter Bereiche: innerhalb der Methoden.[13]

Eine Methode besteht aus zwei Teilen: Dem Methodenkopf und dem Methodenrumpf.

Der Methodenrumpf ist einfach all das, was ihr in die Methode schreibt und was später ausgeführt werden soll. **Wichtig:** die Klammern `{ }` gehören dazu!

```
{
    //tu etwas in diesem Methodenrumpf
}
```

Ein klein wenig schwieriger ist der Methodenkopf. Dieser besteht aus:

(mehrere) Modifizierer + Rückgabe-Datentyp + Methodenname + Parameterliste

Also z.B.

```
private static int Addiere(int a, int b) {
    int ergebnis = a + b;
    return ergebnis;
}
```

[13] Ok, auch innerhalb von Konstruktoren, aber das sind im Prinzip auch nur spezielle Methoden!

Die **Methodensignatur** wird aus dem Methodenkopf erstellt, anhand ihr unterscheidet der Java-Interpreter die Methoden einer Klasse. Folglich darf jede Signatur nur einmal in einer Klasse vorkommen! Die Signatur setzt sich aus folgenden Teilen zusammen:

Methodenname + Datentypen der Übergabeparameter und ihre Reihenfolge

Die folgenden Methoden haben also gleiche Signaturen:

```
private double Berechne(int a, double b) { ... }

private int Berechne(int wert1, double wert2) { ... }

public static int Berechne(int zahl, double nochEineZahl) { ... }
```

Die folgenden Methoden haben unterschiedliche Signaturen:

```
private double Berechne(int a, double b) { ... }

private double Berechne(double b, int a) { ... }

private double Berechne(int a, double b, boolean c) { ... }
```

6.1 Sichtbarkeit

Es gibt in Java vier **Zugriffsmodifizierer**: `public`, `<default>` (`<default>` erhält man nur dann, falls kein anderer Modifizierer angegeben wird!), `protected` und `private`.
Eine kurze Übersicht über die Wirkung der einzelnen Schlüsselworte bietet diese Tabelle:

Modifier	in der Klasse selbst	Paket-Klassen	Unterklassen	sonstige andere Klassen
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	-
keine Angabe	✓	✓	-	-
`private`	✓	-	-	-

Aus Sicherheitsgründen gilt bei der Sichtbarkeit die Faustregel: „Immer nur so sichtbar wie nötig!“. Das ist z.B. bei Variablen wichtig: Stellt euch vor, ihr programmiert eine Klasse mit öffentlich zugänglichen Variablen (`public int var`) und stellt nur die kompilierte Klasse einem anderen Programmierer zur Verfügung. Dieser greift jetzt direkt auf eure Variablen zu und das Programm funktioniert erst mal, wie es soll. Nun kommt ihr daher und überarbeitet eure Klasse. Eventuell gibt es dann noch die vorher definierte Variable, diese hat aber inzwischen eine ganze andere Bedeutung (z.B.: `public String var`). Ihr könnt euch eventuell vorstellen, dass das zu Fehlern führen würde. Deshalb: Gerade Variablen, aber auch Methoden, möglichst immer verstecken!

```
public int x; //überall sichtbar
protected int y; //eingeschränkt sichtbar
int z; //default-Sichtbarkeit
private int w; //nur in der aktuellen Klasse sichtbar
```

Man sieht, dass diese Zugriffsmodifizierer nicht nur für Methoden, sondern auch für Variablen gelten. Sie können allerdings nicht bei lokalen Variablen verwendet werden, das würde auch wenig Sinn machen.

6.2 Rückgabewert

Eine Methode, die Daten verarbeitet, kann an denjenigen, der sie aufruft (die Methode startet) auch etwas zurückgeben. Kann, muss aber nicht. Falls nichts zurückgegeben wird, müsst ihr das trotzdem dem Computer sagen. Dazu dient in Java das Schlüsselwort `void` (engl. leer). Ansonsten steht als Rückgabetyp gerade der Datentyp, der auch wirklich zurückgeliefert wird. Wird ein Rückgabewert im Methodenkopf definiert, erwartet der Compiler das Schlüsselwort **return** gefolgt von dem zurückzugebenden Wert bzw. einer Variable des entsprechenden Typs.

```
1 public void beispielMethode1() {
2     //tue einfach nur was und gib nichts zurück
3 }

4 public int beispielMethode2() {
5     int x = 5;
6     //tue irgendwas mit x
7     return x;
8 }
```

Ihr könnt `return` auch in Methoden mit `void` als Rückgabewert verwenden. In diesem Fall wird die Methode beim Auftreten von `return;` einfach vorzeitig verlassen.

6.3 Methodennamen

Endlich mal etwas, bei dem wir als Programmierer (fast) keinen Einschränkungen unterliegen. Wichtig ist nur, dass der Methodenname mit einem Buchstaben beginnt und keine Leerzeichen oder Sonderzeichen enthält. Des weiteren hat es sich etabliert, das erste Wort komplett klein zu schreiben und jedes weitere Wort mit einem Großbuchstaben zu beginnen (z.B: `meineKleineBeispielMethode()`). Natürlich ist es auch sinnvoll, wenn der Name der Methode erkennen lässt, was die Methode macht und am besten so gewählt ist, dass jemand, der euren kompilierten Code verwendet, auch schon ohne Dokumentation erahnen kann, wie die Methode verwendet wird.

6.4 Methodenparameter

Häufig wollt ihr, dass die Methode nicht nur irgendetwas tut, sondern, dass sie etwas Spezielles mit Werten macht, die ihr ihr übergebt. Damit das funktioniert, müsst ihr beim Definieren der Methode in die Klammern `( )` die Typen der Variablen schreiben, die ihr später der Methode übergeben wollt. Damit ihr auf diese **Methodenparameter** später auch zugreifen könnt, müsst ihr den Variablen auch Namen geben. Das Ganze sieht dann aus wie eine normale Deklaration einer Variablen mit der Ausnahme, dass Java für euch das Initialisieren beim Aufrufen der Methode übernimmt:

```
1 public class Beispiel {
2     public static void wasAusgeben(String vorname, String name,
3     int alter) {
4         //Zur Bedeutung des static siehe Kapitel 6.5
5         System.out.println("Name: " + name);
6         System.out.println("Vorname: " + vorname);
7         System.out.println("Alter: " + alter);
```

```
8       }
9       public static void main(String[] args) {
10          //hier beginnt das Programm zu laufen

11          //1.
12          wasAusgeben("Max", "Mustermann", 23);
13          //wir übergeben hier der Methode 2 Strings und eine ganze Zahl

14          //2.
15          wasAusgeben("Mustermann", "Max", 21);
16          //auch das geht, hat aber ein anderes Ergebnis (s.u.)

17          //3.
18          wasAusgeben("Max", "Mustermann");
19          /*das würde einen Fehler verursachen,
20          * weil eine Methode mit dieser Signatur nicht existiert*/
21      }
22  }
```

Das Ergebnis sähe für die Fälle 1 & 2 so aus:

```
1 (für die 1. Methode):
2 Name: Mustermann
3 Vorname: Max
4 Alter: 23

5 (für die 2. Methode):
6 Name: Max
7 Vorname: Mustermann
8 Alter: 21
```

Wie ihr seht, ist es Java egal, in welcher Reihenfolge die Parameter übergeben werden, wichtig ist nur, dass die Reihenfolge der Datentypen - in diesem Fall `String`-`String`-`int` exakt eingehalten wird! Was die Methode damit macht, ist aber natürlich von der Abfolge der Parameter abhängig, und nicht nur von der Abfolge der Datentypen.

In unserem Beispiel sind die Parameter `vorname` und `alter` sogenannte **formale Parmeter**, die **aktuellen Parameter** beim Methodenaufruf sind z.B. `"Max"` und 23.

Selbstverständlich können Methoden auch keine formalen Parameter haben. Dann lässt man die Klammern einfach leer:

```
1 private int ParameterloseMethode(){
2     return 0;
3 }
```

6.5 Weitere Schlüsselwörter

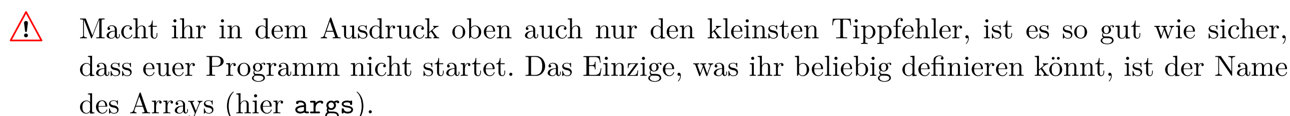
Für Methoden gibt es noch einige Schlüsselwörter, die ihr kennen solltet und die wir euch hier in aller Kürze vorstellen will:

- `abstract`: die Methode muss bei einer Ableitung der Klasse überschrieben werden. Hierdurch wird die Klasse automatisch auch abstrakt und muss ebenfalls mit `abstract` gekennzeichnet werden (mehr dazu im Kapitel „Vererbung”)
- `final`: die Methode darf bei einer Ableitung der Klasse nicht überschrieben werden (mehr dazu im Kapitel Vererbung)
- `static`: statische Methoden sind objektunabhängig und können ausgeführt werden, ohne dass eine Instanz der Klasse vorhanden sein muss - aus diesem Grund ist auch die main-Methode statisch
- `strictfp`: Zwingt Java dazu, Gleitkommaoperationen innerhalb dieser Methode streng nach IEEE 754 auszuführen. Sorgt im Wesentlichen dafür, dass auf allen Systemen bei Rechnungen die gleichen Ergebnisse herauskommen, da die Hardwareunterstützung von floating point values je nach System mehr oder weniger genau ist (teilweise rechnen manche Prozessoren intern mit mehr als 79 Bit). Allerdings wird die Rechnung dabei selbst langsamer und ungenauer.
- `synchronized` und `volatile` haben etwas mit Threading zu tun und seien hier nur der Vollständigkeit halber erwähnt.
- `native` erlaubt das Einbinden von Methoden aus externen Bibliotheken, die in Nicht-Java-Code geschrieben sind

6.6 Der Ursprung: die main-Methode

Damit ein Programm ein eindeutiges Ablaufverhalten besitzt, muss es eindeutig definiert sein. Dazu gehört auch, dass wir Java sagen, wo unser Programm eigentlich starten soll. Dazu gibt es die sogenannte `main`-Methode. Diese hat **IMMER** die exakt gleiche Deklaration:

```
1 public static void main(String[] args) {
2     // das hier wird beim Programmstart ausgeführt
3 }
```

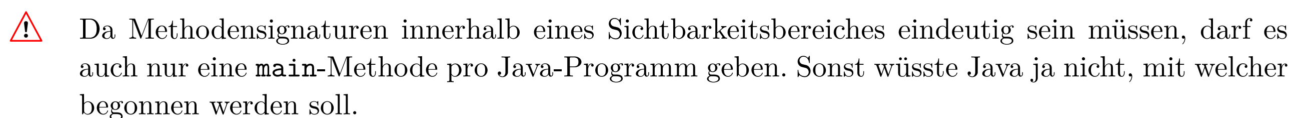


Macht ihr in dem Ausdruck oben auch nur den kleinsten Tippfehler, ist es so gut wie sicher, dass euer Programm nicht startet. Das Einzige, was ihr beliebig definieren könnt, ist der Name des Arrays (hier `args`).

Diese Deklaration ist auch durchaus logisch:

- `public` ist die Sichtbarkeit. Da die Java Virtual Machine sozusagen von außen auf unser Programm schaut, muss die `main`-Methode global sichtbar sein.
- Da zum Zeitpunkt des Programmstartes noch keine Objekte erzeugt wurden, die `main`-Methode aber bereits existieren muss, um ausgeführt werden zu können, muss sie als `static` gekennzeichnet werden.
- Warum sollte unser Programm Objekte an die JVM zurückgeben? Das stünde auch im Kontrast zum Sicherheitskozept von Java. Also eine „leere“ Rückgabe: `void`.
- `main` - Nun ja, jeder wird doch über seinen Namen identifiziert. ;)

- Jetzt wird es ein klein wenig komplizierter: theoretisch könnt ihr euer Programm auch über eine Kommandozeile starten. Dabei könnt ihr eurem Programm diverse Parameter als `Strings` (Zeichenketten) übergeben, die durch ein Leerzeichen getrennt sind. Praktischerweise trennt Java diese Parameter an den Leerzeichen und übergibt sie eurem Programm als `String[ ]`.
- `args`: Den Array könnt ihr nennen wie ihr wollt - allerdings hat sich als Name `args` (für „arguments“) eingebürgert.



Da Methodensignaturen innerhalb eines Sichtbarkeitsbereiches eindeutig sein müssen, darf es auch nur eine `main`-Methode pro Java-Programm geben. Sonst wüsste Java ja nicht, mit welcher begonnen werden soll.

6.7 Überladen von Methoden

Angenommen, wir möchten dem Benutzer mehrere Möglichkeiten bieten, wie er auf ähnliche Funktionalität zugreifen kann, die genaue Ausführung aber von der Art und Anzahl Parameter abhängt. Dafür können wir in Java Methoden überladen, indem wir mehrere Methoden mit gleichem Namen, aber unterschiedlichen Signaturen schreiben. Das ginge z.B. so:

```
1 // eindimensional: ein Parameter
2 public double abs(double value) {
3     return Math.abs(value);
4 }
5 // zweidimensional: zwei Koordinaten
6 public double abs(double x, double y) {
7     return Math.sqrt(x * x + y * y);
8 }
```

Je nachdem, wie viele Parameter der Aufrufer der `abs`-Methode mitgibt, wird die richtige Funktionalität ausgewählt.

6.8 Rekursion

Wenn du noch nicht verstanden hast, was Rekursion bedeutet, dann schau hier: Rekursion.

Okay, Spaß beiseite. Aber eigentlich war diese Zeile alles, was du über Rekursion wissen musst. Die Idee bei Rekursion ist, dass sich eine Methode selbst aufruft. Ja, das ist erlaubt. Beispiel:

```
1 public static void main(String[] args) {
2     int fak = Fakultaet(4);
3 }
4 static int Fakultaet(int input) {
5     if (input == 1) {
6         return input;
7     }
8     return input * Fakultaet(input - 1);
9 }
```

Was passiert hier? Die Idee ist, dass $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1$ gilt. Wir rufen also eine Methode `Fakultaet` auf, die ihren Parameter mit `Fakultät(input - 1)`

multipliziert. Doch wofür ist die `if`-Abfrage da? Wir müssen die Rekursion irgendwann beenden. Es macht keinen Sinn, unser Ergebnis mit $0, -1, -2$ usw. zu multiplizieren - daher befehlen wir eine **Abbruchbedingung**: Die Rekursion beendet sich, sobald `input` 1 ist.

