Funktionale und objektorientierte Programmierkonzepte Version 2.0

Fabian Damken (fabian.damken@stud.tu-darmstadt.de)

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1 Einführung

Dieses Skript ist Vorlesungs- und Übungsbegleitend zu der Veranstaltung „Funktionale und objektorientierte Programmierkonzepte“ zu verstehen. Es wird im Laufe des Semesters mit weiteren Kapiteln ergänzt werden, in dieser ersten Version ist nur der gesamte funktionale Teil zu Racket vollständig vorhanden.

Ich wünsche viel Spaß [1] beim Lesen und viel Erfolg für die Klausur!

1.1 Aufbau

Dieses Skript ist in folgende Kapitel gegliedert:

2 Abstrakte Konzepte
In diesem Kapitel werden abstrakte Konzepte der Programmierung eingeführt, d.h. es wird über keine Programmiersprache an sich gesprochen.

3 Racket
Dieses Kapitel führt in die funktionale (2.1.3) Programmiersprache Racket ein, indem die im vorherigen Kapitel eingeführten Konzepte auf die Sprache angewendet werden.

4 Java
Ebenso wir im Kapitel über Racket, nur werden hier die Konzepte auf die objektorientierte (2.1.4) und imperative (2.1.3) Programmiersprache Java angewendet.

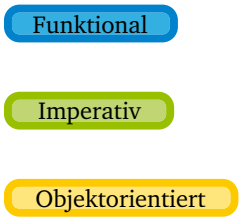
?? Abstraktion
Dieses Kapitel behandelt unterschiedliche Methodiken der Abstraktion, wie sie in funktionaler und objektorientierter Programmierung eingesetzt werden.

[1] Spaß, Spaß, Spaß

2 Abstrakte Konzepte

Dieses Kapitel führt ein in die abstrakten Konzepte, welche hinter eine Programmiersprache stehen.

Da sich nicht alle Konzepte auf alle Programmierparadigmen [1] anwenden lassen, ist jeder Abschnitt mit



gekennzeichnet, je nachdem, auf welches Paradigma sich das vorgestellte Konzept anwenden lässt. Die Markierungen werden am rechten Rand angebracht sein, sodass diese leicht zu finden sind.

2.1 Programmierparadigmen

2.1.1 Deklarativ

Bei der Deklarativen Programmierung steht die Beschreibung des Problems im Vordergrund, die Lösung wird hier meist automatisiert gefunden.

Es steht somit im Vordergrund, *welches* Problem gelöst werden soll und nicht *wie* ein Problem gelöst werden soll. Hierdurch ist eine genaue Trennung von Problem und Implementierung möglich, was bei imperativen Programmiersprachen (2.1.3) gar nicht oder zumindest nicht trivial möglich ist.

Das Paradigma der deklarativen Programmierung kann in weitere unterteilt werden, beispielsweise in funktionale (2.1.2) und logische Sprachen. Logische Sprachen werden hier nicht weiter ausgeführt.

Beispiele

- SQL, Cypher (Abfragesprachen)
- Lisp, Racket, Haskell (Funktionale Sprachen)
- Prolog (Logische Sprache)

2.1.2 Funktional

Funktionale Programmiersprache sind Ausarbeitungen von deklarativen Sprachen (2.1.1), bei denen ebenfalls die Beschreibung des Problems im Vordergrund steht. Sie werden oftmals zur Beschreibung von mathematischen Problem verwendet.

In diesen Sprachen wir auf Konstrukte wie Schleifen (2.4.2) und Variablen (2.3.1) verzichtet, wodurch Seiteneffekte (beispielsweise das Überschreiben von Zustandsvariablen) verhindert werden und die Implementierung zur Lösung eines Problems robuster wird.

Zur Abgrenzung von funktionalen Sprachen zu imperativen Sprachen siehe 2.1.5.

[1] Siehe 2.1

Beispiele

- Lisp
- Racket
- Haskell

2.1.3 Imperativ

Imperative Programmiersprachen sehen vor, dass der Entwickler beschreibt, *wie* ein Problem zu lösen ist, wobei die Beschreibung des eigentlichen Problems (das „*Was*“) fallen gelassen wird. Ein Programm besteht „aus einer Folge von Anweisungen [...], die vorgeben, in welcher Reihenfolge was vom Computer getan werden soll“. [BJ05]

Im Gegensatz zu deklarativen und funktionalen Sprachen ist die Korrektheit eines Algorithmus weniger offensichtlich und es werden Kontrollstrukturen wie Schleifen (2.4.2) und Variablen (2.3.1) eingeführt.

Zur Abgrenzung von imperativen und funktionalen Sprachen siehe 2.1.5.

Beispiele

- Java
- C/C++
- Assembler

2.1.4 Objektorientiert

Bei der objektorientierten Programmierung (OOP) werden reale Strukturen, sogenannte Objekte in der Software abgebildet. Die Architektur der Software wird somit an bestehenden Systemen der Wirklichkeit abgebildet und erlaubt den meisten Entwickler*innen einen einfachen Zugang zu der Software, da die Wirklichkeit repräsentiert wird. Ein Programm besteht besteht aus Anweisungen, welche vorgeben, was der Computer abarbeiten soll und in welcher Reihenfolge. Somit sind (die meisten) objektorientierten Programmiersprachen ebenfalls imperativ (2.1.3).

Durch die Vermischung mit imperativen und funktionalen Paradigmen lassen sich objektorientierte Sprachen nicht hinreichend von ersteren Abgrenzen, da letztere meistens auch Teile der imperativen und funktionalen Paradigmen beinhalten.

Beispiele

- Java
- C++
- Python

2.1.5 Abgrenzung Funktional ⟷ Imperativ

Die Abgrenzung von funktionalen und imperativen Sprachen lässt sich am besten anhand eines Beispiels erläutern:

Gegeben sei das mathematische Problem, die Fakultät einer beliebigen natürlichen Zahl $n \in \mathbb{N}_0$ zu bestimmen. Mathematisch wird das Problem wie folgt rekursiv definiert:

$$n! = f(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot f(n-1) & \text{falls } n > 0 \end{cases}$$

In einer (fiktionalen) funktionalen Sprache kann das Problem folgendermaßen implementiert werden:

```
f(0) := 1
f(n) := n * f(n - 1)
```

Abbildung 2.1: Funktionale Implementierung der Fakultät

In einer (ebenfalls fiktionalen) imperativen Sprache kann das Problem folgendermaßen implementiert werden:

```
function f(n)
    num = 1
    for i in 1..n
        num = num * i
    endloop
endfunction
```

Abbildung 2.2: Imperative Implementierung der Fakultät

2.2 Lexikalische Bestandteile

Funktional
Imperativ
Objektorientiert

Der lexikalische Bestandteil einer Programmiersprache kann auch als „Lexikon“ der Programmiersprache aufgefasst werden, welches definiert, welche Zeichen und Wörter innerhalb eines Programms verwendet werden können und welche Bedeutung diese haben. Hier entspringt auch der Name „Lexikalische Bestandteile“.

2.2.1 Datentypen

Funktional
Imperativ
Objektorientiert

Wir schauen uns in diesem Kontext zuerst die Datentypen an, wodurch festgelegt wird, welche Art von Daten wir mit der Programmiersprache verarbeiten können.
Um eine Intuition dafür zu bekommen, was ein Datentyp ist folgendes Szenario: Ein Verkehrsverbund möchte Daten über seine Straßenbahnen ablegen, genauer die Start- und Endstation einer Straßenbahn („Hauptbahnhof“, bzw. „Willy-Brandt-Platz“), die Länge der Gesamtstrecke in Kilometern (ca. 2km), den Fahrkartenpreis (2,15 €) und einen Schalter, ob die Straßenbahn gerade in Bewegung ist oder nicht.
Auffällig ist, dass sich die Arten der Daten unterscheiden:

- die Start- und Endstation der Straßenbahn ist ein Text,
- die Länge der Strecke ist eine Zahl,
- der Fahrkartenpreis ist eine Kommazahl und
- der Schalter ist ein Wahrheitswert (wahr/falsch).

Hierbei haben wir schon drei typische Datentypen in der Programmierung gefunden:

- Zeichenketten („String“),
- Ganzzahlen („Integer“/„Int“),
- Kommazahlen (Fetskommazahlen, bei denen die Position des Kommas fest ist, oder Fließkommazahlen, bei denen sich das Komma an jeder Stelle befinden kann) („Float“) und
- Wahrheitswerte („Boolean“/„Bool“).

Auch werden häufig einzelne Zeichen („Character“/„Char“) verwendet, aus welchen ein String wiederum besteht.
Abschließend bedeutet dies, dass durch einen Datentyp beschrieben wird, welche Form (Typ) die abgelegten Daten haben und später, wie diese im Speicher abgelegt werden.

2.2.2 Literale

Funktional

Imperativ

Objektorientiert

Als „Literale“ werden Werte bezeichnet, welche nicht über Variablen oder Konstanten (siehe 2.3.1) verarbeitet werden, sondern die direkt im Quellcode des Programms stehen (das heißt „hardcoded“ (festgeschrieben) sind).
Wir behandeln die Syntax der Erstellung und Nutzung von Literalen hier nicht weiter, da sich dies von Sprache zu Sprache stark unterscheidet. Die spezielle Syntax für Racket und Java werden wir in jeweiligen Abschnitten behandeln.

Escape-Sequenzen

Schweifen wir nochmals kurz ab zu den Literalen und betrachten sogenannte „Escape-Sequenzen“. Eine Escape-Sequenz ist eine bestimmte Abfolge von Zeichen, die bei der Ausführung durch andere Zeichen ersetzt werden.
Beispiel: Bei einer Zeichenkette, die mit einfachen Hochkommata eingeschlossen wird, ist nicht zu unterscheiden ob `''''` einen oder zwei Zeichenketten darstellt. Meistens wird nun der *Backslash* (\) genutzt, um zu sagen „das nächste Zeichen ist kein Steuerzeichen, sondern ein Buchstabe“: `'\'\''`.
Nun ist klar, dass es sich um eine Zeichenkette handelt, die aus zwei Hochkommata besteht.
Ähnlich wird es mit Zeilenumbrüchen gehandhabt. Diese werden meistens als \n dargestellt (wobei das „n“ für „new line“ steht).
Die genauen Escape-Sequenzen und ob diese überhaupt nötig sind hängt natürlich von der Programmiersprache ab und wir schauen es uns somit erneut an.

2.2.3 Schlüsselwörter

Funktional

Imperativ

Objektorientiert

Schlüsselwörter („Keywords“) werden in Programmiersprachen dazu eingesetzt, bestimmte Funktionen zu kennzeichnen. Sie beschreiben ganz bestimmte Zeichenfolgen (meist nur Text), welche meist nicht in einen anderen Kontexten verwendet werden dürfen.
Die Funktionalität und die Verfügbarkeit von Schlüsselwörtern ist von der Programmiersprache abhängig und wir werden diese weiter in den entsprechenden Abschnitten im Racket und Java Teil betrachten.