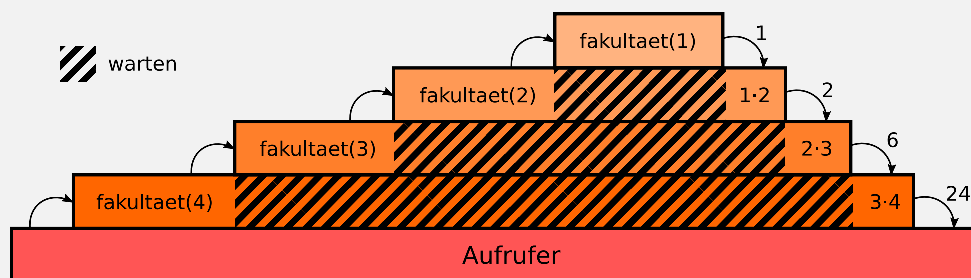
Eine rekursive Methode ruft sich selbst auf und hat eine Abbruchbedingung.

Der Aufrufstapel

Wir werden in Kapitel 10.4 noch sogenannte Stapel kennenlernen. In Zusammenhang mit Rekursion ist es allerdings hilfreich, sich klar zu machen, was ein ganz bestimmter Stapel macht.

Wird eine Methode aufgerufen, landet sie auf dem sogenannten **Aufrufstapel**. Dieser Stapel arbeitet immer die oberste Methode ab. Ist er damit fertig, wird der Aufruf aus dem Stapel entfernt und mit der nun oben liegenden Methode weiter gemacht. Das ist nun die Aufrufermethode. Anschaulich gesprochen springt der Programmzeiger an die Stelle der Aufrufermethode zurück und nimmt dabei das Ergebnis der aufgerufenen Methode mit.

Am Beispiel des vorherigen Code:



Das macht deutlich, wieso wir die Abbruchbedingung brauchen: Auch so ein Stapel ist irgendwann einmal voll. Passiert dies, wirft Java eine sogenannte `StackOverflowException`.

Rekursionen sind übrigens was ganz tolles, weil sich damit viele komplizierte Probleme wie Sortieralgorithmen oder Parser stark vereinfachen lassen und im Code schöner aussehen. Aber sie haben auch einen Haken: Im Vergleich zu iterativen Methoden gleicher Funktionalität sind sie langsamer und brauchen mehr Arbeitsspeicher.

Folgende Aussage ist in diesem Zusammenhang überaus wichtig:

Jeder rekursiver Algorithmus lässt sich auch iterativ implementieren.[14]

Das kann nur manchmal wirklich knifflig werden!

7 Objektorientiertes Programmieren

7.1 Objektorientierte Sprachen

Ihr habt jetzt alle zu Genüge vorgebetet bekommen, dass es sich bei Java um eine **objektorientierte Sprache** handelt. Doch was sind objektorientierte Sprachen eigentlich? Hier zitieren wir mal aus der „iX“ zu zitieren:

[14] Ja, das kann man beweisen und der Beweis findet sich in jedem Buch über theoretische Informatik.

> Objektorientierte Programmiersprachen stellen heutzutage das gängigste Programmierparadigma in der Industrie dar. Daten und die dazugehörigen Algorithmen sind hier in Objekten vereint. Ein Domänen-Modell bildet reale Entitäten durch Klassen ab und manipuliert Daten per Methoden. Mechanismen wie Vererbung, Komposition und Aggregation sowie Datenkapselung machen die Stärken des objektorientierten Paradigma aus.
> *iX, 12 - Dez. 2009, S. 56*

Das ist eine ziemlich komplizierte Definition, aber in den folgenden Kapiteln wollen wir euch die OOP an einfachen Beispielen nahe bringen.

7.2 Objektorientierung

Die Idee hinter Objektorientierung ist, dass wir Programmierer reale Gegenstände als Modelle in unser Programm übertragen und diese miteinander kommunizieren. Das ermöglicht uns bei übersichtlicherem Code sehr viel komplexere Informationsflüsse. Konzepte wie Vererbung und Polymorphie sparen uns jede Menge komplizierte `if`-Verzweigungen und dadurch Zeit und Code. Dadurch macht dieses Konzept an einigen Stellen die Zusammenarbeit von Programmierern einfacher.

Fangen wir mal mit einem einfachen Beispiel an und modellieren ein sehr einfaches Haus. Wodurch wird ein einfaches Haus beschreiben? Ein Haus hat eine Haustür und mehrere Zimmer. Die Tür hat Eigenschaften, z.B. Größe, Material und Position; sie hat auch Methoden, wie Abschließen und Öffnen. Wir können dem Haus nun nicht nur eine Haustür, sondern auch eine Hintertür hinzufügen, ohne viel Code: einfach nur eine Zeile mehr.

```
1  public class Tuer {
2      /*hier werden alle "Eigenschaften" & "Fähigkeiten"
3       *einer Tür im Allgemeinen definiert */
4  }
5  public class Haus {
6      //...
7      Tuer haustuer;
8      haustuer = new Tuer();
9      Tuer hintertuer;
10     hintertuer = new Tuer();
11     int anzahlZimmer;
12     //...
13 }
```

In diesem kurzen Beispiel wird im ersten Teil der „Bauplan“ einer Tür beschreiben - was sie kann, was für Eigenschaften sie hat. Im zweiten Teil deklarieren wir zwei Variablen vom Typ `Tuer`, die also dem genannten Bauplan folgen. Noch weiß der Computer erst, dass es mal solche Variablen geben soll - erst durch den Aufruf `new Tuer();` erzeugen wir eine Tür (was genau das bedeutet, erklären wir später).

Wie gesagt, es lassen sich beliebig viele Objekte vom Typ `Tuer` erzeugen - die Klasse Tür ist also nur eine Blaupause.

Wir können unsere Klassen auch beliebig verschachteln: Man könnte z.B. ein Dorf modellieren, das aus mehreren Häusern besteht, die jeweils mehrere Türen haben. Das geht so lange, bis alle Objekte nur noch aus primitiven Datentypen bestehen.

7.3 Die Schablonen: Klassen

Wer dieses Beispiel verinnerlicht hat, ahnt eventuell: Klassen sind viel mehr als das Sammelsurium für Methoden und Variablen, für das wir sie bisher genutzt haben. Sie entfalten ihr volles Potential erst, wenn man sie als Bauplan für Objekte benutzt.

Eine Klasse definiert einen neuen Datentyp und zugehörige Operationen.

Dieser lässt sich auch instantiieren, so dass ihr Variablen diesen Typs anlegen könnt.

7.3.1 Die Deklaration einer Klasse

Eine Klasse ist zunächst ein Block, der in einem Package liegen kann, aber nicht muss. Pro Datei darf es in Java nur eine Klasse geben, die mit dem Zugriffsmodifizierer `public` versehen ist. Die Deklaration sieht so aus:

Modifizierer + Klassenname

Dabei hat es sich eingebürgert, den Klassennamen groß zu schreiben.

Danach folgt in geschweiften Klammern der Inhalt der Klasse. Was da alles drin stehen kann, wollen wir uns in den folgenden Kapiteln ansehen.

7.3.2 Jetzt geht's los: Instanzen

Wenn ihr ein Objekt mit einem komplexen Datentyp erzeugt, erzeugt ihr eine Instanz einer Klasse, ihr **instantiiert** sie also. Damit ihr euch die Vorgänge, die hierbei ablaufen, besser vorstellen könnt, führen wir jetzt mal ein kleines Modell ein:

- Objekte primitiver Datentypen wären in unseren Fall Notizzettel
- Objekte komplexer Datentypen wären Schachteln/Kartons. Auf den Kartons steht ein Namensschild, das dem Namen des Objekts entspricht
- Methoden: malen/tackern/lochen/etc.
- Klassen sind Packlisten für Zettelkisten

Wenn wir jetzt ein neues Objekt instantiieren, müssen wir den Konstruktor (siehe nächstes Kapitel) der entsprechenden Klasse aufrufen (s.o.: `Tuer meineHausTuer = new Tuer();`). Der Konstruktor nimmt jetzt einen leeren Karton (also ein neues Objekt) und die Packliste, die in der Klasse definiert wird. Anschließend arbeitet er die Liste ab und legt nun Notizzettel (primitive Datentypen), Werkzeuge (z.B. Locher/Stifte) und kleinere Schachteln (komplexe Objekte) in unseren Karton. Am Schluss noch den Deckel drauf und fertig ist das Objekt.

Nun haben wir zwar unser fertiges Objekt (den Karton), wir wollen damit auch arbeiten. Wir haben bisher Methoden immer so aufgerufen, indem wir ihr ein Objekt (z.B. einen `int`) als Parameter übergeben haben. Euch wird aufgefallen sein, dass diese Methoden das Schlüsselwort `static` hatten. Doch was sind nicht-statische Methoden? Ganz einfach: Instanzmethoden! Bei ihnen ist das Ergebnis abhängig von dem Objekt, zu dem sie gehören. Das mag auf dem ersten Blick verwirrend sein, aber schauen wir uns das mal an unserem Modell an:

Als erstes gehen wir davon aus, dass wir ein Blatt (Variablenname `blatt`) lochen wollen. Das gelochte Blatt wird später in der Variable `public Blatt gelochtesBlatt` abgelegt. Unser Locher (die Methode `public Blatt lochen(Blatt b)`) liegt jetzt aber in einer der Schachteln

(Variablenname `werkzeugkoffer`) in unserem Karton (Variablenname `kiste`).

Um an den Locher heranzukommen, müssen wir erst mal dem Computer sagen, wo er zu suchen hat. Wir nehmen also aus der `kiste` die Variable `werkzeugkoffer` heraus und öffnen auch diesen Karton. Nun nehmen wir aus der Schachtel unseren Locher und übergeben ihm das zu lochende Blatt. Der Locher locht das Blatt und gibt es uns zurück. In Java sähe das folgendermaßen aus:

```
1 public Karton kiste = new Karton();
2 public Blatt zuLochendesBlatt = new Blatt();

3 //hier steht der vorangegangene Absatz in Java-Code:
4 public Blatt gelochtesBlatt
5     = kiste.werkzeugkoffer.lochen(zuLochendesBlatt);
```

Das „Öffnen“ und das „Heraussuchen“ wird also in Java mit einem `.` symbolisiert.

Angenommen, die Klasse `Blatt` würde eine **Methode bieten**, die Kaffee über das Blatt kippt, haben wir jetzt wieder zwei Möglichkeiten: Wir könnten einfach im nächsten Schritt die Methode auf unser gelochtes Blatt aufrufen. Oder wir ziehen beides zusammen:

```
1 //...wie oben

2 //1. entweder so:
3 public Blatt gelochtesBlatt =
↪ meinWerkzeug.schachtel.lochen(zuLochendesBlatt);
4 public Blatt dreckigesBlatt = gelochtesBlatt.kaffeeVerschuetten();

5 //2. oder direkt so (eigentlich in einer Zeile!):
6 public Blatt dreckigesBlatt =
↪ kiste.werkzeugkoffer.lochen(zuLochendesBlatt).kaffeeVerschuetten();
```

Das geht deshalb, weil unsere Lochen-Methode ein Objekt vom Typ `Blatt` zurück gibt und dieses eine Instanzmethode `public Blatt kaffeeVerschütten()` hat, die wiederum ein Objekt vom Typ Blatt `Blatt` zurück gibt.

7.3.3 Konstruktoren - Die Baumeister

Wie eben schon angedeutet: Wenn ihr ein neues komplexes Objekt erstellen wollt, müsst ihr einen **Konstruktor** aufrufen. Dazu dient in Java das Schlüsselwort `new`, gefolgt vom Namen der Klasse und runden Klammern, in denen analog zu Methoden dem Konstruktor die eventuell erwarteten Parameter übergeben werden.

Zusammen sieht das so aus: `new String("hallo");`.[15]

Wie man an den runden Klammern sieht, sind auch Konstruktoren eigentlich Methoden, die allerdings genau wie die Klasse heißen und eine Instanz dieser Klasse zurückgeben.

Das heißt aber auch, dass im Konstruktor deklarierte Variablen danach nicht mehr zur Verfügung stehen.

Jede Klasse braucht einen Konstruktor und solltet ihr keinen schreiben, verwendet Java den so genannten **Standard-Konstruktor**. Dieser ist parameterlos und sorgt dafür, dass man erstellte

[15] String ist der einzige komplexe Datentyp, bei dem auch `String var = "hallo";` funktioniert.

Klassen auch instanziieren kann.

⚠ Bei einem Konstruktor darf man **keinen** Rückgabewert angeben, also noch nicht einmal `void`. Außerdem macht es im Allgemeinen Sinn, für Konstruktoren die Sichtbarkeit `public` zu wählen.

```
1  public class Fueller {
2      private boolean patroneEingelegt;
3      //der Konstruktor
4      public Fueller(boolean patroneMitgeliefert) {
5          patroneEingelegt = patroneMitgeliefert;
6      }
7  }

8  public class Test {
9      public static void main(String[] args) {

10         //erstmal deklarieren
11         public Fueller meinFueller;

12         //mit einem neuen Fueller initialisieren,
13         //also den Konstruktor aufrufen:
14         meinFueller = new Fueller(false);
15     }
16 }
```

7.3.4 Instanzvariablen

Das sind die „Eigenschaften“ eines Objekts, deshalb werden Instanzvariablen auch gerne als **Attribute** bezeichnet. Attribute sind Variablen (oder Konstanten), die innerhalb des **Klassen-Rumpfs** (also nicht in Methoden oder dem Konstruktor) mindestens deklariert werden. Sie können dort auch direkt initialisiert werden, wenn man unabhängig vom Objekt zu Beginn einen Standardwert festlegen möchte. Existiert kein solcher Standardwert (z.B. beim Hersteller eines Autos), macht es Sinn, die Variablen abhängig von den im Konstruktor übergebenen Parametern zu initialisieren.

⚠ Aus Gründen der Lesbarkeit des Codes hat sich unter Entwicklern allerdings durchgesetzt, alle Variablen erst im Konstruktor zu initialisieren. Wichtig ist nur, dass Variablen, die später noch verwendet werden sollen, nicht erst im Konstruktor deklariert werden, denn sonst sind es lokale Variablen.

```
1  public class Mitarbeiter {
2      private int hierarchiestufe = 0;  // bitte vermeiden
3      private boolean geschlecht;    //nur deklariert
4      private float stundenLohn;

5      public Mitarbeiter(boolean istMann) {
6          stundenLohn = 27.18f;    // schon viel übersichtlicher!
7          geschlecht = istMann;   // Init. mit aktuellem Parameter
8          char waehrung = 'e';    // Vorsicht: lokale Variable
9      }
```

```
10      public void werteAusgeben() {
11          System.out.println("Stundenlohn: " + stundenLohn + waehrung);
12          // Fehler: "waehrung" war ein lokale Variable des Konstruktors

13          System.out.println("Hierarchiestufe: " + hierarchiestufe);

14          System.out.println(
15              "Geschlecht: " + (geschlecht ? "Männlich" : "Weiblich"));
16      }
17  }
```

Analog könnt ihr natürlich bei Konstanten (Schlüsselwort `final`) vorgehen.

7.3.5 Klassenvariablen - die große Gemeinsamkeit

Während Instanzvariablen fest einem Objekt zugeordnet sind, kann man auch Variablen erstellen, die zu einer Klasse gehören - diese nennt man entsprechend Klassenvariablen.

Das Schlüsselwort `static` kennt ihr ja schon von Methoden. Auf als statisch markierte Methoden kann man zugreifen, ohne dass ein Objekt dieser Klasse gebildet wurden - sie sind objektunabhängig. Einen analogen Effekt lässt sich mit `static` bei Variablen erzielen: Verwendet ihr bei der Deklaration den Modifizierer `static` vor den Variablennamen, ist die Variable „statisch“, sprich sie existiert schon, bevor ein Objekt dieser Klasse erzeugt wurde und ist von den einzelnen Objekten unabhängig.