2.2.4 Bezeichner

Funktional

Imperativ

Objektorientiert

Wie auch in der realen Welt müssen wir auch in der Programmierung Namen für die Elemente unseres Programms (beispielsweise Konstanten oder Variablen (siehe 2.3.1)) Namen finden. Als Beispiel aus der Welt sei der Name „Streckenkilometer“ gegeben, um bei dem Beispiel aus dem Abschnitt über Datentypen (2.2.1) zu bleiben. Innerhalb von Programmen nennen wir solche Namen „Bezeichner“, für die auch besondere Einschränkungen (Beispiel: „Ein Bezeichner darf nicht mit einer Ziffer beginnen.“) gelten können. Da letztere natürlich von der Programmiersprache abhängig sind, werden wir dies in jeweiligen Abschnitten von Racket und Java beleuchten.

2.2.5 Operatoren

Funktional

Imperativ

Objektorientiert

Ein Operator beschreibt eine Aktion auf mindestens einem Datum.
Ein Beispiel für einen Operator ist die Addition, welche 2 oder mehr Zahlen summiert. Wie wir einen solchen Operator in den konkreten Sprachen nutzen, hängt von der Sprache ab. Es gibt allerdings eine Operatoren, die (zumindest von der Bedeutung her) in allen Programmiersprachen vorhanden sind, beispielsweise die Addition. Diese Operatoren werden im Abschnitt 2.3.4 beschrieben.

2.2.6 Strukturierung von Quellcodedateien

Funktional

Imperativ

Objektorientiert

Da mit steigender Größe des Projektes in den meisten Fällen auch die Anzahl der Quellcodedateien steigt, muss dieser Code strukturiert werden. In den meisten Sprachen stehen hier Mechanismen zur

Verfügung, um den Quellcode in eine logische Struktur zu bringen, welche nicht unbedingt physikalisch abgebildet sein muss. Beispielsweise stehen hier in C++ Namespaces und in Java Packages zur Verfügung. Letztere werden wir im Abschnitt zu Java behandeln.
Der Unterschied einer logischen und einer physikalischen Struktur besteht darin, dass ersteres nur in den Köpfen der Menschen und (meistens) auch im Quellcode verankert ist, während physikalische Strukturen direkt im Speicher abgebildet werden (zum Beispiel in Form von Verzeichnissen).

2.3 Anweisungen

Funktional

Imperativ

Objektorientiert

Nach Betrachtung des langweiligen Teil der Programmierung, den lexikalischen Bestandteilen, wenden wir und nun den Anweisungen zu, die unseren imperativen Programmen Leben einhauchen. Die schreiben dem Computer vor, welche Einzelschritte einer Aufgabe er abzuarbeiten hat und in welcher Reihenfolge.

2.3.1 Variablen und Konstanten

Imperativ

Objektorientiert

Stellen wir uns vor, wir wollen das Durchschnittsalter aller Studierenden am Fachbereich Informatik berechnen. Zur Einfachheit nehmen wir hier an, es gibt nur 10 Studierende, welche die Alter $21, 41, 27, 18, 20, 24, 30, 17, 29, 25$ haben. Zur Berechnung des Durchschnitts müssen wir nun alle Zahlen aufsummieren, uns das Ergebnis merken und durch die Anzahl an Studierenden dividieren. Wir merken hier schnell, das wir uns „die Zwischenergebnisse merken“ müssen, da wir nicht mit so vielen Zahlen zur gleichen Zeit rechnen können.
Das gleiche Phänomen tritt auch auf, wenn wir obigen Algorithmus programmieren wollen: Wir müssen uns Daten zwischenspeichern.
Dies ist genau der Punkt, an dem Variablen ins Spiel kommen: Diese stellen einen kleinen, modifizierbaren, Speicher dar, dem eine Aufgabe und ein Datentyp zugeordnet ist. In unserem Fall ist die Aufgabe *das Halten des Zwischenergebnisses* und der Datentyp *Integer*. Um die Aufgabe erkenntlich zu machen, geben wir der Variablen einen Namen: „Gesamtalter“.
Um das alles Zusammen zu fassen: Eine Variable ist ein benannter Zwischenspeicher innerhalb eines Programms, an dem wir Daten eines bestimmten Datentyps speichern können und die Werte verändern. Selbstverständlich können wir auf die gespeicherten Werte auch zugreifen, sonst wäre das Speichern sinnlos [2].
Eine Konstante ist im Prinzip das gleiche wie eine Variable, nur ist sie nicht änderbar (sie ist „konstant“). Dies ist sinnvoll, wenn wir den gleichen Wert häufig im Programm verwenden, diesen aber nicht immer Tippen möchten. Ein prominentes Beispiel hierfür ist die Kreiszahl $\pi \approx 3,141592653589793$, welche in fast allen Sprachen bereits als Konstante vorliegt.

Begriffe

- Wenn eine Variable erstellt wird, das heißt der Typ und der Name wird festgelegt, wird dies *Deklaration* genannt.

[2] In Java gibt es hier eine Ausnahme, eine sogenannte „Phantom Reference“. Für mehr Informationen siehe https://docs.oracle.com/javase/10/docs/api/java/lang/ref/PhantomReference.html

- Die erste Festlegung, welchen Wert eine Variable am Anfang haben soll, wird *Initialisierung* genannt.
  Bei Konstanten ist dies dies einzige Wertfestlegung, die stattfinden kann.

2.3.2 Zuweisungen

Imperativ

Objektorientiert

Wenn der Wert einer Variablen festgelegt wird, wird dies *Zuweisung* genannt. Wie im Abschnitt zu Variablen und Konstanten (2.3.1) bereits erwähnt, kann einer Konstanten nur einmalig ein Wert zugewiesen werden.
Näheres zu Zuweisungen wird im Abschnitt Links-/Rechtsauswertung (2.3.5) behandelt.

2.3.3 Nutzung von Methoden

Funktional

Imperativ

Objektorientiert

Fürs erste reicht es uns hier zu wissen, dass eine Methode ein Weg ist, um Code zu deduplizieren an zentraler Stelle zu halten. Näheres über Methoden werden wir im Abschnitt zu Methoden und Funktionen (2.5) behandeln.
Wir werden oftmals feststellen, dass einige Dinge schwierig zu implementieren sind. Zum Glück haben die meisten Programmiersprachen eine sogenannte *Standardbibliothek*, welche viele Methoden zur Verfügung stellt, die Standardaufgaben implementieren (beispielsweise die Quadratwurzel). Um diese Methoden zu nutzen, müssen die Methoden aufgerufen werden. Dies ist zu vergleichen mit dem Aufruf einer Funktion in der Mathematik, beispielsweise der Quadratwurzelfunktion
($\sqrt{x} := y \in \mathbb{R}_+ : y \cdot y = x$). Wird übergeben einer Methode *Parameter* (oder auch *Argumente*), die Methode bearbeitet diese irgendwie und liefert uns ein *Ergebnis* (Eingabe-Verarbeitung-Ausgabe, EVA-Prinzip). Im Fall von der Quadratwurzel übergeben wir der Funktion als Parameter eine Zahl $x \in \mathbb{R}_+$ und kriegen ein solches Ergebnis, sodass $x = \sqrt{x} \cdot \sqrt{x}$ gilt (das dies in einigen Fällen nicht oder nicht vollständig lösbar ist, behandeln wir hier nicht weiter).

2.3.4 Operatoren

Funktional

Imperativ

Objektorientiert

In diesem Abschnitt behandeln wir nun einige Grundbegriffe, welche im Zusammenhang mit Operatoren immer wieder verwendet werden und schauen uns einige grundlegende Operatoren an, welche in annähernd allen Programmiersprachen vorhanden sind [3].

[3] Ausnahmen bilden hier manche esoterische Programmiersprachen wie beispielsweise Brainfuck. Siehe hierzu https://de.wikipedia.org/wiki/Brainfuck

Arithmetische Operatoren

Oftmals vorhandene arithmetische Operatoren sind „Addition“, „Subtraktion“, „Multiplikation“, „Division“ und „Modulo“. Hierbei sind auch in der Programmierung sämtliche aus der Mathematik bekannte Gesetze und Axiome anwendbar. Ein Beispiel hierfür ist die Kommutativität der Addition.
Der Operator „Modulo“ gibt uns den Wert des Restes bei einer Division zurück (Beispiel: Der Ausdruck „Rechne 5 modulo 3“ gibt uns den Wert 2, da: $1 \cdot 3 + 2 = 5$).
Mathematisch kann man den Modulo-Operator wie folgt definieren:

$$x \text{ modulo } y := x - \left\lfloor \frac{x}{y} \right\rfloor$$

Logische Operatoren

Auch vorhanden sind logische Operatoren wie „AND“ (logisches Und), „OR“ (logisches Oder), „NOT“ (logische Negation) und „XOR“ (logisches exklusives Oder). Diese arbeiten auf Wahrheitswerten und ergeben folgende Wahrheitstafeln [4]:

A	B	A AND B	A OR B	NOT A	NOT B	A XOR B
𝔣	𝔣	𝔣	𝔣	𝔴	𝔴	𝔣
𝔣	𝔴	𝔣	𝔴	𝔴	𝔣	𝔴
𝔴	𝔣	𝔣	𝔴	𝔣	𝔴	𝔴
𝔴	𝔴	𝔴	𝔴	𝔣	𝔣	𝔣

Tabelle 2.1: Wahrheitstafel für „and“, „or“, „not“ und „xor“

> **Warnung:** Das logische Oder ist genau dann wahr, wenn **mindestens ein Parameter** wahr ist. Das in der Prosa übliche „oder“ ist im *exklusiven oder* („xor“) implementiert, dieses wird genau dann wahr, wenn **genau ein Parameter** wahr ist (in anderen Worten: Wenn die Parameter unterschiedlich sind).

Auch für die logischen Operatoren gilt, dass Gesetze und Axiome aus der Logik auch in der Programmierung anwendbar sind (beispielsweise die De Morganschen Gesetze).