Aber Vorsicht: Jede Instanz „verfügt“ zwar über diese Variable/Konstante, allerdings verweisen alle Objekte auf den gleichen Bereich im Speicher. Das bedeutet: Verändert eine Instanz den Wert einer statischen Variablen, ändert sich der der Wert auch für alle anderen Instanzen!

Schauen wir uns mal ein Beispiel an, wo man statische Variablen gut gebrauchen kann:

```
1   public class Druckwarteschlange {
2       static Druckwarteschlange instanz;
3       private Druckwarteschlange(){
4           // nicht öffentlich zugänglicher Konstruktor
5           // Da er existiert, wird kein Standard-Konstruktor erzeugt
6       }
7       public static Druckwarteschlange GetInstanz () {
8           if (instanz == null)
9               instanz = new Druckwarteschlange();
10          return instanz;
11      }
12      public void AuftragEinreihen(Dokument d)
13      {
14          // keine stat. Methode, da das Ergebnis vom Objekt abhängt
15      }
16  }
```

Bei diesem Beispiel handelt es sich um ein sogenanntes **Entwurfsmuster** (engl. pattern) namens „Singleton“, das verwendet wird, wenn mehrere Objekte oder gar Programme auf die

gleiche Funktionalität zugreifen müssen[16].

Eine Anwendung dieses Musters wäre wie im Beispiel eine Druckerwarteschlange. Möchte ein Objekt einen Druckauftrag absenden, wäre es nicht sinnvoll, dass es eine eigene Instanz der Klasse Druckerwarteschlange erzeugt und dieser den Druckauftrag übergibt. Dann gäbe es nämlich mehrere Warteschlangen für eine Ressource (den Drucker), was zu Problemen führen würde. Stattdessen ruft es die aktuelle Instanz der Druckwarteschlange (mit `Druckwarteschlange.GetInstanz()`) ab und übergibt dieser den Druckauftrag. Also so:

```
1 Druckwarteschlange.GetInstanz().AuftragEinreihen(MeinDokument);
```

Normalerweise wird diese Instanz vom System beim Hochfahren erzeugt. Sollte es derzeit jedoch keine Instanz geben - beispielsweise, weil der Treiber abgestürzt ist - wird mit dieser Methode gleich eine erzeugt. Der Code funktioniert in dieser Weise, weil die Variable `instanz` der Klasse `Singleton` statisch ist und daher objektübergreifend verfügbar ist, und weil die Methode `GetInstanz()` ebenfalls statisch und öffentlich ist.

7.3.6 Vorsicht Nebel - Verdeckung

Normalerweise wählt ihr beim Programmieren Variablennamen und Methodenparameter so, dass es für andere, die euren Code lesen beziehungsweise eure Methoden verwenden, klar ersichtlich ist, welche Funktion ein Parameter oder eine Variable hat. Dadurch kann es passieren, dass ihr eine lokale Variable identisch einer Instanz- oder Klassenvariable benennt (ja, das geht!): Innerhalb eurer Methode wird dann über den Variablennamen nur noch die lokale Variable angesprochen. Braucht ihr trotzdem Zugriff auf eine Instanzvariable, hilft euch das Schlüsselwort `this`. Es verweist immer auf die aktuelle Instanz, also auf das Objekt, in dem die Methode gerade abläuft. Zudem lässt sich `this` als Instanzvariable verstehen, und ist somit in statischen Methoden nicht verfügbar.

```
1  public class Kreis {
2      private int radius;
3      // an dieser Stelle Konstruktor, der "radius" initialisiert

4      public int getRadius() {
5          // hier gibts nur eine Variable namens radius,
6          // daher kein "this" notwendig
7          return radius;
8      }
9      public void setRadius(int radius) {
10         //Vorsicht, Verdeckung, deshalb:
11         this.radius = radius;
12     }
13 }
```

7.3.7 Zugriffe steuern und abfangen - get- und set-Methoden

Gerade haben wir zum ersten Mal Get- und Set-Methode verwendet, die man in der objektorientierten Programmierung gerne benutzt, um die Zugriffe auf Instanz- oder Klassenvariablen zu

[16] Zu bemerken sei, dass dieser Code nicht threadsicher ist. Dafür müsste in der `getInstanz`-Methode das Schlüsselwort `synchronized` verwendet werden, was aber an dieser Stelle eher verwirren würde.

regeln. Soll der Kreis, wenn sein Radius geändert wird, gleich neu gezeichnet werden, schreibt man einfach einen entsprechenden Aufruf in die Methode `setRadius(int radius)`. Außerdem soll die Änderung nur gemacht werden, wenn der übergebene Radius positiv ist. Verwendet ihr den `GraphicScreen` der `Prog1Tools`, ginge das z.B. so:

```
1 public void setRadius(int radius) {
2     if (radius > 0) {
3         this.radius = radius;

4         // Der GraphicScreen verwendet das Singleton-Muster!
5         GraphicScreen screen = GraphicScreen.getInstance();

6         // Parameter: Koordinaten, Radius, Ausgefüllt (true / false)
7         screen.drawCircle(x, y, radius, true);
8     }
9 }
```

Wäre stattdessen die Variable `radius` öffentlich zugänglich, gäbe es keine Möglichkeit, das zu steuern. In der Programmierung hat sich heute weitgehend durchgesetzt, dass man Instanzvariablen als `private` deklariert und anderen Objekten nur über Getter und Setter zur Verfügung stellt, selbst, wenn diese nichts anderes machen, als im Beispiel unter Verdeckung. Dieses Konzept nennt man auch **Datenkapselung**. Namenskonvention ist hier, `get` und `set` kleinzuschreiben und die eigentliche Variable in Großschreibung anzuhängen.

7.4 Evolutionstheorie bei Klassen: Vererbung

Bis jetzt haben wir schon viele schöne Klassen erstellt, die alle unterschiedliche Funktionalitäten hatten. Doch was ist, wenn ihr über eine existierende Klasse hinaus eine Klasse erstellen wollt, die sehr ähnliche Funktionalität hat, aber etwas spezialisierter ist? Mit eurem bisherigen Wissen würdet ihr die alte Klasse kopieren und die neuen Funktionen hinzufügen, wo ihr sie braucht. Das ist aus zwei Gründen nicht besonders vorteilhaft:

- Es existiert sehr viel Code doppelt. Das braucht mehr Speicher, mehr Zeit beim Kompilieren und macht mehr Probleme bei der Fehlersuche.
- Wenn ihr beide Klassen um die selbe Funktionalität erweitern wollt, müsst ihr das auch in beiden Klassen machen - und oft gibt es nicht nur eine Unterklasse, sondern mehrere!

Glücklicherweise lässt euch die objektorientierte Programmierung bei solche einer Aufgabe nicht im Stich. Hierfür existiert das Konzept der **Vererbung**, auch „ableiten“ oder „erweitern“ genannt. Das entsprechende Schlüsselwort heißt `extends`, es wird hinter den Namen der abgeleiteten Klasse gestellt und dahinter kommt der Name der vererbenden Klasse. Also:

```
1 public class Haustuer extends Tuer
```

Der Übersichtlichkeit halber: Schreibt auch abgeleitete Klassen in eine eigene Datei.

Folgende Sachen muss man bei der Vererbung verinnerlichen:

- Abgeleitete Klassen haben automatisch alle Methoden und Attribute der vererbenden Klasse.

  Auch private Methoden und Attribute werden vererbt (sonst würden ja viele der vererbten

Methoden nicht funktionieren, da sie auf diese zugreifen). Sie sind allerdings nicht sichtbar! Wollt ihr dennoch auf diese Methoden und Attribute zugreifen, müsst ihr der vererbenden Klasse Getter bzw. Setter hinzufügen oder einen anderen Zugriffsmodifikator wählen.

- In Java kann eine Klasse grundsätzlich nur von einer Klasse erben. Solltet ihr mehr benötigen, müsst ihr Interfaces verwenden.
- Eine Klasse kann jedoch problemlos von einer Klasse erben, die ihrerseits bereits abgeleitet ist - solche Vererbungsketten sind möglich und sinnvoll.
- Wenn eine Klasse als `final` deklariert ist, darf keine andere Klasse von ihr erben. Dieses Schlüsselwort verwendet man aus Sicherheits- und Geschwindigkeitsgründen.

Schauen wir uns mal ein etwas ausführlicheres Beispiel an:

```
1  public class Tuer {
2      protected float hoehe;
3      protected float breite;
4      private boolean istGeschlossen;

5      public Tuer(float breite, float hoehe) {
6          // Initialisieren der Instanzvariablen mit den akt. Parametern
7          this.hoehe = hoehe;
8          this.breite = breite;
9          this.istGeschlossen = true;
10     }
11     public void schliessen() {
12         istGeschlossen = true;
13     }
14     public void oeffnen() {
15         istGeschlossen = false;
16     }
17 }

18 public class Haustuer extends Tuer {
19     String schluessel;

20     //der Konstruktor der Kindklasse
21     public Haustuer (float hoehe, float breite, String schluessel) {
22         //Aufruf des Eltern-Konstruktor
23         super(hoehe, breite);
24         //der String stellt eine Art Schlüsselkombination dar
25         this.schluessel = schluessel;
26         System.out.println(istGeschlossen);
27         //Fehler: Variable "istGeschlossen" hier nicht sichtbar/bekannt
28     }
29 }
```

Ein Schlüsselwort haben wir jetzt noch nicht erklärt: `super`.

Die Unterklasse hat selbstverständlich einen anderen Konstruktor als die vererbende Klasse:

Klar, es müssen ja auch andere Variablen initialisiert und zugewiesen werden. Damit Java jedoch trotzdem versteht, dass ein vom `Haustuer`-Konstruktor erzeugtes Objekt irgendwie trotzdem eine `Tuer` ist, müssen wir beide Konstruktoren verbinden. Das geschieht mit eben diesem Schlüsselwort, dem wir die Parameter eines Elternkonstruktors mitgeben müssen.

`super` ist immer die erste Zeile des Konstruktors der erbenden Klasse.

Das ist nur nötig, wenn es in der vererbenden Klasse keinen parameterlosen Konstruktor gibt. Gibt es dagegen einen, macht Java die nötige Arbeit alleine!

7.4.1 Methoden überschreiben und `@Override`

Manchmal wollen wir nicht alle Methoden der vererbenden Klasse in der abgeleiteten Klassen in gleicher Weise haben. Beispielsweise soll man auch eine Haustür öffnen können, allerdings nur mit dem richtigen Schlüssel. So etwas lässt sich in Java realisieren, indem man in der Unterklasse die fragliche Methode einfach neu definiert. Wenn diese die gleiche Signatur - also gleichen Namen und Parameter - und den gleichen Rückgabetyp hat wie eine Methode der Elternklasse, ist die ursprüngliche Methode nicht mehr vorhanden - das nennt man überschreiben. Solche Methoden markiert man mit `@Override` vor der Methodendeklaration

Wenden wir das einmal auf die oben definierte Klasse `Hautuer` an:

```
1  public class Haustuer extends Tuer {
2      String schluessel;

3      public Haustuer (float hoehe, float breite, String schluessel) {
4          super(hoehe, breite);
5          this.schluessel = schluessel;
6      }

7      @Override
8      public void Oeffnen() {
9          // ohne Schlüssel passiert gar nichts!
10     }
11     public void Oeffnen(String schluessel) {
12         if (schluessel.equals(this.schluessel)) {
13             super.Oeffnen();
14         }
15     }
16 }
```

Wird nun von einer Tür die Methode `Oeffnen()` aufgerufen, unterscheidet sich das Verhalten, je nach dem, ob es sich um eine `Tuer` oder eine `Haustuer` handelt. Eine `Tuer` öffnet sich, bei `Haustuer` wurde die entsprechende Methode jedoch mit einer leeren Methode überschrieben, folglich passiert nichts - das ist unser gewünschtes Verhalten.

In der Klasse `Haustuer` gibt es jetzt jedoch eine Methode, mit der man die Tür öffnen kann - allerdings nur mit dem Schlüssel. Im Gegensatz zu alten Methode hat `Oeffnen(String schluessel)` jedoch einen Parameter. Wenn dieser gleich dem Schlüssel ist, wird die Methode `Oeffnen()` der Elternklasse aufgerufen.[17,18]

[17] Das `super` ist hier notwendig, weil in der abgeleiteten Klasse `Oeffnen()` schon überschreiben wurde

[18] Ja, dieses Beispiel ist eigentlich aus mehreren Gründen etwas unschön. Aber es illustriert das Prinzip.

7.4.2 Gemeinsamkeiten erzwingen - abstrakte Klassen

Bei Vererbung hatten wir es schon erwähnt: Einer der Grundgedanken der objektorientierten Programmierung ist, Objekte erst mal möglichst allgemein zu beschreiben und dann mittels Vererbung zu spezialisieren. In manchen Fällen geht man noch weiter: Man erstellt eine „Oberklasse“, die so allgemein ist, dass es keinen Sinn machen würde, von dieser Klasse ein Objekt zu erstellen (Beispiel: Niemand braucht ein allgemeines „Fahrzeug“, sondern eher ein „Auto“ oder einen „LKW“). Daher gibt es in Java abstrakte Klassen und abstrakte Methoden, in beiden Fällen heißt das entsprechende Schlüsselwort `abstract`.

Auch hier gibt es wieder ein paar Regeln, die man sich merken muss:

- Eine abstrakte Klasse kann nicht instanziiert werden und hat somit auch keinen Konstruktor.
- Bei einer abstrakten Methode wird der Methodenrumpf weggelassen und durch ein Semikolon ersetzt.
- Enthält eine Klasse eine abstrakte Methode, muss sie selbst als abstrakt deklariert werden.
- Eine abstrakte Klasse kann beliebig viele abstrakte Methoden beinhalten, es muss jedoch keine vorhanden sein!
- Eine abgeleitete Klasse muss alle abstrakten Methoden der Elternklasse implementieren, sonst ist sie selbst wieder abstrakt und muss als solche markiert werden.
- Die Kombination von `final` und `abstract` ist nicht erlaubt (macht ja auch keinen Sinn).

Wie mächtig diese Möglichkeit ist, soll folgendes Beispiel zeigen.

```
1  public abstract class GeometrischeForm {
2      public abstract double berechneFlaecheninhalt();
3  }
4  public class Rechteck extends GeometrischeForm {
5      private double laenge;
6      private double breite;

7      public Rechteck(double laenge, double breite) { /* ... */ }

8      public double berechneFlaecheninhalt() {
9          return laenge * breite;
10     }
11 }
12 public class Kreis extends GeometrischeForm {
13     private double radius;

14     public Kreis(double radius) { /* ... */ }

15     public double berechneFlaecheninhalt() {
16         return radius * radius * Math.PI;
17     }
18 }
```

Wie sollte man auch den Flächeninhalt einer geometrischen Form allgemein festlegen? Dieser wird stattdessen in den Unterklassen definiert. Trotzdem: Als Programmierer kann man sich sicher sein, dass jedes Objekt, dass zu einer Unterklasse von `GeometrischeForm` gehört, auch die Methode `berechneFlaecheninhalt()` hat. Praktisch, nicht?