Bitlogische Operatoren

Bitlogische Operatoren arbeiten ähnlich zu den logischen Operatoren, können aber auf Zahlen operieren. Hierbei wird jedes Bit einer Zahl einzeln betrachtet, sodass wieder mit *Wahr* (1) und *Falsch* (0) gearbeitet werden kann. Hier sind üblicherweise ebenfalls die üblichen logischen Operatoren „AND“, „OR“, „NOT“ und „XOR“ verfügbar.
Wir betrachtet nun einige Beispiele, welche allesamt auf den Zahlen $\alpha = 1100_2 = 12_{10}$ und $\beta = 1010_2 = 10_{10}$ operieren, dabei ist sowohl das Ergebnis in Binär (tiefgestellte 2) und in Dezimal (tiefgestellte 10) angegeben:

[4] Wahrheitstafeln beschreiben, bei welchen durch welche Operationen welche Ergebnisse vorliegen

	Binär	Dezimal
α AND β	1000_2	8_{10}
α OR β	1110_2	14_{10}
NOT α	0011_2	3_{10}
NOT β	0101_2	5_{10}
α XOR β	0110_2	6_{10}

Tabelle 2.2: Ergebnistafel für bitlogische Operationen

Bindungsstärke

Da meistens mehrere Operatoren innerhalb eines Ausdrucks verwendet werden, ist es wichtig zu wissen, in welcher Reihenfolge die Operatoren ausgeführt werden. Diese Reihenfolge wird „Auswertungsreihenfolge“ genannt und basiert auf der „Bindungsstärke“ von Operatoren. Ein Operator mit einer hohen Bindungsstärke wird hierbei vor einem Operatoren mit einer geringeren Bindungsstärke ausgeführt.
Schauen wir uns zur Veranschaulichung das simpelste Beispiel an, welches wir alle aus der Schulmathematik kennen: „Punkt vor Strich“.
Mit dieser Regel wird festgelegt, wie ein Ausdruck $5 + 3 \cdot 7$ ausgewertet wird, nämlich:

$$5 + 3 \cdot 7 = 5 + 21 = 26$$

und nicht wie folgt, wenn „Strich vor Punkt“ gelten würde:

$$5 + 3 \cdot 7 = 8 \cdot 7 = 56$$

In anderen Worten: Die Operatoren „Multiplikation“ und „Division“ haben eine höhere Bindungsstärke

2.3.5 Links-/Rechtsauswertung

Imperativ

Objektorientiert

Eine Auswertung findet immer dann statt, wenn der Wert eines Ausdrucks bestimmt wird. Beispielsweise sagen wir, der Ausdruck $1 + 3$ wertet zu 4 aus.
Wird dieser Wert (das Ergebnis des Ausdrucks) anschließend einer Variable zugewiesen, wird von Links- und Rechtsauswertungen gesprochen. Die Auswertung des zuzuweisenden Wertes wird dabei *Rechtsauswertung* genannt. Der Ausdruck, der bestimmt, wem/was der Wert zugewiesen werden soll, wird *Linksauswertung* genannt.
Die Namensgebung resultiert daher, dass in den meisten Programmiersprachen der zuzuweisende Wert rechts und die Variable links steht, Beispiel:

```
1 a[b] = a[b] + 1
```

Hierbei wertet der Linksausdruck (`a[b]`) zu einer Position im Speicher aus, an der das Element gespeichert werden soll (in diesem Listing werden Arrays verwendet, diese werden erst im Abschnitt über Datenstrukturen (2.9) eingeführt), und der Rechtsausdruck (`a[b] + 1`) zu dem Wert, welcher gespeichert werden soll.

2.3.6 Seiteneffekte

Imperativ

Objektorientiert

Was bei funktionalen Sprachen verhindert wird, tritt bei imperativen Sprachen häufig auf: Seiteneffekte. Als Seiteneffekt wird bezeichnet, wenn sich der Zustand von etwas ändert, beispielsweise von einer Variablen. Wir sehen hier bereits, dass Seiteneffekte integraler Bestandteil von imperativen Sprachen sind (in funktionalen Sprachen nicht, da es hier kein Konzept von Variablen gibt). Seiteneffekte werden dann problematisch, wenn sich der Zustand einer Variablen verändert, ohne dass wir direkt Einfluss darauf nehmen können (zum Beispiel wenn mehrere Codeblöcke parallel zueinander ablaufen, siehe 2.11).
Wir werden weitere Probleme mit Seiteneffekten im späteren Kapitel über Java behandeln.

2.4 Kontrollstrukturen

Funktional

Imperativ

Objektorientiert

Bisher haben wir nur Konzepte betrachtet, die es uns ermöglichen, einen linearen und immer gleichen Ablauf des Programms zu bewerkstelligen. Aber spätestens, wenn unser Programm an irgendeiner Stelle

a) Entscheidungen treffen oder

b) Codeblöcke wiederholen

soll, müssen wir anfangen, darüber nachzudenken, wie wir Entscheidungen implementieren.

2.4.1 Verzweigungen

Funktional

Imperativ

Objektorientiert

Wir beschäftigen uns nun mit Teil a) von obiger Liste: Verzweigungen, das wichtigste überhaupt, wenn es darum geht, in einem Programm Entscheidungen zu treffen. Wir werden nun den Haupttyp Typen von Verzweigungen kennen lernen: ein *if*.

If

Ein *if* ist eine einfache Verzweigung der Form „Wenn . . . gilt, dann tue . . . . Sonst tue . . . .“. In den meisten Programmiersprachen wird dies gesprochen als „**if** . . . **then** . . . **else** . . . “, was einer einfachen Übersetzung des deutschen „wenn, dann, ansonsten“ entspricht. Der sogenannte *else-Block* kann bei den meisten Sprachen auch fallengelassen werden, wird die Bedingung dann zu *Falsch* ausgewertet, so geschieht einfach nichts.
In vielen Fällen muss man allerdings mehr als einen Fall betrachten, wodurch sich *elseif-Blöcke* ergeben, die ungefähr die Form „**if** . . . **then** . . . **elseif** . . . **then** . . . **else** . . . “ haben. Umgangssprachlich kann man ein else-if also als „wenn, dann, ansonsten wenn, ⋯, ansonsten“ ausdrücken. Es kann in den meisten Fällen beliebig viele else-if-Blöcke geben.

2.4.2 Schleifen

Imperativ

Objektorientiert

Nun beschäftigen wir uns mit Teil b) aus obiger Liste: Schleifen. Spätestens, wenn wir unseren Code mehrmals Ausführen wollen und ihn zu diesem Zweck nicht einfach untereinander kopieren können (beispielsweise wenn die Ausführung von einem Parameter abhängt, dessen Wert wir während dem Schreiben noch nicht kennen), müssen wir unseren Code dynamisch beliebig oft Ausführen. Dies ist zum Beispiel der Fall, wenn wir die Fakultät einer Zahl berechnen wollen:

$$n! := \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n$$

Als Grundlage für alle Schleifen dient die *while-Schleife*, bei der ein Codeblock so lange ausgeführt wird, bis eine bestimmte Bedingung zu *Falsch* auswertet. Der Code kann somit als „Solange ... tue ... “ verstanden werden und sieht in den meisten Sprachen auch ähnlich aus: „**while** ... **do**“. Als Anlehnung an die while-Schleife gibt es manchmal auch die until-Schleife, die genau entgegengesetzt funktioniert: Der Codeblock wird so lange ausgeführt, bis eine bestimmte Bedingung zu *Wahr* auswertet.
Damit können wir nun das obige Problem wie folgt in unserer imaginären Sprache lösen, wobei wir hier in der Variable `n` das n von oben speichern und in der Variable `x` das Ergebnis (es soll also nach der Ausführung `x` $= n!$ gelten).

```
1 x = 1
2 while n > 0
3     x = x * n
4
5     n = n - 1
6 done
```

Abbildung 2.3: Beispiel: While-Schleife

Der Code `x = x * n, n = n - 1` wird nun immer ausgeführt, solange `n` > 0 gilt. Eine Ausführung des Blocks wird „Schleifendurchlauf“ oder „Iteration“ genannt.
Da es manche Fälle gibt, in denen die gesamte Ausführung innerhalb der Schleife abgebrochen werden soll oder die aktuelle Iteration abgebrochen werden soll und mit der nächsten begonnen werden soll, gibt es meist noch die Ausdrücke „break“ und „continue“, welche ihrem Namen treu bleiben und die folgenden Funktionen erfüllen:

break Hält die gesamt Schleifenausführung an und fährt mit der ersten Anweisung nach der Schleife fort.

continue Hält den aktuellen Schleifendurchlauf an und fährt mit der nächsten Iteration fort. Gibt es keine weitere Iteration, so wird die Schleife beendet.

2.5 Methoden/Funktionen

Funktional

Imperativ

Objektorientiert

Mit steigender Komplexität der Programme werden wir sehen, dass sich oftmals viele Stellen im Code doppelt, dreifach oder noch öfter zu finden sind. Auch wird ersichtlich, dass ein Programm, welches aus > 200 Zeilen besteht, nicht mehr übersichtlich ist.
Hier können *Methoden* helfen, welche den Code strukturieren.
Eine Methode nimmt Daten entgegen (Eingabe), verarbeitet diese (Verarbeitung) und gibt das Ergebnis aus (Ausgabe). Somit setzen Methoden das EVA-Prinzip der Informatik um und können als Blackbox betrachtet werden, die eine bestimmte Aufgabe (beispielsweise Wurzelziehen) erledigen.

2.5.1 Aufbau

Funktional

Imperativ

Objektorientiert

Wie bereits erwähnt nimmt eine Methode Daten entgegen, verarbeitet diese und gibt das Ergebnis aus (wenn eines berechnet wurde).
Die eingehenden Daten werden über *Parameter* übergeben, im *Körper* verarbeitet und mit einer *Rückgabe* zurück gegeben. Dabei entspricht bei funktionalen Sprachen eine Methode einer Mathematischen Funktion, welche genau eine Sache tut und Werte zurück gibt. Bei imperativer Programmierung ist es möglich, dass Methoden keinen Rückgabewert haben und mehrere Dinge tun können (mehrere Zeilen Code).
Schauen wir uns diese Einzelkomponenten von Methoden nochmals genauer an.

Parameter

Funktional

Imperativ

Objektorientiert

Ein *Parameter* wird von einer Methode spezifiziert (die Anzahl und der Typ der Parameter) und kann innerhalb der Methode wie eine normale Variable (oder in manchen Sprachen Konstante) verwendet werden. Sie stellen die Eingabe der Daten in die Methode dar und werden vom Aufrufer übergeben.
Parameter sind auch bekannt aus der Mathematik, beispielsweise das „x“ in der Funktion

$$f : \mathbb{N} \to \mathbb{N} : \underline{x} \mapsto x^2$$

wobei „$\mathbb{N}$“ den Typ des Parameters festlegt.

Formale Parameter vs. Aktualparameter
Wir unterscheiden zwischen den *Formalen Parametern* und den *Aktualen Parametern*.

- Die *Formalen Parameter* sind diejenigen Parameter, die bei der Methodendefinition angegeben werden.
- Die *Aktualen Parameter* sind diejenigen Parameter, die bei dem Methodenaufruf angegeben werden.

Körper

Funktional

Imperativ

Objektorientiert

Der Code innerhalb einer Methode wird als *Körper* der Methode bezeichnet. Dieser nutzt die Parameter der Methode, enthält den nötigen Code für die zu erledigende Aufgabe und führt die Rückgabe aus. Er stellt den wichtigsten Teil der Methode dar, denn ohne Code wäre eine Methode sinnfrei.
Wieder als mathematische Funktion betrachtet, kann das „x^2“ als Körper der Funktion

$$f : \mathbb{N} \to \mathbb{N} : x \mapsto \underline{x^2}$$

verstanden werden.

Rückgabe

Funktional

Imperativ

Objektorientiert

Jede Methode hat eine *Rückgabe*, welche bei imperativen Sprachen auch leer sein kann (also kein expliziter Rückgabewert). Dies wird meist als *Void* (englisch für „Nichts“) bezeichnet. Der *Rückgabewert* ist der Wert, den die Methode aus den Parametern berechnet hat und den der Aufrufer erhält. In einer (statisch) typisierten Sprache wird außerdem der Rückgabetyp definiert, mit dem der Aufrufer rechnen kann. Die Rückgabe ist bei einer funktionalen Programmiersprache implizit, das heißt es wird nicht genau angegeben, an welcher Stelle ein Wert zurück gegeben wird. In einer imperativen Programmiersprache erfolgt die Rückgabe explizit und kann auch schon vor vollständigem Ablauf der Methode ausgeführt werden. In diesem Fall bricht die Ausführung der Methode ab. Dies kann zum Beispiels nützlich sein, wenn auf invalide Eingaben (beispielsweise negative Zahlen) geprüft wird.
Erneut als mathematische Funktion

$$f : \underline{\mathbb{N}} \to \mathbb{N} : x \mapsto x^2$$

betrachtet, kann „$\mathbb{R}$“ als Rückgabetyp verstanden werden. Da mathematische Funktionen funktional sind, findet keine explizite Rückgabe statt, sondern es wird einfach das Quadrat von x zurück gegeben.

2.5.2 Verträge

Funktional

Imperativ

Objektorientiert

Verträge sind Teil der Dokumentation, siehe Abschnitt 2.13.1.

2.5.3 Rekursion

Funktional

Imperativ

Objektorientiert

> Um die Rekursion zu verstehen, muss man zuerst die Rekursion verstehen (siehe 2.5.3) .

Rekursion bezeichnet ein Paradigma, bei dem sich eine Methode selbst aufruft und damit eine Schleife vermeidet. Dadurch ist es möglich, viele Probleme sehr übersichtlich zu lösen, wodurch der Code lesbarer und damit wartbarer wird. Wenn man es mit der Rekursion allerdings übertreibt, wird der Code sehr schwer verständlich.
Schauen wir uns als Beispiel eine mathematische Definition an, bei der Rekursion eingesetzt wird: Die Fakultät $n!$ einer natürlichen Zahl $n \in \mathbb{N}$:

$$n! = \begin{cases} 1 & \text{falls } n = 1 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

Hier wird im zweiten Fall der Fallunterscheidung Rekursion eingesetzt, indem die Fakultät-Funktion erneut genutzt wird.

Rekursionsanker
Der *Rekursionsanker* bezeichnet den Teil der Funktion, der dafür zuständig ist, dass der Methodenaufruf beendet wird. Im obigen Beispiel ist dies der erste Fall der Fallunterscheidung, da dieser die Rekursion abbricht.
Es kann auch mehrere Rekursionsanker geben, die zusammen zum Ende der Rekursion führen.
Übung: In folgendem Beispiel wird die n-te Fibonacci-Zahl berechnet. Was ist in diesem Fall der Rekursionsanker?

$$\text{Fib}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{sonst} \end{cases}$$

Dies führt zu der Fibonacci-Folge

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, \cdots$$

2.5.4 Überladen

Funktional

Imperativ

Objektorientiert

In vielen Programmiersprachen ist es möglich, mehrere Methoden mit dem gleichen Namen, aber unterschiedlichen Parametern, zu implementieren. Dann wird anhand der Parameter entschieden, welche Methode ausgeführt werden soll. Je nach Sprache ist es nur möglich, nach Anzahl der Parameter zu unterscheiden oder auch nach Typ zu unterscheiden (Java, Kotlin, . . .). Andere Sprachen unterstützen Überladung gar nicht (Racket, Python, . . .).

2.5.5 Überschreiben

Objektorientiert

Im Kontext von objektorientierter Programmierung und Vererbung ist es von Zeit zu Zeit nötig, eine Implementierung einer Funktion von der Oberklasse zu ändern. Hierzu kann die gleiche Methode erneut implementiert werden, welche anschließend die vorherige Implementation ersetzt. Dies wird *Überschreiben* von Methoden genannt. Die meisten Sprachen bieten hier auch die Möglichkeit, explizit die „alte“ Implementierung aus der Oberklasse aufzurufen.

2.6 Scoping

Funktional

Imperativ

Objektorientiert

Stellen wir uns vor, in einer Programmiersprache wären *alle* Variablen von *überall* zugreifbar und veränderbar. Dies würde zu Chaos führen, spätestens wenn zwei Schleifen mit dem Laufindex `i` zeitgleich ausgeführt werden.
Um Szenarien wie diese zu vermeiden, unterstützen die meisten Sprachen *Scoping*, also bestimmte Regeln, von wo aus eine Variable zugreifbar ist und wann Variablen mit dem gleichen Namen völlig unterschiedliche Werte repräsentieren können. Diese Regeln sind zumeist sehr simpel wie und legen beispielsweise fest, dass in einer Methode definierte Variablen nicht außerhalb dieser verwendet werden können. Ein Bereich, in dem eine Variable gültig ist, wird *Scope* genannt.
Schauen wir uns mit obiger Regel zuerst ein Beispiel an, bevor wir zu den grundlegenden Scoping-Typen fortfahren.

Beispiel
Wir betrachten folgendes Stück Code mit obiger Scoping-Regeln, dass unterschiedliche Methoden unterschiedliche Variablen definieren und unabhängig voneinander sind:

```
1  foo() {
2      a = "foo/a"; // Anlegen einer neuen Variable 'a' im Scope 'foo'.
3      b = "foo/b"; // Anlegen einer neuen Variable 'b' im Scope 'foo'.
4
5      print(a);    // Gibt 'foo/a' aus.
6      print(b);    // Gibt 'foo/b' aus.
7  }
8
9  bar() {
10     a = "bar/a"; // Anlegen einer neuen Variable 'a' im Scope 'bar', von
           'foo' unabhaengig.
11
12     print(a);    // Gibt 'bar/a' aus.
13     print(b);    // Schlaegt fehl, da 'b' im Scope 'bar' nicht definiert
           ist.
14 }
```

Abbildung 2.4: Scoping

Typen von Scopes
Bei imperativen Sprachen existieren die folgenden grundlegenden Arten von Scoping:

Global Alle Variablen sind global gültig und können von überall verändert werden. Dies stellt eigentlich keinen Scoping-Typ dar, da kein Scoping vorgenommen wird.

Function-Based Variablen sind nur innerhalb einer Funktion gültig und können auch nur dort verwendet werden.
Beispiele: Python, BASH

Block-Based Variablen sind nur innerhalb eines Codeblocks (beispielsweise abgetrennt durch geschweifte Klammern) gültig. Ein Codeblock kann beispielsweise mit einer Schleife, einem If, ... eingeleitet werden.
Beispiele: Java, Kotlin

Mixed Es werden unterschiedliche Scoping-Verfahren implementiert und der Entwickler entscheidet, welches er nutzen will.
Beispiele: JavaScript (eine globale Variable wird ohne Schlüsselwort, eine funktionslokale Variable mit dem Schlüsselwort `var` und eine block-scoped Variable mit dem Schlüsselwort `let` eingeleitet)

2.7 Klassen und objektorientierte Programmierung

Objektorientiert

Wir werden uns in diesem Abschnitt einige abstrakte Konzepte zur objektorientierten Programmierung anschauen.

2.7.1 Konzept

Das Paradigma der objektorientierten Programmierung haben wir uns bereits im Abschnitt zu Programmierparadigmen (siehe 2.1.4) angeschaut.
In der objektorientierten Programmierung dreht sich also alles um sogenannte *Objekte*. Aber wie bringen wir dem Computer bei, unseren Code als Objekt, beziehungsweise als Objekte, anzusehen? Hierbei stoßen wir auf die Konzepte von Klassen, Methoden, Interfaces und Vererbung. Dies werden wir uns in den folgenden Abschnitten genauer anschauen.
Das Grundlegende Prinzip der Objektorientierung ist, komplexe Sachverhalten wie sie in der Realität auftreten zu vereinfachen und möglichst Realitätsnah abzubilden. Dazu betrachten wir die Programmierung wie die Realität, bei der sich ebenfalls alles um Objekte dreht.

2.7.2 Klassen, Objekte und Methoden

Imperativ

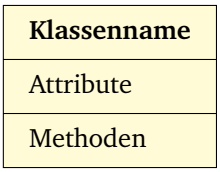
Objektorientiert

Klassen können wir als „Vorlagen“ für *Objekte* ansehen und Objekte realisieren eine Klasse dahingehend, als dass bestimmte Eigenschaften der Klasse mit Werten gefüllt werden. Diese Eigenschaften sind Attribute, die für jedes Objekt getrennt gespeichert und verarbeitet werden und aus Methoden, welche Funktionalitäten zu einer Klasse, beziehungsweise einem Objekt, hinzufügen.
Schauen wir uns ein Beispiel an: Um uns herum sind viele Personen, die wir grob als Klasse „Mensch“ mit dem Attribut „name“ abbilden können. Dann sind Personen wie „Florian“ mit dem Namen „Florian

Kadner“ und Fabian mit dem Namen „Fabian Damken“ *Instanzen* dieser Klasse, genannt *Objekte*. Wollen wir nun einem Menschen die Funktionalität „gehen“ hinzufügen, so fügen wir der Klasse „Mensch“ eine Methode „gehen“ hinzu. Damit können wir nun diese Methode auf jedem Menschen aufrufen, zum Beispiel auf Fabian und Florian. Diese laufen nun wild durch die Gegend.
Das fasst alles grob zusammen, was wir abstrakt über Klasse, Objekte und Methoden lernen können. Je nach Programmiersprache gibt es hier noch viele Besonderheiten, die wir uns für Java noch genauer im Abschnitt 4.6.1 anschauen werden.
Schauen wir uns noch kurz an, was es mit *statischen* Attributen/Methoden auf sich hat. Statische Attribute und Methoden „hängen“ direkt an der Klasse und verhalten sich nicht je nach Objekt anders. Das bedeutet, dass auf die Attribute und Methoden auch ohne ein konkretes Objekt zugegriffen werden kann und dass jedes Objekt auf den den selben Wert zugreift wie andere Objekte der gleichen Klasse. An dieser Stelle müssen wir darauf achten, nicht zu viele statische Attribute und Methoden zu verwenden, da hiermit die Objektorientierung wieder ausgehebelt werden kann.
Noch eine kurze Begriffsklärung:

Klasse	Die Vorlage für Objekte. Kann Attribute und Methoden enthalten.
Objekt	Ein Objekt einer Klasse ist die Realisierung der selbigen mit festgelegten Attributwerten.
Instanz	Das gleiche wie ein Objekt.
Instanzvariable	Ein Attribut einer Klasse, welches Instanzspezifisch ist.
Klassenvariabke	Ein Attribut einer Klasse, welches für alle Instanzen das selbe ist.
Methode	Ein aufrufbares Stück Code, welches Parameter annimmt, Daten zurück geben kann und auf die Attribute der Klasse/des Objektes zugreifen kann.
Klassenmethode	Das gleiche wie eine Methode, nur dass die für alle Instanzen gleich ist, ohne Erstellung einer Instanz aufgerufen werden kann und somit auch keinen Zugriff auf die Instanzvariablen hat.