7.4.3 Java, wechsel dich: Polymorphie

Polymorphie bedeutet so viel wie „Vielgestaltigkeit“ und ist eines der zentralen Konzepte der objektorientierten Programmierung. Es gibt mehrere Arten der Polymorphie, dabei haben wir die Vielgestaltigkeit von Methoden beim Überladen schon kennengelernt (siehe Kapitel 6.7). Schauen wir uns nun die Polymorphie von Objekten an: diese bedeutet, dass man jedes Objekt auch als Objekt seiner Elternklasse ansehen kann[19]. Andersherum ist das nicht so ohne weiteres möglich. Ein einfacherer Vergleich aus der Realität: Ein Vorstandsvorsitzender ist ein Angestellter, aber nicht jeder Angestellter ein Vorstandsvorsitzender.

```
1  class Angestellter
2  {
3      protected String name;
4      protected double gehalt;
5      protected Date geburtsdatum;

6      public Angestellter(String name, double gehalt, Date datum) {
7          this.name = name;
8          this.gehalt = gehalt;
9          this.geburtsdatum = datum;
10     }

11     public void personalienAusgeben() {
12         System.out.println("Name: " + name);
13         System.out.println("Gehalt: " + gehalt + " Euro");
14         System.out.println("Geburtsdatum: " + geburtsdatum);
15     }
16 }

17 class Abteilungsleiter extends Angestellter
18 {
19     protected String abteilung;

20     public Abteilungsleiter(/* wie oben */, String abteilung) {
21         super(name, gehalt, datum);
22         this.abteilung = abteilung;
23     }

24     @Override
25     public void personalienAusgeben() {
26         super.personalienAusgeben();
```

[19] Jedes Objekt hat eine Elternklasse: in Java erben alle Objekte von der Klasse `Object`, u.a. die Methode `toString()`.

```
27          System.out.println("Abteilung: " + abteilung);
28      }

29      public void befoerdern(Angestellter a) {
30          a.gehalt *= 1.1;
31      }
32  }
```

In diesem einfachen Beispiel wird eine Methode in der abgeleiteten Klasse `Abteilungsleiter` überschrieben, nämlich `personalienAusgeben()`. Hier rufen wir zunächst die Methode der Oberklasse auf, die uns die allgemeinen Daten eines Angestellten ausgibt, danach wird noch die Abteilung ausgegeben. Zudem geben wir dem `Abteilungsleiter` die Fähigkeit, einen beliebigen Angestellten zu befördern, dessen Referenz der Methode übergeben wird.

Hier nun die Main-Methode, an der wir die Polymorphie demonstrieren können:

```
1  public static void main(String[] args) {
2      Angestellter a0001
3          = new Abteilungsleiter("Chef", 5000, new Date(50, 5, 5), "BMW");
4      Angestellter a1234
5          = new Angestellter("Peter Meier", 3000, new Date(80, 1, 1));

6      a0001.personalienAusgeben();

7      ((Abteilungsleiter)a0001).befoerdern(a1234);
8      a1234.personalienAusgeben();

9      // ((Abteilungsleiter) a1234).befoerdern(a1234);
10 }
```

Wir deklarieren und initialisieren zunächst zwei Objekte vom Typ `Angestellter`. Das erste wird jedoch mit dem Konstruktor der Klasse Abteilungsleiter initialisiert - das geht, weil `Abteilungsleiter` von `Angestellter` erbt und somit Instanzen dieser Klasse auch Objekte vom Typ `Angestellter` sind, das zweite stellt einen normalen Angestellten dar. Diese Fähigkeit mag trivial erscheinen, ist aber ziemlich praktisch: So könnten wir beispielsweise alle Angestellten einer Firma - egal, ob `Angestellter` oder `Abteilungsleiter` - in einem Array vom Typ `Angestellter`. Anschließend könnten wir für jedes Element des Arrays eine Methode aus `Angestellter` aufrufen, z.B. `personalienAusgeben()`. Dabei ist jedoch zu beachten, dass stets die spezialisierteste Methode aufgerufen wird - bei einem `Abteilungsleiter` wird also auch die zusätzliche Funktionalität der überschriebenen Methode verwendet.

Wollen wir jedoch wieder auf die spezialisierten Objekte selbst zugreifen, also z.B. auf die Methode `befoerdern(Angestellter a)`, über die nur ein `Abteilungsleiter` verfügt - müssen wir das Objekt erst wieder konvertieren. Den Operator kennen wir schon aus Kapitel 2.2.2 und nennt sich Cast. Genauer gesagt handelt es sich hier um einen **Downcast**. Dieser ist natürlich nur durchführbar, wenn alle benötigten Informationen vorhanden sind, das zu castende Objekt also ursprünglich ein `Abteilungsleiter` war.[20] Dann können wir auch auf die enthaltenen Methoden und Attribute zugreifen!

[20] Aus diesem Grund würde die auskommentierte 10. Zeile einen Fehler produzieren - `a1234` ist kein `Abteilungsleiter`.

Implizit haben wir gerade auch schon einen **Upcast** verwendet - und zwar in der zweiten Zeile der `main`-Methode. Hier haben wir ein Objekt vom Typ `Abteilungsleiter` erzeugt und - ohne weitere Umwandlung - in einer Variable vom Typ `Angestellter` abgelegt. Ein bisschen erinnert dieses Verhalten an die automatische Typumwandlung, und tatsächlich erledigt Java das für uns. Dabei spricht man auch von der **Reduzierung einer Schnittstelle**: In diesem Zustand werden alle zusätzlichen Methoden und Attribute der Klasse `Abteilungsleiter` versteckt und sind erst nach einem Downcast wieder sichtbar.

Jedes Objekt lässt sich auch in Variablen mit dem Datentyp der Elternklasse speichern.

7.5 Interfaces

Wie du schon gelernt hast, definieren Objekte die Interaktionsmöglichkeiten mit anderen Objekten durch Methoden. Die Methoden sind also die **Schnittstelle** - engl. interface - zum restlichen Programm. Ein Handy hat beispielsweise einen An-Schalter, der die elektrischen Komponenten auf der einen Seite und den Benutzer auf der anderen verbindet. Du drückst auf an, und das Handy schaltet sich ein - glücklicherweise musst du nicht jede Komponente einzeln einschalten.

In der OOP gibt es eine analoge Umsetzung. Eine Schnittstelle definiert die Methoden, die eine Klasse später mal haben soll, ohne dass diese gleich implementiert werden muss. So können andere Programmteile diese Schnittstelle schon verwenden, bevor die eigentliche Klasse vorhanden ist. Denn: Ein anderer Programmierer kann sich darauf verlassen, dass die dort spezifizierten Methoden auch wirklich implementiert werden!

Ein weiter Grundgedanke von Schnittstellen ist, dass eine Klasse mehrere Verbindungen zur Umwelt haben kann. Bei Java ist nur Einfachvererbung möglich, aber eine Klasse kann beliebig viele Schnittstellen implementieren.

Diese Eigenschaften sorgen dafür, dass Schnittstellen sehr gerne eingesetzt werden, um abstrakte Funktionalität aus Klassen auszulagern und in vielen Klassen weiterzuverwenden.[21]

In einem Interface definiert man die Köpfe der öffentlich sichtbaren Programmteile, die in jedem Fall in der implementierten Klasse vorhanden sein sollen. Dazu gehören:

- Die Signaturen und Rückgabewerte der öffentlich sichtbaren Methoden
- Die geforderten öffentlichen Variablen und Konstanten (wobei das aufgrund der Kapselung recht selten auftritt)

Eine Klasse, die ein Interface implementiert, also in ein reales Objekt „einbaut“, muss alle dort definierten Methoden und Attribute implementieren.[22] Sie kann natürlich auch darüber hinaus Methoden und Attribute definieren, auch öffentlich zugängliche.

Ein Codebeispiel für ein Interface sähe zum Beispiel so aus:

```
1 interface Vergleichbar {
2     public int vergleiche(Vergleichbar anderes);
3 }
```

[21] Wenn du dir noch nicht direkt etwas darunter vorstellen kannst: Man könnte beispielsweise ein Interface `Addierbar` machen, das nur eine Methode `Addieren()` abstrakt definiert und dessen Implementierung in den einzelnen Klassen voneinander abweicht.

[22] Sollte eine Methode dort nicht ausgeschrieben werden, wird die Klasse selbst wieder abstrakt und muss entsprechend gekennzeichnet werden.

Möchten wir nun eine Klasse mit dem Interface „verbinden“, gibt es dafür das Schlüsselwort `implements`. Möchte man mehrere Interfaces implementieren, trennt man die einzelnen Schnittstellen in der Klassendeklaration durch Kommata. Eine Klasse, die dieses Interface erfüllt (also ausschreibt/implementiert), könnte z.B. die Klasse `SGanzzahl` (kurz für Sortierbare Ganzzahl) sein:

```
1  class SGanzzahl implements Vergleichbar {
2      public int i;

3      public SGanzzahl(int i) {
4          this.i = i;
5      }

6      public int vergleiche(Vergleichbar anderes) {
7          if (! (anderes instanceof SGanzzahl))
8              return -2; // Vergleich nicht anwendbar
9          else {
10             if (i < ((SGanzzahl) anderes).i)
11                 return -1; // dieses Objekt < anderes
12             else if (i == ((SGanzzahl) anderes).i)
13                 return 0; // dieses Objekt == anderes
14             return 1; // dieses Objekt > anderes
15         }
16     }
17 }
```

Wir haben also eine Klasse erstellt, die unser sehr abstraktes Interface implementiert. Dabei wurde die Methode `Vergleiche()` ausgeschrieben. Der Trick an dieser Methode ist, dass ihr egal ist, was für ein Datentyp übergeben wird, solange dieser das Interface `Vergleichbar` implementiert. So können wir auch verschiedene Datentypen vergleichen, solange das Sinn macht und wir das in Java formulieren können.

Schauen wir uns die Methode `Vergleiche` noch mal genauer an: Zunächst wird überprüft, ob sich `anderes` in `SKommazahl` casten lässt. Dafür gibt es das praktische Schlüsselwort `instanceof`, das genau diesen Zweck erfüllt. Wenn dies nicht der Fall ist, wird `-2` für „nicht anwendbar“ zurückgegeben. Wenn sich die Werte vergleichen lassen, wird `anderes` in eine `Ganzzahl` konvertiert und die Attribute verglichen. Je nach Ergebnis dieses Vergleichs wird `0`, `1` oder `-1` zurückgegeben.

Schreiben wir noch eine andere Klasse, die das gleiche Interface implementiert:

```
1  class SKommazahl implements Vergleichbar {
2      public double d;

3      public SKommazahl(double d) {
4          this.d = d;
5      }

6      public int vergleiche(Vergleichbar anderes) {
7          if (!(anderes instanceof SKommazahl))
```

```
8             return -2; // Vergleich nicht anwendbar
9         else { /* analog zu oben */ }
10    }
11 }
```

Und eine passende Main-Methode:

```
1 public class InterfaceTest {
2     public static void main(String[] args) {
3         Vergleichbar[] array = {
4             new SGanzzahl(5), new SGanzzahl(4),
5             new SKommazahl(3.141), new SKommazahl(5) };
6         vergleicheAufeinanderfolgende(array);
7     }
8     public static void vergleicheAufeinanderfolgende(Vergleichbar[] ar) {
9         for (int i = 0; i < ar.length - 1; i++) {
10            System.out.println(ar[i].vergleiche(ar[i + 1]));
11        }
12    }
13 }
```

Also: Auch ein Interface ist ein Datentyp, den man casten kann. Die Regeln für Casts sind analog zur Polymorphie unter Vererbung.

Alle Klassen, die dieses Interface implementieren, haben die Methode `Vergleiche()`, die sich auch aufrufen lässt. Dadurch ergibt sich in dieser Reihenfolge der Output `-1`, `-2` und `1`. Wir könnten nun also eine Sortiermethode schreiben, die vollkommen unabhängig vom Datentyp ist, solange er vergleichbar ist!

In diesem Beispiel wird der Vorteil dieses Systems vielleicht noch nicht deutlich genug, aber wir können beispielsweise eine Klasse `SKennzeichen` schreiben, die `Sortierbar` implementiert - und damit dann einen Array von KFZ-Kennzeichen sortieren, ohne auch nur ein Zeichen an unserem Sortieralgorithmus zu verändern.

7.5.1 Interface oder abstrakte Klasse?

Im Prinzip sieht eine abstrakte Klasse, die keine Methode direkt implementiert, genauso aus wie ein Interface. Der Unterschied liegt vor allem darin, dass in einem Interface keine Methoden ausgeschrieben sein **dürfen** und eine Klasse mehrere Interfaces implementieren kann, aber nur von einer Klasse erben.

Daher sind Interfaces oft die sauberere Lösung, wenn Funktionalität so abstrakt formuliert werden soll, dass alle Methoden erst in den Unterklassen ausgeschrieben werden können.

Als Faustregel kann man sich folgendes merken:

Interfaces beschreiben nur abstrakte Funktionalität,
abstrakte Klassen beschreiben auch abstrakte Objekte.

8 Wegweiser in Java: Referenzen

Wenn ihr eine Variable deklariert, also z.B. `int eineZahl;`, gebt ihr dabei Datentyp und Variablenname an. Bei Java handelt es sich um eine **stark typisierte** Programmiersprache: Daher

könnt ihr in der Variable, nachdem sie deklariert wurde, nur Objekte diesen Datentyps oder davon abgeleitete Objekte (siehe Polymorphie), in dieser Variable „speichern“.

Doch zurück zu Referenzen: Im vorangegangenen Absatz haben wir „speichern“ in Anführungszeichen geschrieben. Dies hat einen ganz einfachen Grund: Nur bei Variablen eines primitiven Datentyps wird der Wert in der Variable gespeichert. Sobald ihr aber ein Objekt habt, ist dies nicht mehr der Fall. Das Objekt wird bei der Instanziierung im Arbeitsspeicher des Computers einmalig erzeugt. Wird das Objekt jetzt einer Variablen zugewiesen, erhält und speichert diese nur die Adresse des Objekts im Arbeitsspeicher.

Ein passendes Analogon dazu ist eine Verknüpfung auf eurem Desktop. Diese weist auch nur auf eine Datei oder ein Programm, das eigentliche Programm liegt glücklicherweise an einer anderen Stelle. Wenn ihr die Verknüpfung kopiert, dann wird kein Duplikat des Programms erzeugt, und wenn ihr etwas am Programm ändert, wirkt sich das auch auf beide Verknüpfungen aus.

Daher: Wenn man zwei Variablen einander zuweist (`Var1 = Var2`), wird einfach nur die Speicheradresse aus `Var2` in `Var1` kopiert, sprich, das eigentliche Objekt bleibt davon unberührt. Das heißt aber auch, dass, wenn das Objekt z.B. über `Var2` manipuliert wird, jegliche Änderung des Objekts auch für `Var1` gilt, da `Var1` und `Var2` ja auf das identische Objekt verweisen und nicht auf zwei verschiedene Objekte!