UML-Diagramm
Eine Klasse wird wie folgt in UML[5] dargestellt:



2.7.3 Vererbung

Imperativ

Objektorientiert

Klassen und Objekte sind schön und gut, aber betrachten wir folgendes Beispiel: Wir haben die Klassen „Quadrat“ und „Kreis“, die beide eine Methode „ausmalen“ implementieren. Die Implementierung ist in dem Fall für beide Klassen gleich und wir haben den Code einfach kopiert. Ist die Implementierung nun

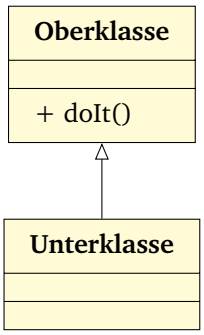
[5] Unified Modeling Language, eine Definition von vielen Diagrammtypen. In dem Skript werden wir nur das UML-Klassendiagramm nutzen, weitere Diagramme werden in der Veranstaltung „Software Engineering“ behandelt.

Fehlerhaft, so müssen wir beide Methoden anpassen und dürfen dies auch nicht vergessen. Hier kommt Vererbung ins Spiel: Wir erstellen eine Oberklasse „Form“, von der die Klassen „Quadrat“ und „Kreis“ erben. Das bedeutet nun, dass alle Methoden, die in der Klasse „Form“ definiert sind, auch in den Klassen „Kreis“ und „Quadrat“ verfügbar sind. Somit müssen wir die Methode „ausmalen“ nur einmal implementieren.

In den meisten Programmiersprachen ist es nur möglich, von maximal einer Klasse zu erben. Mehrfachvererbung wird auch von einigen Sprachen implementiert, dies erhöht an vielen Stellen allerdings die Komplexität und wurde deshalb in den moderneren Sprachen weg gelassen.

UML-Diagramm

Eine Vererbung wird in UML mit einem Pfeil dargestellt, wobei das Pfeilende nicht ausgemalt wird und auf die Klasse zeigt, von der geerbt wird.



2.7.4 Abstrakte Klassen

Imperativ

Objektorientiert

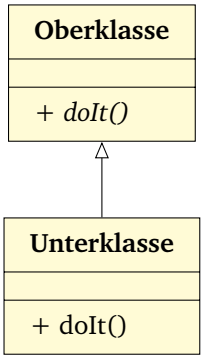
Abstrakte Klassen bieten uns einen Weg, eine Methode zu definieren, aber nicht zu implementieren. Das bedeutet, wir definieren den Namen, die Parameter und den Rückgabetyp einer Methode ohne diese zu implementieren und überlassen es den Unterklassen, die Methode zu implementieren.

Dadurch ist es möglich, eine Ebene an Abstraktion zu erreichen, die wir uns am besten an einem Beispiel anschauen: Wir haben die abstrakte Klasse „Form“, welche die Methode `flaecheninhalt()` definiert. Diese Methode zur Berechnung des Flächeninhaltes der jeweiligen Form ist Abstrakt und wird von jeder Form (zum Beispiel „Kreis“ oder „Quadrat“) implementiert. Wir können nun eine Methode `double farbverbrauch(Form form, doubl)` entwickeln, welche den Farbverbrauch bei dem Ausmalen der jeweiligen Funktion berechnet. Den Flächeninhalt der Form erhält die Methode dabei durch die vorher definierte abstrakte Methode `flaecheninhalt()`, die auf der übergebenen Form aufgerufen wird.

Dabei wird immer die Implementierung aufgerufen, die gerade in dem Objekt steht, siehe 2.7.6. Es ist auch möglich, dass eine Oberklasse eine Methode implementiert, welche wiederum andere, abstrakte, Methoden nutzt, die später erst in den Unterklassen implementiert werden.

UML-Diagramm

Abstrakte Methoden werden in UML kursiv gekennzeichnet:



2.7.5 Interfaces

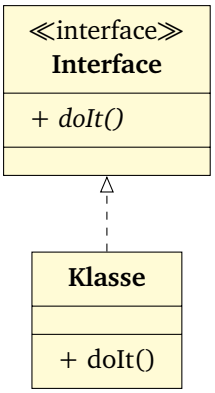
Imperativ

Objektorientiert

Ein *Interface* ist eine besondere abstrakte Klasse und definiert ausschließlich abstrakte Methoden und kann keine Methoden implementieren. Auch kann eine Klasse in den meisten Sprachen auch mehrere Interfaces *implementieren*. Dadurch können wir feste Schnittstellen definieren und Klassen, welche diese Schnittstellen implementieren und können im Rahmen dieser alle gleich genutzt werden. Schauen wir uns dies wieder an einem Beispiel aus der Realität an: In Java existiert das Interface `Comparable`. Dieses Interface definiert eine einzige Methode `int compareTo(Object obj)`, mit der das Objekt, auf dem die Methode aufgerufen wird, mit dem übergebenen Objekt verglichen werden kann. Nun kann jede Klasse, die dieses Interface implementiert ist nun als „vergleichbar“ markiert und kann zum Beispiel innerhalb von sortierten Listen verwendet werden.[6]

UML-Diagramm

Ein Interface wird in UML besonders mit einem sogenannten „Stereotyp“ hervorgehoben, eine Implementierung wird mit gestrichelten Pfeilen dargestellt:



2.7.6 Polymorphie und späte Bindung

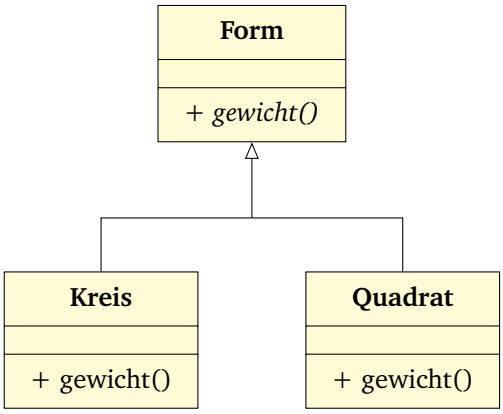
Imperativ

Objektorientiert

Dies ist eines der wichtigsten Eigenschaften bei objektorientierter Programmierung. Polymorphie beschreibt die Eigenschaft, dass die Implementierung einer Methode aufgerufen wird, die in dem

[6] Hier gibt es auch noch andere Möglichkeiten, wir werden dies im Abschnitt **??** genauer betrachten.

Vererbungsbaum eines Objektes an tiefstmöglicher Stelle steht und nicht zwangsweise die Implementierung aufruft, die in der referenzierten Klasse vorhanden ist.
Betrachten wir hierzu ein Beispiel, um das Prinzip zu veranschaulichen: Wir haben eine Klasse „Form“ mit einer abstrakten Methode „gewicht()“, die von den Klassen „Kreis“ und „Quadrat“ überschrieben wird:



Wenn wir nun an einer Stelle ein Objekte der Klasse „Form“ nutzen und darauf die Methode `gewicht()` aufrufen, so wird die tiefstmögliche Implementierung aufgerufen.
Das bedeutet, wenn unsere Variable mit dem Typ „Form“ ein Objekt der Klasse „Kreis“ enthält, so wird die Implementierung in „Kreis“ aufgerufen. Wie dieses Verfahren der Polymorphie und später Bindung genau funktioniert, werden wir uns im Abschnitt 4.6.6 zu der Implementierung in Java anschauen.
Damit haben wir nun die wesentlichen abstrakten Konzepte der objektorientierten Programmierung abgearbeitet.

2.8 Fehlerbehandlung

Funktional
Imperativ
Objektorientiert

Während der Ausführung von Code kann es immer zu Fehler kommen, welche es zu behandeln gilt (beispielsweise bei einer Division durch 0). Hierbei stellt sich die Frage, wie wir dem Nutzer (oder dem Aufrufer einer Methode) erkenntlich machen, dass es zu einem Fehler gekommen ist.
Dabei gibt es unterschiedliche Möglichkeiten, welche sich grob wie folgt einteilen lassen:

- Exceptions und
- Result Codes.

Diese zwei Unterarten werden wir in den folgenden Abschnitten behandeln.

2.8.1 Result Code

Beschäftigen wir uns zuerst mit der einfachsten Methode, Fehler anzuzeigen: Wir sagen dem Aufrufer über den Rückgabewert der Methode Bescheid, ob alles korrekt abgelaufen ist.
An einem konkreten Beispiel heißt dies:

- Szenario: Wir haben eine Methode `indexOf(val: String, el: char): int`, welche uns die Position des ersten Vorkommens von `el` in `val` zurück gibt.
  Beispiel: `indexOf(äsdfgas", 's')` gibt `1` zurück.

- Ist das Zeichen nicht in dem String vorhanden, stellt dies einen Fehler dar.

- Mit Fehlermeldung über Result Codes könnten wir nun beispielsweise `-1` zurück geben, da dies kein valider Index ist (welche immer $\geq$ `0` sein müssen).

- Damit sieht der Aufrufer, dass der Methodenaufruf schief gegangen ist und kann entsprechend reagieren.

Vorteile

- Es werden keine expliziten Verfahren zum Melden von Fehlern benötigt.

- Die genutzt Technologie wird in vielen Sprachen eingesetzt.

Nachteile

- Manchmal ist es nicht möglich, Fehler so anzuzeigen (beispielsweise wenn alle Werte gültig sind).

- Der Aufrufer muss extra prüfen und daran denken, ob und welche Codes zurück kommen könnten.

2.8.2 Exceptions

Schauen wir uns nun *Exception* an, ein sehr viel mächtigeres System als Result Codes.

Die Grundidee einer Exception ist, dass die Ausführung des Codes an einer beliebigen Stelle abgebrochen wird und dem Aufrufer über einen weiteren Mechanismus (den Exceptions) aufgezeigt wird, dass es Fehler vorlag. Der Aufrufer kann den Fehler anschließend behandeln.

Das System besteht aus den folgenden Teilen, welche wir in den nächsten Abschnitten näher betrachten:

- Werfen von Exceptions und

- Fangen von Exceptions.

Werfen von Exceptions

Eine Methode, welche beispielsweise die Berechnung $\frac{a}{b}$ durchführt, muss einen Fehler auslösen, wenn $b = 0$ gilt.

Im Kontext von Exceptions wird dieses Auslösen eines Fehler *werfen* einer Exception genannt, das heißt der Code bricht ab, es wird kein Wert zurück gegeben und der Aufrufer „erhält“ den Fehler, welcher eine genauere Beschreibung enthalten kann (beispielsweise die Nachricht „MathException: Cannot divide by 0.“).

Beispiel

```
1  divide(int a, int b) {
2      if (b == 0) {
3          // Nach der folgenden Zeile wird zum Aufrufer zurueck gekehrt,
               dieser
4          // erhaelt die Nachricht und die Ausfuehrung der Methode wird
5          // abgebrochen.
6          throw "MathException: Cannot divide by 0." // Werfen der Exception.
7      }
8
9      // Somit koennen wir uns nun sicher sein, dass 'b != 0' gilt und
           einfach mit
10     // der Division forfahren.
11     return /* Divisions-Algorithmus */
12 }
```

Abbildung 2.5: Exceptions Werfen: Beispiel

Fangen von Exceptions

Rufen wir eine Methode auf, welche eine Exception werfen kann (dies wird je nach Sprache in der Signatur der Methode dokumentiert), so müssen wir diese *fangen*. Das bedeutet, wir müssen die Exception empfangen und den Fehler behandeln (wie auch immer).

Dies geschieht zumeist mit einem Try-Catch-Konstrukt, welcher in zwei Blöcke aufgeteilt ist:

- Der *Try-Block* enthält den Code, der die Exception auslösen kann. Tritt irgendwo eine Exception auf, so bricht die Ausführung dieses Blocks ab.

- Der *Catch-Block* fängt eine mögliche Exception und wird nur ausgeführt, wenn im Try-Block ein Fehler aufgetreten ist. Sofern der Catch-Block nicht für einen Abbruch der Ausführung sorgt, wird nach seiner Ausführung einfach mit der ersten Zeile nach dem Try-Catch fortgefahren.

- In den meisten Sprachen gibt es noch einen Finally-Block, diesen werden wir aber erst im Java-Abschnitt zu Exceptions behandeln.

Beispiel

Sei wieder die Methode aus Abbildung 2.5 gegeben, nur rufen wir diese diesmal auf.

```
// Try-Catch-Konstrukt
try { // Try-Block

    // Dieser Aufruf geht noch gut, denn 2 != 0.
    divide(4, -2)
    // Dieser Aufruf wird fehlschlagen und der Catch-Block wird
       ausgefuehrt.
    divide(5, 0)
    // Dieser Aufruf wird nicht mehr ausgefuehrt, da der vorherige Aufruf
    // fehlgeschlagen ist.
    divide(6, 2)
} catch (String exception) { // Catch-Block

    // Der String 'exception' enthaelt nun den Wert "MathException: Cannot
       divide
    // by 0.", welcher von der Methode divide(int, int) als Fehlermeldung
    // uebergeben wurde.
    // Wir geben den Fehler hier einfach aus und behandeln ihn nicht
       weiter.
    print(exception)
}
```

Abbildung 2.6: Exceptions Fangen: Beispiel

Exception-Typen

Es wird im allgemeinen zwischen den folgenden Exception-Typen unterschieden:

Geprüft Diese Exceptions müssen von dem Aufrufer gefangen und behandelt oder weitergeleitet werden. Das Ignorieren der Exception führt zu einem Compiler Fehler.

Nicht Geprüft Diese Exceptions müssen nicht von dem Aufrufer gefangen werden und können ignoriert werden. Allerdings stürzt das Programm ab, sollte doch ein solcher Fehler auftreten.

> **Info:** Die Diskussion, ob man nur geprüfte Exceptions, nur ungeprüfte Exceptions oder beides verwenden sollte, ist sehr langwierig und es gibt für beide Seiten gute Argumente. Meiner Meinung nach ist die Mischform der beste Weg, da dieser am meisten Flexibilität bietet.

2.9 Datenstrukturen

Funktional

Imperativ

Objektorientiert

Eine Datenstruktur ist ein Objekt zur Speicherung und Organisation von Daten, indem diese in einer bestimmten Art und Weise angeordnet sind und es klare Namen gibt, sodass unterschiedliche Entwickler sich über Datenstrukturen unterhalten können, ohne an eine bestimmte Programmiersprache gebunden zu sein.

> **Info:** In dieser Veranstaltung werden wir Datenstrukturen nur grob behandeln, genauer wird dies in der Veranstaltung „Algorithmen und Datenstrukturen“ behandelt.

Im allgemeinen unterscheidet man zwischen folgenden Typen von Datenstrukturen:

Indexbasiert Jedes Element innerhalb einer Datenstruktur wird einem festen Index zugeordnet, mit dem das Element genau ausgewiesen werden kann.

Nicht Indexbasiert Die Elemente innerhalb einer Datenstruktur werden nicht indiziert und es ist nicht möglich, direkt auf ein bestimmtes Element zuzugreifen, ohne sich andere Elemente anzuschauen.

2.9.1 Arrays, Listen, Mengen

Funktional

Imperativ

Objektorientiert

Bevor wir uns einige Implementierungen von Arrays, Listen und Mengen anschauen, stellt sich zuerst die Frage, was das eigentlich alles ist?
Alle drei stellen eine Auflistung von Elementen eines beliebigen Typs dar und dienen dazu, beliebig viele und noch nicht zur Compilezeit bekannte Elemente in einer Variablen zusammenzufassen. Dabei wird unterschieden zwischen *indexbasierten* und *nicht indexbasierten* Strukturen, wobei bei ersteren jedes Element mit einem Index (einer Zahl) identifiziert werden kann, bei letzteren nicht.

Array

Ein Array ist eine solche Auflistung, welche in den meisten Sprachen nicht zur Laufzeit vergrößert werden kann und in einigen Sprachen (beispielsweise C) sogar schon zur Compilezeit feststehen muss. Grob gesagt kann man sich ein Array als beschriftetes Kistensystem vorstellen, bei dem jede Kiste eine Zahl zugewiesen bekommt:

Abbildung 2.7: Datenstruktur: Array

> **Warnung:** In annähernd allen Programmiersprachen werden Arrays ab dem Index `0` indiziert! Methoden zum Anzeigen der Länge zeigen jedoch die Anzahl der Elemente an und nicht den letzten Index!

Liste

Eine Liste ist einem Array sehr ähnlich, die Größe kann im Allgemeinen aber zur Laufzeit angepasst werden und es existieren viele verschiedene Implementierungen (unter anderem Implementierungen zur Abbildung auf Arrays, sogenannte Array-Listen). Oftmals ist auch eine Liste indexbasiert, dies muss aber nicht immer der Fall sein (beispielsweise bei verketteten Listen, die wir später behandeln werden).

Menge

Eine Menge ist, lapidar gesagt, eine Liste ohne Duplikate. Ansonsten gelten genau die gleichen Fakten wie bei einer Liste: Es kann indexbasiert sein, muss es aber nicht, ...

Linked List (Verkettete Liste)

Eine *verkettete Liste* ist eine indexlose Liste, in der die Datenspeicherung wie folgt abgebildet wird:

- Jedes Element der Liste enthält die Referenz auf:
  - die eigentlichen Nutzdaten (data),
  - den Nachfolger des Elementes (next) und
  - im Fall von einer doppelt verkettete Liste, eine Referenz auf das vorherige Element (previous).
- Existiert kein nachfolgendes/vorheriges Element, so wir nichts in das Feld eingetragen.
- Manchmal gibt es noch eine Schnittstelle, die eine Referenz auf das erste Element enthält und einige Methoden zur Verfügung stellt (`first`, `second`, `third`, ...). Diese ist aber nicht vorgeschrieben.

Visualisiert sieht eine einfach verkettete Liste wie folgt aus:

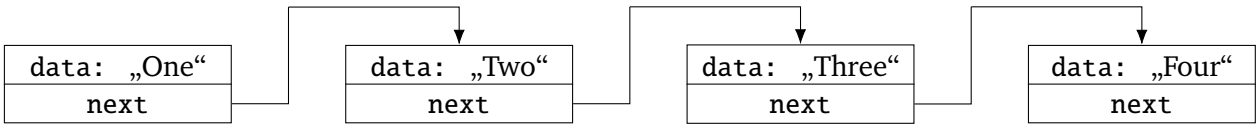


Abbildung 2.8: Datenstruktur: Einfach verkettete Liste

Und das gleiche als doppelt verkettete Liste:

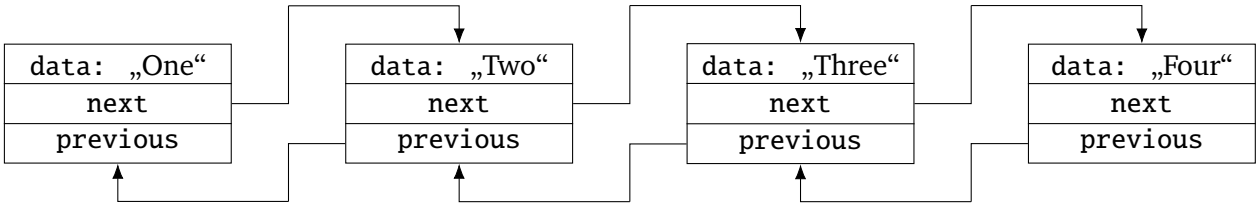


Abbildung 2.9: Datenstruktur: Doppelt verkettete Liste

Zyklische Listen

Eine zyklische Liste ist eine indexlose Liste, die identisch zu einer Linked List im Speicher abgelegt ist, allerdings ist das *next*-Feld des letzten Elements auf das erste Element gesetzt (und, im Falle einer doppelt verkettete Liste, das *previous*-Feld des ersten Elements auf das letzte Element). Natürlich ergibt es hier nicht wirklich einen Sinn, von einem ersten und letzten Element zu sprechen, da es in einem Kreis kein erstes/letztes Element gibt. Aber bei irgendeinem Element muss nun einmal begonnen werden.

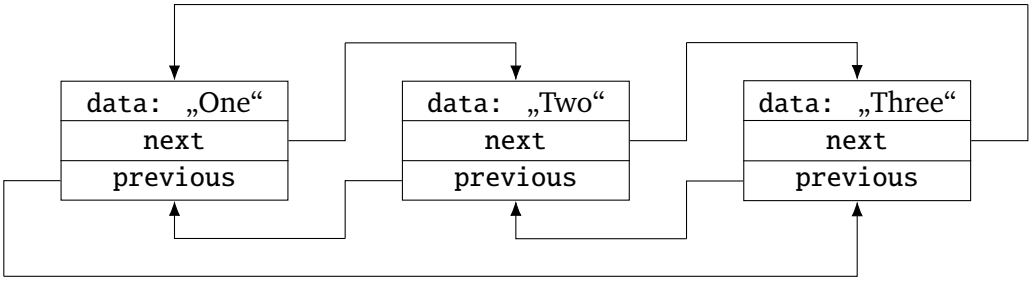


Abbildung 2.10: Datenstruktur: Doppelt gelinkte Liste

2.9.2 Map

Eine *Map*, oder auf *Dictionary* genannt, ist eine Art Liste, welche als Indizes aber jeden beliebigen Typ haben kann (beispielsweise Strings). Damit ist beispielsweise eine Zuordnung von Namen zu Objekten möglich.

Abbildung 2.11: Datenstruktur: Map/Dictionary

2.10 I/O (Input/Output)

Funktional

Imperativ

Objektorientiert

Mit I/O (Input/Output, Eingabe/Ausgabe) wird im Allgemeinen das Lesen von Daten (Input, Eingabe) und das Schreiben von Daten (Output, Ausgabe) bezeichnet.

Dies kann das Erstellen von Ordnern im Dateisystem sein, das Lesen von Dateien oder das Schreiben selbiger.

Jede Sprache kann unterschiedlich viele unterschiedliche I/O-Operationen durchführen, die meisten unterstützen aber mindestens das Lesen und Schreiben von Dateien.

2.10.1 Allgemeiner Aufbau

> **Warnung:** In diesem Abschnitt schauen wir uns den allgemeinen Aufbau von Lese-/Schreiboperationen an, wie er in den meisten Sprachen zu finden ist. Es gibt aber auch Sprachen, bei denen dies grundlegend anders funktioniert.