Wer schon mal in anderen Sprachen programmiert hat (z.B. C++), den wird dieses Prinzip an „Zeiger“ erinnern. Der große Unterschied ist, dass Java die von dort bekannten Zeigeroperationen nicht erlaubt.

Im Prinzip könnt ihr euch das so vorstellen, dass eine Referenz ein Wegweiser für den Prozessor ist, auf dem steht, an welcher Stelle er im Arbeitsspeicher nach dem Objekt suchen soll.

8.1 Verweise ins Leere: Null-Referenz

Deklariert ihr eine Variable und initialisiert sie nicht (also weist ihr noch kein Objekt zu), existiert der Platz, in dem später die Adresse des Objekts stehen wird, bereits. Da die Variable aber noch auf kein Objekt zeigt, steht an dieser Stelle auch noch nichts. Dieses „nichts“ heißt in Java `null`.

Also: Unser Wegweiser für den Prozessor ist schon mit dem Namen des Objekts beschriftet, zeigt aber noch nirgendwohin.

Wurde einer Variable schon mal ein Objekt zugeordnet, wurde also bereits eine Speicheradresse in der Variable hinterlegt, und ihr wollt diese Referenz, aus welchen Gründen auch immer, löschen, funktioniert das auch wieder mit `null`:

```
1 Object var;          // Deklaration: leere Referenz erzeugen
2 var = new Object(); // neues Objekt im Speicher erzeugen
3                     // und seine Adresse in Var hinterlegen
4 var = null;         // Inhalt der Referenz löschen
```

Es gibt durchaus gute Gründe, eine Referenz wieder zu löschen, aber das gehört zu den Themen für Fortgeschrittene.

8.2 Postbotentalk: Referenzen vergleichen

Dieses Thema ist ein Fallstrick, über den auch erfahrene Programmierer noch stolpern: Vergleicht man zwei Variablen mit den Vergleichsoperatoren, vergleicht man nur die darin gespeicherten Referenzen, also die Speicheradressen!

Wie gesagt: man kann wunderbar Referenzen mit den Vergleichsoperatoren speichern. Wirklich sinnvoll ist allerdings nur der Einsatz von „ist gleich“ (`==`) und „ist ungleich“ (`!=`). Noch mal zum Mitschreiben: So werden NUR die REFERENZEN verglichen, nicht die Objekte. Sind die Objekte zwar identisch, allerdings an unterschiedlichen Stellen im Speicher abgelegt, gibt ein Vergleich also `false` zurück. Daher müsst ihr beim Vergleichen von Strings auch die Methode `equals()` verwenden.[23]

Demonstrieren wir das mal an einem Beispiel:

```
1  String st1 = new String("Foo");
2  String st2 = "Foo";
3  String st3 = st1;    //Referenz von st1 in st3 kopieren

4  boolean b1 = (st1 == st2);
5      // false: st1 und st2 weisem auf unterschiedliche Objekte,
6      //        daher ist ihre Referenz verschieden
7  boolean b2 = (st1 != st2);
8      // true:  Begründung siehe oben
9  boolean b3 = (st1.equals(st2));
10     // true:  Beide String-Objekte haben den gleichen Inhalt
11 boolean b4 = (st1 == st3);
12     // true:  st3 hat die gleiche Referenz wie st1,
13     //        da diese kopiert wurde
14 boolean b5 = (st1 == "blub");
15     // false: Hier wird ein neues Objekt vom Typ String
16     //        erzeugt, das logischerweise nicht die selbe
17     //        Speicheradresse hat
```

Es gibt in Java auch Wege, komplett eigenständige (flache) Kopien von Objekten anzulegen, aber auch das gehört bereits zu den Themen für Fortgeschrittene und soll aus diesem Grund an dieser Stelle nicht weiter behandelt werden.

9 UML - Unified Modeling Language

Kommen wir zu einem Thema, das ihr primär aus einem Grund lernt: Damit ihr euch mit einem Programmierer unterhalten könnt. Nun unterhaltet ihr euch mit Programmierern nicht in Java-Code. Eventuell wird der Programmierer auch eine andere Programmiersprache verwenden wollen (z.B. C++). Um sich trotzdem verständigen zu können, gibt es UML. UML ist sehr weit verbreitet und eines der wichtigsten Kommunikationsmittel in der Informatik.

9.1 Das Klassendiagramm

UML kennt mehrere Diagrammtypen. Im folgenden werden wir aber nur auf einen Vertreter, das **Klassendiagramm**, eingehen. Wie bereits dargelegt: UML ist dafür da, mit anderen Program-

[23] Es gibt auch noch die Methode `String.compareTo()`. Diese kommt daher, dass `String` das Interface `Comparable` implementiert. Wie das funktioniert, haben wir in Kapitel 7.5 gezeigt.

mierern zu kommunizieren. Daher muss in einem Klassendiagramm nicht der komplette Code beschrieben sein mit allen Methoden und Attributen. In ein Klassendiagramm gehört nur das, was man angeben möchte und zum Verständnis der zu beschreibenden Funktionen notwendig ist. Wenn man jedoch etwas angibt, muss man sich unbedingt an die Konventionen halten.

9.1.1 Darstellung von Klassen

Ein Klassendiagramm besteht aus lauter Kästen, die jeweils eine Klasse beschreiben, und Linien, die diese Kästchen verbinden.

So ein Kasten besteht aus drei Zeilen, die jeweils durch einen Strich voneinander getrennt werden. In den Zellen steht

- oben: Der Klassenname **fett** gedruckt
- in der Mitte: Attribute mit Typ und Sichtbarkeit nach folgendem Schema:

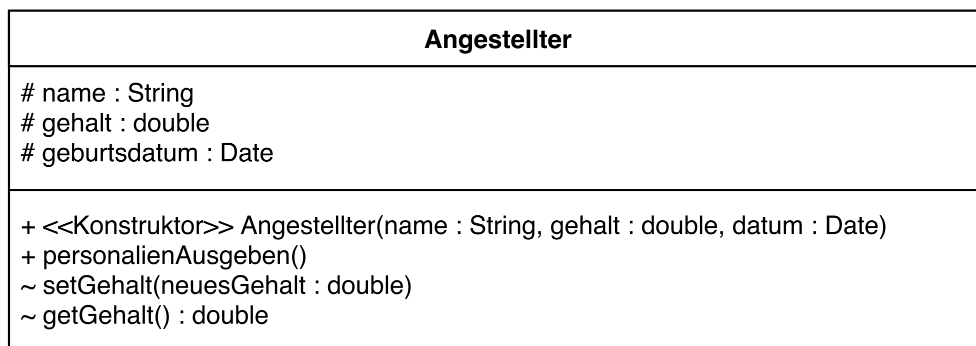
  Sichtbarkeitszeichen Attributname : Datentyp

- unten: Methoden mit Rückgabetyp, Sichtbarkeit und Parametern, sowie der Konstruktor, nach folgendem Schema:

  [«Konstruktor»] Sichtbarkeitszeichen Methodenname(Parameterliste) : Rückgabetyp

  Dabei werden bei einer leeren Parameterliste die Klammern trotzdem geschrieben, die Parameter selbst werden wieder mit Parametername[24] : Datentyp angegeben. Gibt eine Methode nichts zurück (`void`), wird dies nicht angegeben.

Das sieht dann ungefähr so aus (Beispiel nach unserer Klasse `Angestellter`):



Es gibt folgende Sichtbarkeitszeichen:

Symbol	Sichtbarkeit
+	`public`
#	`protected`
~	keine Angabe
-	`private`

Dazu kommen noch die Notationen für die Spezialfälle:

[24] In der Klausur reicht es, den Datentyp anzugeben.

- statische Variablen und Methoden werden unterstrichen
- abstrakte Methoden und Klassen schreibt man *kursiv*
- und Konstanten in GROSSBUCHSTABEN.

Ist eine Klasse abstrakt, schreibt man das in spitzen Klammern darüber und den Klassennamen ***fett und kursiv***.

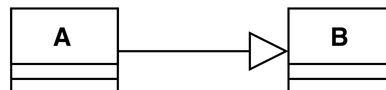
Bei Interfaces benutzt man ebenfalls eckige Klammern, ansonsten aber unformatierten Text.

9.1.2 Verknüpfungen

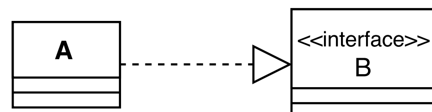
Nun stehen Klassen aber auch in Beziehung zueinander, die man in einem Klassendiagramm darstellen möchte. Das führt uns zu einem der heikelsten Kapitel in UML: Den Pfeilen.

Das wichtigste an diesen Pfeilen ist: Ihr müsst die Pfeile exakt so zeichnen, wie sie definiert sind, sonst bedeuten sie etwas anderes!

Vererbungspfeil Dabei handelt es sich um einen Pfeil mit geschlossener, nicht ausgemalter Spitze, die sich bei der Elternklasse befindet.



Implementierungspfeil Ganz ähnlich dazu ist der Implementierungspfeil für Interfaces. Dieser sieht genau so wie ein Vererbungspfeil aus, nur ist die Linie gestrichelt. Die Pfeilspitze befindet sich beim zu implementierenden Interface.

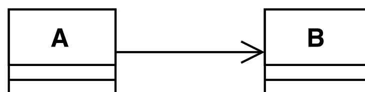


Bleiben noch die übrigen Beziehungen: die Assoziation, die Aggregation und die Komposition.

Assoziation Die Assoziation ist die einfachste und häufigste Beziehung zwischen zwei Klassen.

Bei der Assoziation haben beide Objekte unabhängige Lebenszyklen und es gibt keinen Eigentümer. Nehmen wir zwei Ehepartner: Ein Mensch kann einen anderen Menschen heiraten, sie können sich aber auch wieder trennen und existieren dann nach wie vor als eigenständige Objekte. Solch eine Verbindung kann ich eine Richtung, aber auch in beide Richtungen vorliegen.

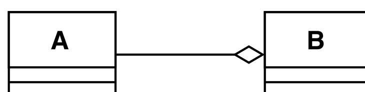
Der Pfeil dazu ist ein Pfeil mit offener Spitze beim assoziierten Objekt.



Geht die Beziehung in beide Richtungen (ist sie bidirektional), dann verwendet man keine Pfeilspitzen.

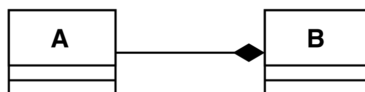
Aggregation Die Aggregation ist eine spezielle Form der Assoziation. Hier haben nach wie vor alle Objekte einen eigenen Lebenszyklus, aber in diesem Fall ist die Hierarchie beider Objekte klar definiert. Zum Beispiel gibt es kein Fahrzeug ohne Hinterrad (sonst mutiert es zum Stehzeug), aber das Hinterrad ist durchaus ein eigenständiges Objekt, das es auch gibt, wenn gerade kein Fahrzeug zur Hand ist. Auch sollte das Fahrrad die Drehung des Hinterrades steuern und nicht umgekehrt.

Die Aggregation wird dargestellt durch eine leere Raute beim übergeordneten Objekt.[25]

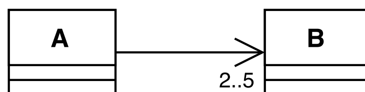


Komposition Eine Komposition dagegen ist es, wenn das zusammenfassende Ganze den Teil erst zu einem sinnvollen Objekt macht, daher „stirbt“ das untergeordnete Objekt, wenn das übergeordnete Objekt gelöscht wird. Ein Beispiel wäre ein Haus: Hier ist die Hierarchie zwischen Haus und Zimmer sehr eindeutig, und wird das Haus abgerissen, sind auch die Zimmer nicht mehr existent.

Die Komposition wird durch eine ausgefüllte Raute beim übergeordneten Objekt dargestellt.



Kardinalität Manchmal ist es auch wichtig, die Zahlenverhältnisse bei so einer Beziehung darzustellen. Zahlenverhältnisse nennt der Informatiker Kardinalität (in UML spricht man allerdings von **Multiplizität**). Wenn die Zahl genau festgelegt ist, dann schreibt ihr direkt die Zahl. Die Zahl steht bei der assoziierten Klasse. Ihr schreibt die Zahl über den Assoziationspfeil.
Liegt die Zahl dagegen nicht genau fest, müsst ihr das angeben, z.B. als Zahlenbereich. Die Notation ist die Gleiche wie bei Maple: Für einen Zahlenbereich von 2 bis 5 schreibt ihr `2..5`. Habt ihr ein Ende nicht explizit gegeben, schreibt ihr statt der Zahl ein n oder einen Stern: `1..n` bzw. `1..*`. Wenn die Zahl völlig egal ist, es also auch erlaubt wäre, wenn die Anzahl der assoziierten Objekte 0 ist, dann könnt ihr auch nur den Stern schreiben. Ein Stern ist also äquivalent zu `0..n` bzw. `0..*`. Komplett sieht ein Assoziationspfeil mit Kardinalität also so aus:



[25] An dieser Stelle sei erwähnt, dass die Aggregation in der eigentlichen UML-Spezifikation nicht genau definiert wird. Tatsächlich sind die abgrenzenden Kriterien für die Aggregation sehr vage und auch ein vieldiskutierter Punkt in der Literatur. Daher kommt dieser Pfeil in UML-Diagrammen selten vor; man verwendet lieber den allgemeineren Assoziations-Pfeil.

Auf ein komplettes Beispiel für ein UML-Diagramm verzichten wir an dieser Stelle, da in Übung und Vorlesung einige Beispiele behandelt werden. Auch im Internet gibt es viele gute Beispiele.[26] Darüber hinaus seien auch die Bücher in der Lehrbuchsammlung Informatik der Bibliothek empfohlen, die ebenfalls gute Beispiele enthalten und das Thema UML noch einmal weit umfassender behandeln, als das im Rahmen der Lehrveranstaltung möglich wäre.

9.2 Vorsicht Kleinkram!

Bei den UML-Diagrammen müsst ihr aufpassen wie ein Luchs: Es geht dabei wirklich um Kleinigkeiten. Ein Pfeil mit geschlossener Spitze hat eine andere Bedeutung als ein offener, einer mit einer ausgefüllten Raute eine andere als einer mit einer leeren, einer mit gestricheltem Hals eine andere als einer mit einem durchgezogenen Hals usw... Haltet also die Augen offen und merkt euch die kleinen Unterschiede gut, denn gerade diese Winzigkeiten können in der Klausur aberwitzig viele Punkte kosten.

10 Die Datenkraken: Collections

Bis jetzt habt ihr für jedes Objekt, das ihr erstellt habt und auch weiter verwenden wolltet, eine eigene Variable deklariert. Bei einer geringen Anzahl von Objekten ist das kein Problem. Aber mal angenommen, ihr möchtet für das Präsidium der Uni ein Programm schreiben, um sämtliche Studenten zu verwalten. Das wäre bei mehr als 20.000 Studenten eine richtige Sisyphusarbeit. Außerdem wäre das Programm äußerst unflexibel, weil ihr, jedes Mal, wenn sich die Anzahl der Studenten ändert, dass Programm umschreiben müsstet (Variablen löschen oder neue deklarieren). Das erste Problem, also die große Anzahl der einzuführenden Variablen, solltet ihr mit eurem bisherigen Wissen schon lösen können: Wir führen einfach einen Array ein, in dem wir alle Studenten speichern. Bliebe aber das zweite Problem: die sich dauernd ändernde Anzahl an Studenten. Da ein Array mit einer festen Größe initialisiert wird und sich diese nicht mehr ändern lässt, müsstet ihr ständig Referenzen zwischen Arrays hin- und herkopieren, was ziemlich fehleranfällig ist.

Um's kurz zu machen: die Lösung für unser Problem heißt **Collections**. Doch was sind Collections? Bildlich gesprochen ist eine Collection ein großer Sack für beliebig viele andere Objekte beliebigen Typs, mit ein paar Werkzeugen, um sie zu verwalten.

10.1 Einführung

Collection ist einfach der Überbegriff und stammt vom gleichnamigen Interface, das fast alle Collections direkt oder indirekt implementieren. In Java gibt es einen ganzen Haufen von Collections: Im Prinzip sind das also nichts mehr als Objekte, die Referenzen auf andere Objekte in sich aufnehmen können. Die Unterschiede zwischen den einzelnen Typen bestehen einfach darin, wie die Objekte gespeichert und wie mit ihnen umgegangen wird. Je nach Anwendungsfall sollte man also die Collection bedacht wählen. Wir stellen euch hier `Stack` und `Queue` vor, andere wichtige Typen sind `Vector`, `LinkedList` und `ArrayList`.

Prinzipiell gibt es zwei Typen von Collection, also Sammlungen von Objekten: Indexbasierte und solche, bei denen kein Zugriff auf den Index möglich ist. Beide haben ihre Vor- und Nachteile. Die, die keine Zugriffe auf den Index eines Elements ermöglichen, implementieren `Collection` direkt, die, die ihn erlauben, implementieren `List`, was wiederum eine Erweiterung von `Collection`

[26] Achtung: Viele Online-Beispiele stimmen nicht komplett mit der UML-Norm überein.

ist. Zum ersten Typ gehört z.B. `Queue`, zum zweiten Typ `ArrayList`, `LinkedList`, `Vector` oder `Stack`[27]. Spätestens, wenn ihr Collections verwendet, solltet ihr angesichts der Vielfalt von Auswahlmöglichkeiten angewöhnen, mit der Java-API zu arbeiten.

Ein Collection-Objekt legt ihr fast genauso an wie ein Objekt eines jeden anderen Typs. Nur wird eine Collection meist typisiert, d.h. es wird noch angegeben, was für Objekte sie später überhaupt enthalten darf (hier gelten natürlich auch die Grundsätze der Polymorphie). Um einer Collection Objekte zu übergeben ruft ihr die entsprechende Methode (meist `add()`) des Collection-Objekts auf und übergebt ihr das zu speichernde Objekt.

`Student` ist im folgenden Beispiel als bereits geschriebene Klasse anzusehen.

```
1        //Collection zum Speichern aller Studenten erzeugen:
2        ArrayList<Student> immatrikulierte = new ArrayList<Student>();

3        Student stud = new Student("Peter"); //Student erzeugen

4        //erzeugten Student der Collection hinzufügen:
5        immatrikulierte.add(stud);

6        //oder kürzer:
7        immatrikulierte.add(new Student("Roland"));
```

10.2 Generische Datentypen

Endlich mal etwas relativ einfaches. Generische Datentypen, auch „Java generics“ oder „parametrisierte Datentypen“ genannt, erlauben es dem Programmierer eine Klasse zu schreiben, ohne sich bezüglich der in der Klasse verwendeten Datentypen festlegen zu müssen. Dies überlässt der Autor der Klasse demjenigen, der diese später einsetzt und verwendet deshalb stattdessen einen Platzhalter (meist einfach `T` oder `E`). Gibt der Verwender der Klasse keinen Datentyp an, wird der Platzhalter automatisch auf `Object` gestellt (Erinnerung: `Object` ist die Basisklasse in Java, d.h. alle anderen Klassen sind auch `Objects`).

```
1  public class Schachtel<T> {
2      private T inhalt;

3      void hineinlegen(T dasObjekt) {
4          inhalt = dasObjekt;
5      }
6      T herausnehmen() {
7          T tmp = inhalt; // Refernez zwischenspeichern
8          inhalt = null;    // Referenz von "inhalt" löschen
9          return tmp;
10     }
11 }
```

[27] Die Klasse Stack ist so ziemlich die krasseste Design-Fehlentscheidung im Java-Framework. Im Gegensatz zu den anderen Listen ist der Stack kein Interface, was erlauben würde, die abstrakte Stack-Funktionalität für unterschiedliche Anwendungsfälle zu spezialisieren, außerdem implementiert Stack `List` und damit indexverwendende Methoden, die in einem Stack nach dem LIFO-Prinzip eigentlich nichts zu suchen haben.

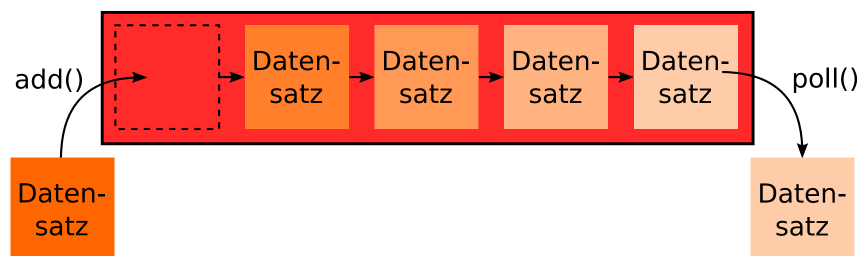
Wie ihr seht, ist `T` hier ein Platzhalter für einen beliebigen Datentyp und wird einfach anstelle eine expliziten Datentyps hingeschrieben. Praktisch ist auch, dass man angeben kann, dass dieser Platzhalter nur Objekte eines bestimmten Datentyps - oder noch besser, mehrerer Datentypen - annimmt. Dieses Prinzip funktioniert genauso mit Interfaces oder abstrakten Klassen. Natürlich ist es auch möglich, eine Klasse mit zwei Platzhaltern zu parametrisieren (`<T, E>`), darauf stößt man allerdings eher selten[28].

⚠ Wichtig beim Schreiben einer generischen Klasse ist, dass ihr den Platzhalter nicht einfach castet und dann typspezifische Methoden verwendet, denn dann käme es zu Fehlermeldung, wenn ein „falscher“ Datentyp festgelegt wird. Wollt ihr dies machen, müsst vor den Typ der Objekte überprüfen und entsprechend casten.

10.3 Die Warteschlange: Queue

Die Queue - zu deutsch „Warteschlange“- ist in der Informatik eine spezielle Datenstruktur, die andere Daten in sich aufnehmen kann. In Java ist `Queue` ein Interface und wird von diversen Klassen implementiert (wer es genauer wissen will, dem sei ein Blick in die Java-API empfohlen). Dieses Interface definiert Methoden, die eine Warteschlange haben muss - das sind mindestens mal „Objekt am Ende der Warteschlange hinzufügen“ und „Objekt am Anfang der Warteschlange entfernen“.

Deutlich wird das vielleicht an folgendem Bild:



Stellt euch also einfach eine perfekte Warteschlange an einer Kasse vor, in der keiner seitlich weggeht und keiner drängelt: Hinten stellen sich neue Leute an und der jeweils Vorderste wird bedient. Die Queue ist daher ein **FIFO-Speicher** (first in, first out): Wer als erstes da war, ist auch als erstes dran.

Eine Queue ist ein generisches Interface, genau wie das von ihr erweiterte Collection-Interface. Sie definiert folgende Methoden:

- `boolean add(E element)` - fügt am Ende einer Schlage das übergebene Objekt ein und gibt im Normalfall `true` zurück
- `E peek()` - liefert das vorderste Objekt der Schlange zurück
- `E poll()` - liefert das vorderste Objekt der Schlange zurück und entfernt es aus dieser

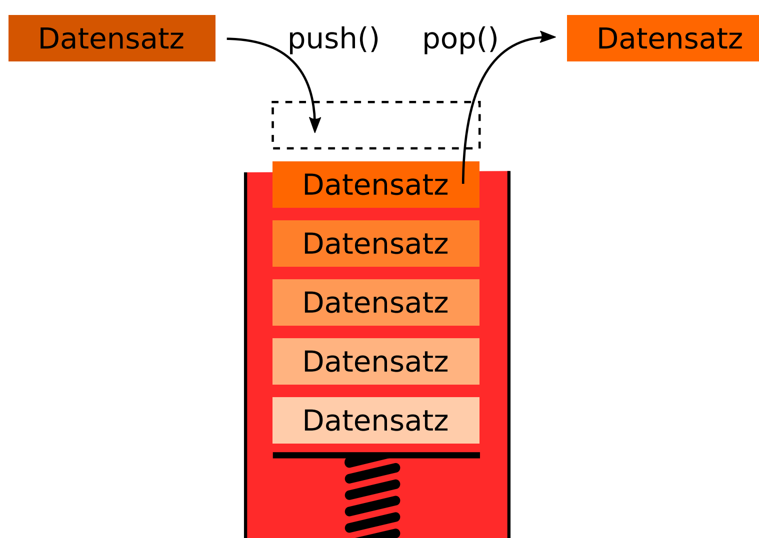
Wenn ihr eine Queue verwendet, empfehlen sich dafür die Klassen `ConcurrentLinkedQueue` oder `ArrayDeque` (sprich: „Deck“), die beide die genannten Methoden implementieren.

[28] Ein Beispiel wäre ein Dictionary, das praktisch eine Liste von Tupeln ist, wovon der erste Wert ein Schlüssel (key) ist und der zweite ein Wert (value). Genau wie ein echtes Wörterbuch darf kein Schlüssel mehrfach vorkommen. Dabei können Schlüssel und Wert unterschiedliche Datentypen haben.

10.4 Sardinen in der Dose: Stacks

Der Stack (engl. Stapel oder **Kellerspeicher**), funktioniert deutlich anders als eine Warteschlange. Stellt euch den Stapel wie den Tellerstapel in der Mensa vor, der so gefedert ist, dass der oberste Teller immer mit der Oberkante des Behälters abschließt. Genauso funktioniert dieser Stapel: Die Datensätze werden oben draufgelegt und auch oben wieder weggenommen. Daher nennt man dies auch **LIFO-Prinzip** (last in, first out - es wird zuerst der Datensatz entnommen, der als letztes hinzugefügt wurde). Wie in dem Mensatellerstapel ist es nicht möglich, das unterste Element zu entnehmen.

Dieses Bild soll das Prinzip des Stapels veranschaulichen:



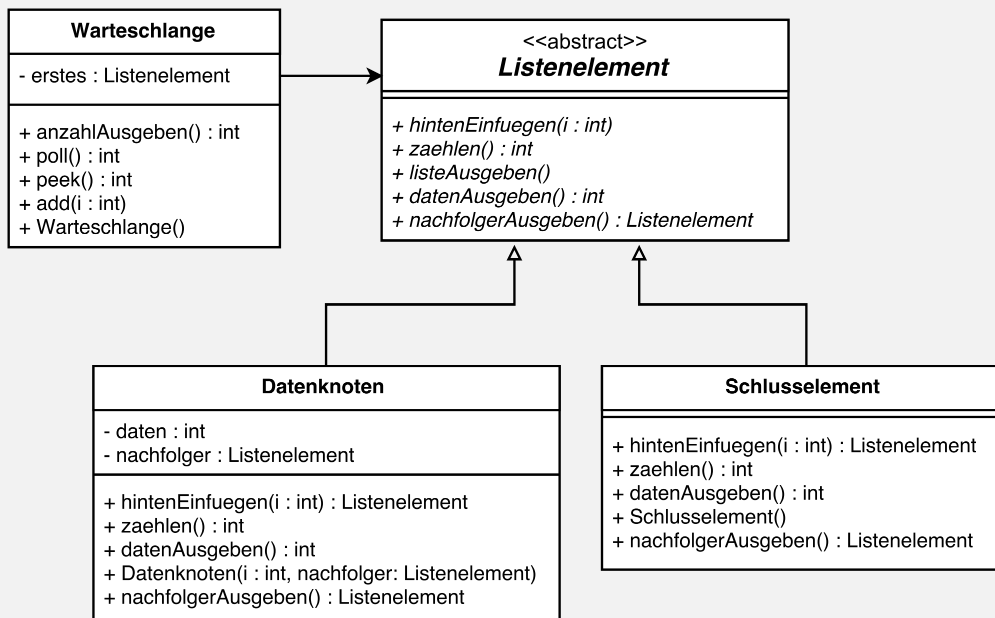
Ein Beispiel für die Implementierung von Stacks in Java, die dem hier beschriebenen Prinzip grob entspricht, wäre zum Beispiel die Klasse `Stack`; seit Java 7 wird stattdessen die Verwendung einer der vom Interface `Deque` abgeleiteten Klassen empfohlen, auch wenn diese mehr als nur die Stack-Funktionalität bieten und die Methoden etwas anders heißen.
Um mit Stacks arbeiten zu können, braucht ihr primär folgende Methoden:

- `push(E element)` - Legt das übergebene Objekt auf den Stapel
- `E peek()` - Liefert das oberste Objekt des Stapels zurück
- `E pop()` - Liefert das oberste Objekt des Stapels zurück und entfernt es aus diesem

Implementierung einer rekursiven Warteschlange

Vielleicht habt ihr euch gefragt, wie eine solche Collection eigentlich implementiert wird. Grob gesagt gibt es zwei Typen: die array-basierten Implementierungen und die rekursiven Implementierungen. Beide haben ihre Vor- und Nachteile und sind in manchen Anwendungsfällen jeweils schneller als die andere. Wir möchten hier exemplarisch die rekursive Liste besprechen; der Einfachheit halber sparen wir uns die Generics und nehmen an, dass unsere Liste den Typ `int` speichert.

Dieser Teil ist nicht klausurrelevant und nur zum Verständnis gedacht.



Die Grundidee ist erst einmal: Die Liste selbst kennt nur das erste Element, und jedes Element kennt seinen Nachfolger. Das ermöglicht uns recht einfaches Einfügen an einer beliebigen Stelle, da wir nur zwei Referenzen verändern müssen. Doch es gibt ein Problem: Das letzte Element der Liste hat keinen Nachfolger, wir müssten also bei allen Methoden, die auf den Nachfolger eines Elements zugreifen, zunächst überprüfen, ob es überhaupt einen Nachfolger hat! Das ist ziemlich ineffektiv.

An dieser Stelle verwenden wir das sogenannte **Kompositum**-Entwurfsmuster: Das erste Element und auch die Nachfolger haben den Datentyp `Listenelement`, dieser wird jedoch durch eine abstrakte Klasse beschrieben und hat zwei konkrete Implementierungen: `Datenelement` und `Schlusselement`. Jede Liste hat ein Schlusselement, das der Nachfolger des letzten Datenelements ist. Dieses ermöglicht es uns, die ganzen Überprüfungen wegzulassen. Schwer vorzustellen? Okay, ein paar Codeschnipsel.

```
1  // in Klasse Warteschlange
2  public Warteschlange() {
3      erstes = new Schlusselement();
4  }
5  public int poll() {
6      int zwischenspeicher = erstes.datenAusgeben();
7      erstes = erstes.nachfolgerAusgeben();
8      return zwischenspeicher;
9  } // peek analog dazu (nur ohne löschen)
10 public void add(int i) {
11     erstes = erstes.hintenEinfuegen(i);
12 }
13 public int anzahlAusgeben() {
14     return erstes.zaehlen();
15 }
```

Hier haben wir schon einen Vorteil der rekursiven Implementierung: Wollen wir das erste Element löschen, müssen wir nur die Referenz auf das erste Element verändern.