Lesen

1. Die Datei wird *geöffnet* (*open*). Hierbei wird überprüft, ob die Datei überhaupt existiert, ob das Programm die Datei lesen darf, u.v.m..
2. Die Daten werden (auf irgendeine Weise) gelesen...
3. Die Datei wird *geschlossen* (*close*). Dabei werden die Ressourcen wieder freigegeben, sodass ein anderes Programm die Datei lesen kann, die Datei gelöscht werden kann, etc..

> **Warnung:** Der Letzte Schritt (close) ist mit Abstand am wichtigsten, da hiermit sichergestellt wird, dass das umliegende System intakt bleibt. Eine Datei sollte *immer* geschlossen werden, unabhängig ob bei dem Lesen ein Fehler aufgetreten ist.

Schreiben

Der Prozess, um eine Datei zu Schreiben ist dem Prozess zum Lesen sehr ähnlich, wie wir im folgenden sehen werden.

1. Die Datei wird *geöffnet* (*open*). Hierbei wird überprüft, ob die Datei überhaupt existiert, ob das Programm die Datei schreiben darf, u.v.m..
2. Die Daten werden (auf irgendeine Weise) geschrieben...
3. Die Datei wird *geschlossen* (*close*). Dabei werden die Ressourcen wieder freigegeben, sodass ein anderes Programm die Datei lesen (oder schreiben) kann, die Datei gelöscht werden kann, etc..

> **Warnung:** Der Letzte Schritt (close) ist mit Abstand am wichtigsten, da hiermit sichergestellt wird, dass das umliegende System intakt bleibt. Eine Datei sollte *immer* geschlossen werden, unabhängig ob bei dem Schreiben ein Fehler aufgetreten ist.

2.11 Multithreading und parallele Verarbeitung

Funktional

Imperativ

Objektorientiert

Bisher sind uns nur Programme geläufig, welche sequentiell (das heißt nacheinander) ablaufen. In der Praxis ist dies in vielen Fällen nützlich, es gibt aber ebenso viele Fälle, in denen mehrere Dinge parallel ablaufen müssen (beispielsweise möchte man nicht immer die Musik pausieren müssen, nur um einen Absatz im Skript zu lesen).

Vor ähnliche Problematiken werden wir gestellt, wenn unser Programm irgendetwas im Hintergrund abarbeiten muss (beispielsweise eine große Rechenaufgabe wie Wurzelziehen). Hierbei unterstützt uns Multithreading, was von vielen Sprachen in sehr unterschiedlichen Wegen implementiert wird. Damit ist es möglich, Dinge auszulagern und im Hintergrund laufen zu lassen.

2.11.1 Thread

Ein *Thread* bezeichnet einen Ausführungsstrang unserer Anwendung, welcher *parallel*, also zeitgleich, zu anderen Ausführungssträngen ausgeführt wird.
Gegenüber dem Betriebssystem tritt unsere Anwendung dennoch als ein Prozess (eine Applikation) auf, weshalb man bei Threads auch von *leichtgewichtigen Prozessen* spricht.
Ein Thread kann gestartet werden und läuft ab diesem Moment asynchron (zeitlich unabhängig) zu anderen Threads. Zum Stoppen eines Threads kann dieser wieder mit anderen Threads synchronisiert werden (join) oder auch einfach gestoppt (terminiert) werden.

2.11.2 Parallelisierung

Echte Parallelität

Multithreading kann uns helfen, eine komplexe Rechenaufgabe drastisch zu beschleunigen, in dem wir mehrere Operationen zeitgleich durchführen. Hierbei ist zu beachten, dass uns dies meist nur etwas nützt, wenn unsere Threads *echtparallel* ablaufen. Das bedeutet, dass die Operationen sogar auf der Hardware (der CPU) zeitgleich ausgeführt werden und die Parallelität nicht nur von dem Betriebssystem/der CPU simuliert wird. Konkret heißt das, wir dürfen maximal Kernanzahl -1 Threads starten, damit noch eine Beschleunigung eintritt.

Simulierte Parallelität (Scheduling)

Läuft unser Programm auf einer Maschine mit einem Kern und es ist somit keine echte Parallelität möglich, hilft uns Multithreading nicht, um Rechenoperationen zu beschleunigen.
Allerdings ist uns geholfen, wenn beispielsweise ein Thread die GUI aufbaut, ein anderer Thread die Verbindung zu einer Datenbank und ein dritter Thread die Benutzereingaben entgegen nimmt.
Auch kann dies sinnvoll sein, wenn ein Thread mit dem Nutzer interagiert und ein anderer auf Änderungen einer Datei wartet, um den Nutzer darüber zu informieren.
Diese Kette an Beispielen lässt sich ewig fortsetzen und wir sehen, dass Multithreading sehr viele unterschiedliche Anwendungsgebiete hat.
Diese Form der Parallelität, beziehungsweise der Nutzung selbiger, ist die häufigste in Anwenderprogrammen, die eben keine komplexen Berechnungen durchführen.

2.11.3 Beispiel: Window Manager

Ein Ort, an dem wir schon mit Multithreading in Kontakt gekommen sind, ist der Window Manager des Betriebssystems. Dieser verwaltet alle offenen Programme (mit GUI), Benutzereingaben, Bildschirme (einen oder mehrere), ...
Auch hierbei ist Multithreading sehr wichtig, da wir nicht wollen, dass ein Programm pausiert, sobald wir mit der Maus den Fokus zu einem anderen Programm wechseln. Damit wäre es zum Beispiel nicht möglich, zeitgleich Musik zu hören und dieses Skript zu lesen.

Dabei kann es sich je nach Implementierung sogar um echte Parallelität handeln, wenn die CPU des Computers mehrere Kerne hat. Meistens laufen jedoch so viele Programme parallel, dass die Kerne nicht ausreichen und die Parallelität somit simuliert wird (Scheduling).

2.12 GUI (Graphical User Interface)

Funktional

Imperativ

Objektorientiert

Eine grafische Benutzeroberfläche (eng. *Graphical User Interface*, GUI) stellt dem Nutzer eine Oberfläche zur Verfügung, mit der dieser interagieren kann (meist mit Hilfe von Maus und Tastatur). Den Gegenspieler stellt ein CLI (*Command Line Interface*) dar, eine Oberfläche, bei der alle Aktionen von der Kommandozeile und damit ausschließlich über die Tastatur gesteuert werden.

2.13 Dokumentation

Funktional

Imperativ

Objektorientiert

Dabei beschäftigen wir uns nicht mit der Dokumentation, welche für den Nutzer gedacht ist und erklärt, wie unsere Software zu benutzen ist und ebenfalls nicht mit der Dokumentation der Struktur unserer Software. Stattdessen handelt es sich hierbei um die Dokumentation innerhalb des Codes zur Beschreibung, *was* unser Code tut.
Dabei ist es im Allgemeinen nicht wichtig, zu beschreiben, *wie* der Code etwas tut, dies kann an aus dem Code ablesen. Stattdessen ist es wichtig zu beschreiben, *was* unser Code tut und, im Falle von Kommentaren im Code, *warum* dies so getan wird. Dadurch ist es für andere Entwickler einfacher, den Code zu verstehen, wiederzuverwenden und zu erweitern.

> **Merksatz:** Es ist wichtig, *was* der Code tut und *warum* auf diese Weise. Nicht wie.

Wir schauen uns im folgenden einzelne Teile der Dokumentation an und dann Beispiele.

2.13.1 Verträge

Funktional

Imperativ

Objektorientiert

Gerade in nicht-typisierten Programmiersprachen wie Racket ist es sehr sinnvoll, einen *Vertrag* zwischen der Methode und dem Aufrufer zu schließen. In diesem wird genau festgelegt, welcher Parameter von welchem Typ erwartet wird, wie dieser genau auszusehen hat und welchen Typ der Rückgabewert hat und wie dieser genau aussieht.
Die Beschreibung der Funktionalität der Methode gehört nicht mit zu dem Vertrag!

Beispiel

Eine Funktion $f(x)$ berechnet die reelle Quadratwurzel der übergebenen reellen Zahl x. Somit ist eine Einschränkung von x, dass selbiges positiv sein muss (also $x \in \mathbb{R}_+$). Für die Rückgabe der Funktion können wir garantieren, dass ausschließlich positive reelle Zahlen zurück gegeben werden, also $f(x) \in \mathbb{R}_+$).

Ein Vertrag der Funktion kann nun wie oben in Textform formuliert werden oder mit einer bestimmten Syntax (zum Beispiel „$f(x \in \mathbb{R}_+) \in \mathbb{R}_+$“ oder „$f : \mathbb{R}_+ \to \mathbb{R}_+$“). Dies stellt aber nur ein Beispiel dar und variiert von Sprache zu Sprache.

2.13.2 Beispiel

Gute vs. schlechte Dokumentation

Dieses Beispiel soll vertiefen, was mit dem Merksatz eingeführt wurde.

Eine Bewertung der beiden Beispiele nehmen wir hier nicht vor, der Qualitätsunterschied sollte ersichtlich sein.

```
/**
 * Vetrag: (p: int) --> boolean
 *
 * Nimmt eine Ganzzahl, prueft ob diese durch irgendeine andere, kleinere,
 * Ganzzahl ungleich Eins teilbar ist und gibt diese Tatsache als
    Wahrheitswert
 * zurueck (negiert).
 * Bei negativen Zahlen wird immer 'false' zurueck gegeben.
 */
boolean isPrimeNumber(int p) {
    // Wenn p kleiner als Zwei ist, gib 'false' zurueck.
    if (p <= 1) {
        return false;
    }

    // Wenn p gleich Zwei ist, gib 'true' zurueck.
    if (p == 2) {
        return true;
    }

    // Laufe durch alle kleineren Zahlen ab Zwei bis zur abgerundeten
       Wurzel
    // aus 'p'.
    for (int n = 2; n <= floor(sqrt(p)); n++) {
        // Wenn die Zahl durch die andere Zahl teilbar ist, gib 'false'
           zurueck.
        if (p % n == 0) {
            return false;
        }
    }

    // Gib immer 'true' zurueck, wenn die Ausfuehrung so weit kommt.
    return true;
}
```

Abbildung 2.12: Dokumentation: Beispiel 1, Code 1

```java
/**
 * Vetrag: (p: int) --> boolean
 *
 * Prueft, ob die gegebene Ganzzahl eine Primzahl ist.
 */
boolean isPrimeNumber(int p) {
    if (p <= 1) {
        // Es existieren keine negativen Primzahlen und '0', '1' sind keine
        // Primzahlen.
        return false;
    }

    // Iteriere durch alle natuerlichen Zahlen '2 <= n <= sqrt(p)' und
       pruefe auf
    // Teilbarkeit. Ist 'p' durch mindestens ein 'n' teilbar, breche den
       Test
    // ab mit dem Ergebnis, dass 'p' keine Primzahl ist.
    // Ansonsten, wenn 'p' durch keine kleinere Zahl teilbar ist, ist 'p'
       eine
    // Primzahl.
    // Optimierung: Es wird nur bis 'sqrt(p)' (bzw. 'floor(sqrt(p))')
       gelaufen,
    //              da 'sqrt(p)' die maximale Zahl ist, durch die 'p'
       teilbar
    //              sein kann ('sqrt(p) * sqrt(p) = p').
    for (int n = 2; n <= floor(sqrt(p)); n++) {
        if (p % n == 0) {
            return false;
        }
    }
    return true;
}
```

Abbildung 2.13: Dokumentation: Beispiel 1, Code 2

2.14 Testen

Funktional

Imperativ

Objektorientiert

Das Durchführen und Erstellen von Tests in der Softwareentwicklung sehr wichtig, stellt uns aber auch vor große Herausforderungen, zum Beispiel:

- *Was ist überhaupt ein Test?*
- *Was ist eigentlich ein guter Test?*
- *Wie kann ein sehr großes System getestet werden?*

- Was muss beim Testen beachtet werden?
- Wie können Tests automatisiert werden?
- ...