Aber die eigentlichen Tricks kommen erst in den beiden anderen Klassen (`DE` steht für `Datenelement`, `LE` für `Listenelement`)

```java
// in Klasse Datenelement
public DE(int i, LE nachfolger) {
    daten = i;
    this.nachfolger = nachfolger;
}

public LE hintenEinfuegen(int i) {
    nachfolger =
     nachfolger.hintenEinfuegen(i);
    return this;
}

public int zaehlen() {
    return nachfolger.zaehlen()+ 1;
}

public int datenAusgeben() {
    return daten;
}
// analog bei nachfolgerAusgeben()
```

```java
// in Klasse Schlusselement
public Schlusselement() {
    //tue nichts
}

public LE hintenEinfuegen(int i) {
    return new DE(i, this);
}

public int zaehlen() {
    return 0;
}

public int datenAusgeben() {
    // Exception werfen: Liste leer
}
// analog bei nachfolgerAusgeben()
```

Besonders schön sieht man das Prinzip beim Zählen: Rekursiv wird immer die Zählen-Methode des Nachfolgers aufgerufen. Wird jedoch die Zählen-Methode des Schlusselements aufgerufen, wird 0 zurück gegeben. Nun wird der Aufrufstapel rückwärts abgearbeitet, wobei in jedem Schritt 1 hinzugefügt wird. Kommt das Programm dann wieder beim Aufrufer (der Liste) an, erhält diese genau die Anzahl an Elemente.

Das Hinzufügen funktioniert ähnlich: Das hinzuzufügende Element wird immer weitergegeben, bis es beim Schlusselement ankommt. Dieses erstellt ein neues Datenelement, das als Nachfolger das Schlusselement hat (es soll ja am Ende sein). Das wird nun zurückgegeben und folglich als Nachfolger des vorherigen Elements gesetzt. Danach geben alle Elemente sich selbst zurück, so dass sich im vorderen Teil der Liste nichts verändert.

Euch wird aufgefallen sein, dass unser Konzept einen kleinen Haken hat: Soll der Nachfolger oder die Daten eines Listenelements abgefragt werden, muss dies nicht vorhanden sein, denn ein `Listenelement` kann ja auch ein `Schlusselement` sein. In diesem Fall soll eine Exception geworfen werden (mehr dazu im nächsten Kapitel). Dieser Fehler kann nämlich nur auftreten, wenn die Liste leer ist - da die fraglichen Methoden nur in der Liste selbst aufgerufen werden (bei `poll()` und analog bei `peek()`).

11 Mit Fehlern spielen: Exceptions

Zum Schluss dieses Kompendiums wollen wir nun noch zu einem Thema kommen, um das mal als Programmierer leider nicht herumkommt: Fehler. Was machen wir, wenn eine Methode in ihrer Ausführung feststellt, dass irgendetwas nicht so läuft, wie es sollte? Bei Fehlern in Java handelt es - ganz einfach - um Objekte. Verrückt, oder? Sogar Fehler sind Objekte!

Exceptions sind also Unregelmäßigkeiten im Programmablauf. Exceptions „treten z.B. auf“, wenn ihr versucht, einen `double`-Wert von der Konsole zu lesen, aber dort ein `String` eingetippt wurde. Oder wenn ihr durch null teilt, oder wenn ihr auf einen Index außerhalb des Arrays zugreifen wollt. Diese Ausnahmen gibt es schon. Ihr könnt Exceptions aber auch in euren Methoden benutzen, um dem Aufrufer mitzuteilen, dass seine Eingabewerte unzulässig sind. Ein Programmierer würde sagen, ihr „werft“ in eurer Methode eine Exception.

Das tolle an Exceptions ist: Dadurch, dass es sich dabei um ganz normale Objekte handelt (Datentyp `Exception` oder ein davon erbender Datentyp) könnt ihr euch jederzeit euren eigenen `Fehlertyp` basteln. Um genau zu sein, muss eure neue Exception noch nicht einmal von `Exception` erben - es reicht wenn sie von `Throwable` erbt (die Klassen `Error`[29] und `Exception` erben beide von `Throwable`).

Definieren wir uns doch einfach mal eine eigene Exception:

```
1  public class MyException extends Exception {
2      //in dieser Variable speichern wir den Zeitpunkt des Fehlers
3      private long ts; // (Unix-Timestamp -> Wikipedia)

4      //Konstruktor 1
5      public MyException(long ErrorTimeStamp) {
6          // Exception hat einen parameterlosen Konstruktor,
7          // daher wird super() automatisch aufgerufen
8          ts = ErrorTimeStamp;
9      }
10     //Konstruktor 2 (kann alternativ aufgerufen werden)
11     public MyException(String msg, long ErrorTimeStamp) {
12         super(msg);
13         ts = ErrorTimeStamp;
14     }

15     public long getTS(){
16         return ts;
17     }
18 }
```

Jetzt haben wir einen neuen Exception-Typ, der jedoch als zusätzliche Information den Zeitstempel des Fehlers in sich trägt.

11.1 Mit Fehlern um sich werfen: `throw`

Nun haben wir bereits geklärt, was Exceptions sind und wie wir uns neue Exception-Typen definieren können. Im Folgenden wollen wir uns anschauen, wie ihr selber Exceptions erzeugt, wenn euer Programm in eine Sackgasse gelaufen ist. Hierzu dient das Schlüsselwort `throw` gefolgt vom Aufruf des Konstruktors der Exception.

Angenommen wir befinden uns innerhalb einer Methode, die uns eine Division durchführt:

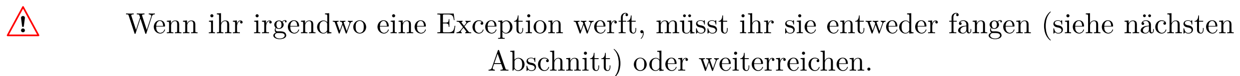
[29] Falls ihr euch fragt, was ein `Error` genau ist: Es handelt sich hierbei um einen Fehler, der innerhalb der JVM auftritt und nur in äußerst ungewöhnlichen Situationen auftreten sollte. Hab ich euch schon gesagt, das euer Computer nichts in einem Kernreaktor zu suchen hat?

```
{
    //double a,b werden der Methode als Parameter übergeben
    if(b == 0) {
        //Division durch 0 ist unzulässig
        throw new MyException("Division durch 0!",
            Instant.now().getEpochSeconds());
            // nur eine Java-Methode zum Auslesen der Unix-Zeit
    }
    return a / b;
}
```

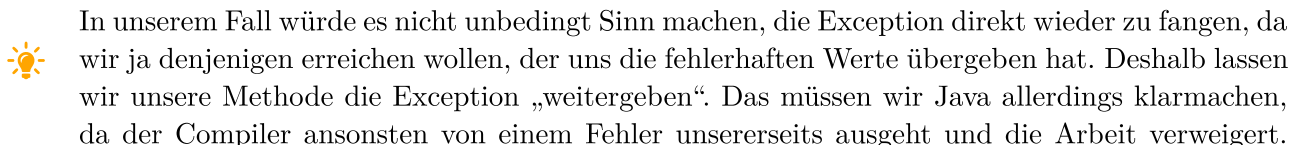
Normalerweise würde man sich für diese Anwendung eine Exception `DivisionDurchNullException` definieren und diese werfen, um den Code übersichtlicher zu halten. Aber das würde die Angelegenheit hier unnötig strecken.

11.2 Fehlerketten: `throws`

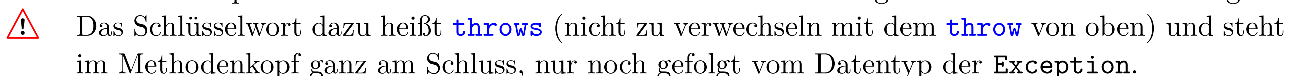
Euch ist jetzt garantiert aufgefallen, dass wir darauf verzichtet haben, den Methodenkopf hinzuschreiben. Warum? Ganz einfach: Hierfür müssen wir ein weiteres Schlüsselwort einführen: `throws`.



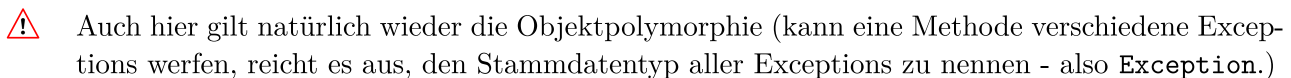
Wenn ihr irgendwo eine Exception werft, müsst ihr sie entweder fangen (siehe nächsten Abschnitt) oder weiterreichen.



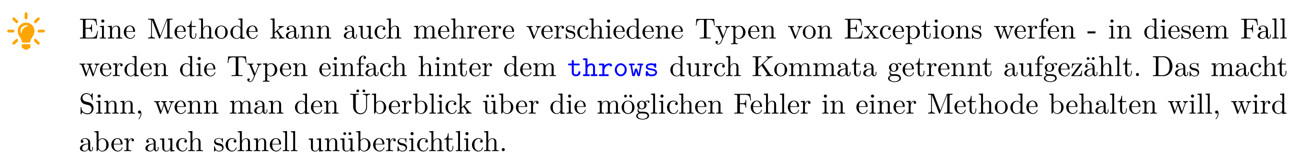
In unserem Fall würde es nicht unbedingt Sinn machen, die Exception direkt wieder zu fangen, da wir ja denjenigen erreichen wollen, der uns die fehlerhaften Werte übergeben hat. Deshalb lassen wir unsere Methode die Exception „weitergeben“. Das müssen wir Java allerdings klarmachen, da der Compiler ansonsten von einem Fehler unsererseits ausgeht und die Arbeit verweigert.



Das Schlüsselwort dazu heißt `throws` (nicht zu verwechseln mit dem `throw` von oben) und steht im Methodenkopf ganz am Schluss, nur noch gefolgt vom Datentyp der `Exception`.



Auch hier gilt natürlich wieder die Objektpolymorphie (kann eine Methode verschiedene Exceptions werfen, reicht es aus, den Stammdatentyp aller Exceptions zu nennen - also `Exception`.)



Eine Methode kann auch mehrere verschiedene Typen von Exceptions werfen - in diesem Fall werden die Typen einfach hinter dem `throws` durch Kommata getrennt aufgezählt. Das macht Sinn, wenn man den Überblick über die möglichen Fehler in einer Methode behalten will, wird aber auch schnell unübersichtlich.

Das Beispiel von oben, diesmal mit Methodenkopf:

```
public double dividieren(double a, double b) throws MyException {
    //die Methode von oben
}
```

Auch wenn ihr eine Methode verwendet, die eine Exception wirft, kann die aufrufende Methode die Exception wiederum nur weiterreichen.

```
public double rechnen() throws Exception {
    //Exception statt MyException geht dank Polymorphie
    //Methode (fehlerhaft) aufrufen
    return dividieren(3, 0);
}
```

Auf diese Weise lassen sich regelrechte „Ketten“ aufbauen - die Exception wird dabei einfach solange den Aufrufstapel entlang weitergegeben, bis sie irgendwo behandelt wird oder in der JVM landet. Das heißt aber: Wenn ihr in eurer main-Methode eine Methode aufruft, die eine Exception wirft, müsst ihr den Fehler dort ebenfalls behandeln oder weitergeben.

11.3 Alles wird gut: mit Fehlern weiterarbeiten

Wenn eine Methode einen Fehler wirft, muss dieser entweder weitergegeben oder aufgefangen und nach Möglichkeit behoben werden. Den ersten Teil kennen wir jetzt schon, das geht mit `throws`. Für den zweiten Teil gibt es in Java drei verschiedene Schlüsselwörter, die, richtig kombiniert, diese Wirkung erzielen. Daher können wir auch nicht gleich ein vollständiges Beispiel geben, da wir diese Schlüsselwörter Stück für Stück einführen müssen.

11.3.1 Problemebereiche eingrenzen: `try`

Wenn ihr eine Exception werft, müsst ihr sie auch wieder, direkt oder indirekt, auffangen. Doch zuvor müsst ihr Java erstmal sagen, in welchem Block die JVM auf auftretende Exceptions aufpassen sollte. Das geschieht mit dem Schlüsselwort `try` gefolgt von einem Block in geschweiften Klammern. Ihr solltet aber beachten, dass, sobald innerhalb eines `try`-Blockes eine Exception geworfen wird, der `try`-Block abgebrochen und zum passenden `catch`-Block gesprungen wird (siehe nächstes Kapitel).

Im Normalfall werdet ihr Exceptions indirekt fangen:

```
1 try {
2     inDieserMethodeKannEinFehlerAuftreten();
3 }
```

Direkt geht auch, ist aber in den seltensten Fällen sinnvoll:[30]

```
1 try {
2     throw new Exception();
3 }
```

11.3.2 Der Torwart: Fehler auffangen mit `catch`

Jetzt haben wir Java bereits klar gemacht, dass irgendwo in unserem `try`-Block Exceptions geworfen werden können, aber noch nicht, was die JVM damit machen soll. Dazu existiert das Schlüsselwort `catch` gefolgt von einer Parameterliste `(Parameter p)` mit nur einem Parameter und einem Block. Als Parameter wird dem `catch`-Block, ähnlich einer Methode, eine Exception übergeben und als Referenz in einer Variable gespeichert.

Wichtig: Auf einen `try`-Block **muss** direkt im Anschluss mindestens ein `catch`- oder `finally`-Block folgen! Allerdings dürfen mehrere `catch-Blöcke` aneinandergereiht werden. Dies dient dem Fall, dass verschiedene Exceptiontypen eine unterschiedliche Behandlung erfahren sollen, gilt aber eigentlich als schlechter Stil. Schöner ist folgendes:

[30] Vielleicht kommt ihr auf total verrückte Ideen, was man damit alles anstellen könnte. Mir würde beispielsweise einfallen, dies als Ersatz für `break` oder `continue` zu verwenden. Aber bitte denkt daran: Exceptions zu werfen und zu fangen ist, relativ gesehen, verdammt langsam.

```
1  public static main(String[] args) {
2      try{
3          double result = dividieren(5, 0);
4          double result2 = rechnen();
5      }
6      catch(Exception e) {
7          if (e instanceof MyException) {
8              // Typen passen zusammen: e zu MyException casten
9              MyException m = (MyException) e;
10             // Fehlertext und Timestamp auf der Konsole ausgeben
11             System.err.println(m.getMessage());
12             System.err.println(m.getTS());
13         }
14         else
15         {
16             // für alle anderen Exceptions
17         }
18     }
19 }
```

Die Methode `rechnen` wirft per definitionem eine `Exception`. Würden wir nur Ausnahmen vom Typ `MyException` fangen, würde das nicht alle Fälle abdecken, was der Compiler nicht akzeptieren würde. Wollen wir bei einer `MyException` trotzdem auf den Zeitstempel zugreifen, müssen wir diese casten; damit das keine Probleme macht, sollten wir vorher noch die Typen vergleichen.

Wir haben gerade die Methode `System.err.println()` aufgerufen. Wie der Name vermuten lässt, verwendet man diese Methode, um Fehler auf der Konsole auszugeben. Diese Meldungen werden, sofern die benutze Konsole das unterstützt, mit roter Schriftfarbe hervorgehoben[31].

11.3.3 Endstation: `finally`

Wie erwähnt, wird beim Auftreten einer Exception der `try`-Block sofort verlassen und nach dem passenden `catch`-Block gesucht. Nun kann es durchaus vorkommen, dass zum Ende eurer Methode gewisse Abschlussarbeiten durchgeführt werden sollen, egal, ob ein Fehler aufgetreten ist oder nicht. Das könnte z.B. das Schließen einer Datenbankverbindung oder das Schließen eines Dateistreams sein. Diese Dinge schreibt man in einen sogenannten `finally`-Block[32].

Der `finally`-Block steht immer hinter der `try..catch`-Kombination, muss aber nicht deklariert werden. Dieser wird auf jeden Fall ausgeführt. Der `finally`-Block kann übrigens durchaus unterbrochen werden, wenn euer Code mitten im `finally`-Block eine Exception wirft - dann gilt unter Umständen das Gleiche wie für den try-Block. In diesem Fall ist es ratsam, innerhalb des `finally`-Blockes ein erneute Kombination von `try`- und `catch`-Block zu verwenden.

Ein Beispiel hierzu, das schon etwas weiter in das JDK hineinreicht:

[31] Der EJE kann das leider nicht.

[32] Jetzt könntet ihr sagen: Warum schreibt man den fraglichen Code nicht einfach hinter den `try..catch`-Block? Ganz einfach: Ein `finally`-Block wird auch dann ausgeführt, wenn der `catch`-Block eine Exception wirft oder in ihm ein `return` oder `break` aufgerufen wird.

```
import java.io.*;

public class FinallyTest {
    public static void main(String[] args) {
        try {
            // Wenn hier ein Fehler auftritt, hat der nichts mit dem
            // finally-Block zu tun!
            BufferedReader reader
                = new BufferedReader(new FileReader("test.txt"));
            try {
                String zeile = null;
                while ((zeile = reader.readLine()) != null) {
                    // Zeile verarbeiten
                }
            }
            finally {
                // reader wurde bereits initialisiert, daher nicht null
                // sollte trotzdem ein Fehler auftreten, wird der vom
                // umgebenden catch gefangen
                reader.close();
            }
        }
        catch(IOException ex){
            System.err.println(ex.getMessage());
        }
    }
}
```

Unchecked Exceptions

Im letzten Kapitel habt ihr gelernt, dass, wenn eine Exception geworfen wird, diese verarbeitet oder weitergereicht werden muss, auch durch die Main-Methode hindurch. Das ist richtig, gilt jedoch nicht für alle Exceptions. Das habt ihr sicher schon gemerkt, der folgende Code kompiliert schließlich einwandfrei:

```
public static void main(String[] args) {
    int[] array = {1, 2, 3, 5};
    array[10] = 10 / 0;
}
```

Dabei treten in der dritten Zeile gleich zwei Fehler auf: Wir teilen durch 0, zudem greifen wir auf einen Index außerhalb des Arrays zu. Korrekterweise wird der Fehler auf der Konsole ausgegeben.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Diese Fehler landen also in der JVM, ohne, dass sie weitergegeben werden müssen. Solche Fehler - auch `RuntimeException`, eine Unterklasse von `Exception`, genannt, können also in Methoden auftreten, auch wenn ihre Signatur das nicht direkt verrät. Der Aufrufer

muss sich also auch nicht unbedingt um Exception Handling kümmern. Daher nennt man diese Exceptions auch **unchecked**.

Warum haben die Designer von Java diese Unterscheidung eingebaut? Ganz einfach: Ungeprüfte Ausnahmen, wie das durch 0 teilen, kann der Aufrufer vermeiden und vorhersehen. Eine Deklaration dieser macht daher nicht so viel Sinn. Geprüfte Ausnahmen, wie beispielsweise `IOExceptions`, sind daher schlecht vorhersehbar - woher soll das Programm auch wissen, ob eine Datei defekt ist, ohne das Öffnen wenigstens zu versuchen.

Es ist daher auch eher unüblich, derartige Exceptions abzufangen, auch wenn dies möglich ist. Denn: Treten diese Fehler auf, so liegt das an einem logischen Fehler des Entwicklers. Diese sollten also eher an der Wurzel des Übels mit `if`-Verzweigungen behandelt werden, statt nur die Symptome zu behandeln.

Wollen wir bei einer eigenen Exception-Klasse entscheiden, ob diese checked oder unchecked sein soll, gilt folgende Faustregel:

Beschreibt die Ausnahme einen Denkfehler des Aufrufers, sollte sie unchecked sein. Beschreibt sie eine Situation, aus der sich der Aufrufer durch geschicktes Error-Handling „retten“ kann, sollte sie checked sein.

12 Nachwort

So, wir hoffen, dass wir euch das Programmieren in Java ein wenig näher bringen konnten. Und immer dran denken:

> *A programmer is just a tool that converts coffee to code.*

In diesem Sinne: Happy Coding!

13 Glossar

13.1 Die wichtigsten Schlüsselwörter

Es folgt eine Auflistung der für euch wichtigsten Schlüsselwörter in Java.

`abstract`	Kennzeichnet eine Methode / Klasse als abstrakt. Abstrakte Klassen können nicht instanziiert werden, abstrakte Methoden müssen in nicht abstrakten Subklassen überschrieben werden. → 7.4.2, 7.5
`boolean`	Datentyp für einen booleschen Wert, auch Wahrheitswert genannt → 2.1
`break`	Verlässt eine Schleife oder einen `switch`-Block endgültig → 4.4.1
`byte`	Datentyp für ganzzahlige Werte, Wertebereich: $-2^7..(2^7-1)$ → 2.1
`catch`	Fängt innerhalb eines `try`-Blockes aufgetretene Exceptions → 11.3.2
`char`	Datentyp für ein Zeichen (Unicode-kodiert) → 2.1
`class`	Definiert eine Klasse und damit einen neuen Datentyp → 7.3
`continue`	Verlässt die aktuelle Iteration einer Schleife und springt zur nächsten → 4.4.2
`do`	Leitet den Codeblock einer do-while-Schleife ein. Nach diesem muss `while` mit einer Bedingung folgen → 4.2
`double`	Datentyp für Fließkommazahlen mit größtmöglichem Wertebereich. → 2.1
`else`	Wird ausgeführt, sollte keine vorangegangene Bedingung einer Verzweigung zugetroffen haben → 3.2
`extends`	Kennzeichnet eine Vererbung und erweitert damit den Datentyp mit allen Konsequenzen → 7.4
`false`	Ein möglicher boolescher Wert (siehe auch: `true`, `boolean`)
`final`	Deklariert eine nicht änderbare Variable → 1.3, eine nicht ableitbare Klasse → 7.4 oder eine nicht überschreibbare Methode → 7.4.1
`finally`	Definiert einen stets auszuführenden Block, unabhängig davon, ob im vorangehenden `try..catch`-Block Fehler aufgetreten sind → 11.3.3
`float`	Datentyp für einen Fließkommawert, kleiner Wertebereich als `double` → 2.1
`for`	Leitet eine spezielle Schleife ein. Wird von Iteratorvariable, Bedingung und Inkrementanweisung gefolgt. → 4.3
`if`	Definiert einen bedingungsabhängigen Codeblock → 3.1
`implements`	Kennzeichnet das Implementieren eines Interfaces → 7.5
`import`	Ermöglicht den Zugriff auf Klassen/Interfaces in einem anderen Paket, z.B. `java.util.*` für Collections
`instanceof`	Gibt true zurück, wenn sich das Objekt vor dem Schlüsselwort in den Datentyp nach dem Schlüsselwort casten lässt, sonst false. → 7.5
`int`	Datentyp für Ganzzahlen mit 4 Byte, Wertebereich: $-2^{31}..(2^{31}-1)$. Ganzzahlliterale werden ohne weitere Angabe als `int` interpretiert.→ 2.1
`interface`	Definiert ein Interface *vgl. Kapitel 7.5*
`long`	Datentyp für Ganzzahlen, Wertebereich: $-2^{63}..(2^{63}-1)$→ 2.1
`new`	Gibt ein neues Objekt zurück und erwartet einen Konstruktoraufruf. → 7.3.2
`null`	Stellt eine leere Referenz dar → 8.1

`package`	Kann um die Definition einer Klasse gelegt werden, um die Zugehörigkeit zu einem Paket anzugeben.
`private`	Sichtbarkeitsmodifizierer: nur innerhalb der Klasse sichtbar → 6.1
`protected`	Sichtbarkeitsmodifizierer: innerhalb dieser und davon erbenden Klassen sichtbar → 6.1
`public`	Sichtbarkeitsmodifizierer: global sichtbar → 6.1
`return`	Beendet eine Methode und kann dabei einen Wert zurückgeben. Erwartet den Datentyp, der in der Methodendeklaration angegeben wurde → 6.2
`short`	Datentyp für Ganzzahlen, Wertebereich: $-2^{15}..(2^{15}-1)$ → 2.1
`static`	Definiert eine statische Methode oder ein statisches Attribut. Diese gehören zur Klasse und sind unabhängig von Instanzen → 1.4, 7.3.5, 6.5
`super`	Stellt den Bezug zu einer Elternklasse her → 7.4
`synchronized`	Macht eine Methode threadsicher → 6.5
`this`	Verweist auf das aktuelle Objekt, u.a. hilfreich bei Verdeckung → 7.3.6
`throw`	Erzeugt in Kombination mit `new` ein neues Exception-Objekt → 11.1
`throws`	Weist Methoden an, eventuelle Exceptions an den Aufrufer weiterzuleiten Zwingt Methoden eventuelle Exceptions an den Aufrufer weiterzuleiten → 11.2
`true`	Ein möglicher boolescher Wert (siehe auch: `false`, `boolean`)
`try`	Definiert einen Block, der auftretende Exceptions auffängt → 11.3.1
`void`	Kennzeichnet eine Methode ohne Rückgabewert → 6.2
`while`	Leitet eine kopfgesteuerte Schleife ein → 4.1

13.2 Operatoren

Im Dokument wurden teilweise Operatoren verwendet, ohne sie ordentlich einzuführen. Diese Tabelle gibt zumindest einen Überblick darüber, mit welcher Priorität die Operatoren behandelt werden. Steht ein Operator in dieser Tabelle weiter oben als ein anderer, wird er zuerst ausgewertet. Grundsätzlich wird immer (außer bei Zuweisungen) von links nach rechts ausgewertet.

Fachbegriff	**Operatoren und Erklärung**	
Postfix	`expr++`	erhöht den vorangehenden Ausdruck um 1, **nachdem** der Ausdruck ausgewertet wurde ausgeführt wurde
	`expr--`	erniedrigt den vorangehenden Ausdruck um 1, nachdem der Ausdruck ausgewertet wurde ausgeführt wurde
Unäre Operatoren	`++expr`	erhöht den nachstehenden Ausdruck um 1 vor der Auswertung
	`--expr`	verringert den nachstehenden Ausdruck um 1 vor der Auswertung
	`+expr`	indiziert ein positives Vorzeichen
	`-expr`	kehrt das Vorzeichen des nachstehenden Ausdrucks um
	`~expr`	invertiert den nachstehenden Ausdruck bitweise
	`!expr`	invertiert den nachstehenden logischen Ausdruck

<table>
<tr><td rowspan="2">Multiplikativ</td><td>a*b, a/b</td><td>multipliziert bzw. dividiert zwei Zahlen</td></tr>
<tr><td>a % b</td><td>berechnet den Rest einer Division (Modulo-Operator)</td></tr>
<tr><td rowspan="2">Additiv</td><td>a + b</td><td>addiert zwei Zahlen bzw. verbindet zwei Strings</td></tr>
<tr><td>a - b</td><td>berechnet die Differenz zweier Zahlen</td></tr>
<tr><td rowspan="3">Bitschiebe-Operatoren</td><td>a >> b</td><td>schiebt die Bits von a um b Stellen nach rechts, wobei das Vorzeichen erhalten bleibt</td></tr>
<tr><td>a << b</td><td>schiebt die Bits von a um b Stellen nach links, wobei das Vorzeichen erhalten bleibt</td></tr>
<tr><td>a >>> b</td><td>schiebt die Bits von a um b Stellen nach rechts, wobei auch das Vorzeichen mitverschoben wird</td></tr>
<tr><td rowspan="5">Vergleich</td><td>a < b</td><td>Ist a kleiner als b, wird true zurückgegeben, sonst false.</td></tr>
<tr><td>a > b</td><td>Ist a größer als b, wird true zurückgegeben, sonst false.</td></tr>
<tr><td>a <= b</td><td>Ist a kleiner gleich b, wird true zurückgegeben, sonst false.</td></tr>
<tr><td>a >= b</td><td>Ist a größer gleich b, wird true zurückgegeben, sonst false.</td></tr>
<tr><td>a instanceof b</td><td>überprüft die Möglichkeit, a nach b zu casten → 7.5</td></tr>
<tr><td rowspan="2">Gleichheit</td><td>a == b</td><td>Istgleich</td></tr>
<tr><td>a != b</td><td>Ungleich</td></tr>
<tr><td>Bitweises AND</td><td>a & b</td><td>Führt die Und-Operation auf alle Stellen von a und b aus.</td></tr>
<tr><td>Bitweises OR</td><td>a | b</td><td>Führt die Oder-Operation auf alle Stellen von a und b aus.</td></tr>
<tr><td>Bitweises XOR</td><td>a ^ b</td><td>Führt die Exklusives-Oder-Operation auf alle Stellen von a und b aus.</td></tr>
<tr><td>Logisches AND</td><td>a && b</td><td>Und-Operation zweier logischer Ausdrücke, dabei wird b aber nur ausgewertet, wenn a wahr ist (sonst ist das Ergebnis ja sowieso false).</td></tr>
<tr><td>Logisches OR</td><td>a || b</td><td>Oder-Operation zweier logischer Ausdrücke, dabei wird der b aber nur ausgewertet, wenn a falsch ist (sonst ist das Ergebnis ja sowieso true).</td></tr>
<tr><td>Ternärer Operator</td><td colspan="2">(Bedingung) ? Ausdruck1 : Ausdruck2 → 3.3</td></tr>
<tr><td rowspan="2">Zuweisung</td><td>a = b</td><td>weist den Ausdruck b der Variable a zu</td></tr>
<tr><td></td><td>+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>= und >>>= sind Kurzversionen von Zuweisungen.
Beispiel: a += b entspricht a = a + b. Ansonsten haben die Zeichen die gleiche Bedeutung wie bei den anderen Operatoren.</td></tr>
</table>

Auch wenn man sich auf die Reihenfolge dieser Tabelle verlassen kann, ist es aus Gründen der Lesbarkeit sinnvoll, bei allen über Punkt vor Strich hinausgehenden Reihenfolgen Klammern zu setzen.