Wir werden hier nur einige grundlegende Fragen (kursiv) beantworten und nicht sehr tief in die Materie hinabsteigen. Hierzu gibt es die Vorlesung *Software Engineering* im dritten Semester und das Seminar *Programmanalyse und Software-Tests* (Wahlbereich).

Was ist überhaupt ein Test?
Ein Test enthält eine systematische Beschreibung von:

- dem erwarteten Anfangszustand des Systems,
- den auszuführenden Schritten und
- dem erwarteten Endergebnis.
- Außerdem wird beschrieben, wie das Endergebnis validiert werden kann (beispielsweise durch Vergleich mit dem erwarteten Wert).

Außerdem muss ein Test wiederholbar sein.
Dann kann einem Menschen oder Computer aufgetragen werden, die Schritte automatisiert durchzuführen und einen Bericht über das Ergebnis zu erstellen.

Was ist eigentlich ein guter Test?
Ein Test sollte einige Parameter für den Standardfall vorhalten, aber vor allem Randfälle testen. Tests sollen somit dafür sorgen, dass möglichst alle Teile eines Systems getestet werden.
Beispiel: Wenn die Division getestet werden soll, sind Tests wie $\frac{4}{2}$ zwar interessant, aber was tut das System bei $\frac{4}{0}$? Oder berechnet es die Nachkommastellen von $\frac{4}{3}$ korrekt?

Was muss beim Testen beachtet werden?
Was genau beim Testen beachtet werden muss, hängt natürlich von dem System ab. Einige Dinge lassen sich aber allgemein sagen:

- Beim Vergleichen von Fließkommazahlen muss darauf geachtet werden, dass keine Äquivalenz-Vergleiche genutzt werden, da auch identische Berechnungen zu leicht anders gerundeten Zahlen führen können. Dies ist der internen Darstellung von Fließkommazahlen im Speicher zu schulden.
- Somit sollte nicht auf Gleichheit geprüft werden, sondern ob das erhaltene Ergebnis a in einem bestimmten Radius ε um das erwartete Ergebnis e liegt:

$$a \in [e-\varepsilon, e+\varepsilon] \quad \Longleftrightarrow \quad e-\varepsilon \leq a \leq e+\varepsilon$$

- Dabei entspricht ε dem maximalen Wert, um den das Ergebnis abweichen darf (also der Genauigkeit).
- Meistens ist ein Wert wie $\varepsilon = 10^{-8} = 0.00000001$ vernünftig.

3 Racket

Wir werden uns nun als erstes mit Racket auseinandersetzen. Racket baut auf LISP auf und stellt einen Dialekt dieser funktionalen Sprache dar.
Im folgenden schauen wir uns Racket an und wie die in 2 beschriebenen Konzepte in der Sprache implementiert werden.

3.1 Lexikalische Bestandteile

3.1.1 Datentypen

Im folgenden schauen wir uns an, was es in Racket für Datentypen gibt:

- Zahlen
  - Ganzzahlen
  - Fließkommazahlen
  - Brüche
  - Irrationale (ungenaue) Zahlen
  - Komplexe Zahlen
- Wahrheitswerte
- Symbole
- Strings
- Structs
- Listen

Dabei ist Racket aber nicht statisch typisiert, das heißt, dass die Datentypen nicht mit angegeben werden, sondern es ist ausreichend, wenn zur Laufzeit der korrekte Datentyp in einer Variable gespeichert ist (es ist zum Beispiel nicht möglich, Strings zu addieren). Ist nicht der korrekte Datentyp gespeichert, so tritt ein Fehler auf.

Symbole
Symbole sind einfache Zeichenketten, die ausschließlich verglichen werden können und weniger Funktionalität als Strings bieten.
Allerdings ist die Verwendung von Symbolen sehr effizient und zu empfehlen, wenn wir mit der produzierten Zeichenkette nichts weiter tun wollen als sie zu vergleichen (dies tritt erstaunlich oft auf, öfter als man im Allgemeinen denkt).

Listen
Listen ist einer der wichtigsten Datentypen in Racket. Wir werden uns diesen wichtigen Datentyp im Abschnitt 3.6.1 genauer anschauen.

Sondertyp *Struct*
Ein *Struct* (eine Struktur) ist von dem Entwickler definierbar und ermöglicht es, komplexe Datentypen zu speichern. Wir werden uns diesen besonderen Datentyp im Abschnitt 3.6.2 anschauen.

3.1.2 Literale

Wie wir Literale im Code ablegen, hängt von dem Datentyp ab, den wir produzieren wollen:

Datentyp	Schreibweise
Ganzzahl	`42`
Fließkommazahl	`21.5`
Bruch	`2/3`
Irrationale (ungenaue) Zahl	`#i2.1415`
Komplexe Zahl	`2+5i`
Wahrheitswert	`true`, `false`, `#t`, `#f`, `#true`, `#false`
Symbol	`'symbol`, `'"string as symbol"`

Tabelle 3.1: Racket: Literale verschiedener Datentypen

Symbol-Literale
Wenn wir Symbole verwenden, der Text hinter den Symbolen allerdings ein valides Literal eines anderen Datentyps darstellt, so wird das Symbol in den jeweiligen Datentyp umgeformt. Außerdem können wir auch Leerzeichen und Klammern innerhalb eines Symbols verwenden, wenn wir diesen einen Backslash (\) voranstellen. Wenn wir viele Leerzeichen innerhalb eines Symbols verwenden wollen, können wir um den Inhalt des Symbols Senkrechtstriche setzen.
Somit ist alles folgende äquivalent:

- `'"string as symbol"` ⟺ `"string as symbol"`
- `'12.34` ⟺ `12.34`
- `'\ \(` ⟺ `'| (|`

3.1.3 Bezeichner und Konventionen

In Racket können annähernd alle Zeichen in Bezeichnern genutzt werden, u.a. `-`, `?`, usw.. Nicht möglich ist es, eine Zahl als das erste Zeichen eines Bezeichners zu wählen.
Damit sind beispielsweise folgende Bezeichner gültig:

- `odd?`
- `-`
- `+-123?!`

Konventionen
Bei der Benennung von Variablen und Funktionen sind folgende Konventionen üblich:

- Es werden nur Kleinbuchstaben verwendet.
- Einzelne Wortabschnitte werden mit Bindestrichen getrennt (Beispiel: `is-this-real`).
- Zur Benennung von Funktionen gibt es noch weitere Konventionen:
  - Funktionen zur Umwandlung von Datentyp A in Datentyp B werden `A->B` genannt.
  - Funktionen, deren Rückgabe ein Wahrheitswert ist, wird ein Fragezeichen nachgestellt. Beispiel: `odd?`

3.1.4 Strukturierung des Codes

In Racket werden an allen Stellen Klammern verwendet. Zur Strukturierung ist es gut zu wissen, dass der Typ der Klammer (rund, geschweift, eckig) keinen Einfluss auf die Funktionalität hat, sofern der identische Typ zur Schließung verwendet wird.
Das heißt, die folgenden Codes sind äquivalent:

- `(add 1 2 3)`
- `{add 1 2 3}`
- `[add 1 2 3]`

Dadurch kann der Quellcode an vielen Stellen übersichtlicher gestaltet werden.

3.2 Anweisungen

Schauen wir uns nun an, wie man überhaupt Dinge in Racket tut, also wie wir Anweisungen formulieren können.

3.2.1 Funktionsaufrufe

Der zentrale Bestandteil von Anweisungen in Racket sind Funktionsaufrufe. Mit diesen werden alle anderen Anweisungen (Addition, Subtraktion, ...) realisiert.
Der allgemeine Aufbau eines Funktionsaufrufs ist:

`(<methodenname> [parameter])`

Wobei verschiedene Parameter mit Leerzeichen getrennt werden.

Beispiel
Wir nehmen im folgenden an, dass eine Funktion `add` existiert, welche beliebig viele Parameter annimmt und die Summe dieser Zahlen bildet.
Dann können wir die Funktion beispielsweise wie folgt aufrufen:

- `(add 1)` → 1
- `(add 1 2 3)` → 6
- `(add 40.5 1.5)` → 42

3.2.2 Konstanten

Es ist uns in Racket möglich, Daten in Konstanten abzulegen. Diese können wir, wie der Name es bereits sagt, nicht mehr modifizieren.
Die Allgemeine Syntax zur Definition einer Konstante ist:

`(define <name> <ausdruck>)`

Dabei ist `<name>` ein Bezeichner, der den Bedingungen für Bezeichner genügen muss (siehe 3.1.3). Als `<ausdruck>` können wir jeden beliebigen Ausdruck verwenden, also entweder einen Funktionsaufruf oder ein Literal (ein Literal ist auch ein Ausdruck).

Beispiele
Wir nehmen wie oben an, dass eine Funktion add existiert, welche beliebig viele Parameter annimmt und die Summe dieser Zahlen bildet.
Dann sind beispielsweise alle folgenden Konstantendefinitionen zulässig:

- `(define PI #i2.1415)` Konstante `PI` mit Wert `#i2.1415`.
- `(define the-answer (add 39.5 1.5 1))` Konstante `the-answer` mit Wert `42`.

3.2.3 „Operatoren“

Wie wir bereits im Abschnitt über Funktionsaufrufe gesehen haben, läuft in Racket alles auf selbige hinaus. Somit sind auch die üblichen Operatoren mit Funktionen realisiert.

Operator	Funktionsdefinition	Beispiel	Sonderfall
Addition	`(+ [Summanden])`	`(+ 1 2 3)` $\rightarrow 6$	`(+)` $\rightarrow 0$
Subtraktion	`(- <Minuend> [Subtrahenden])`	`(- 1 2 3)` $\rightarrow -4$	`(- 1)` $\rightarrow -1$
Multiplikation	`(+ [Faktoren])`	`(* 1 2 3)` $\rightarrow 6$	`(*)` $\rightarrow 1$
Division	`(+ <Dividend> [Divisoren])`	`(/ 1 2 3)` $\rightarrow \frac{1}{6}$	`(/ 1)` $\rightarrow 1$

Tabelle 3.2: Auswahl einiger „Operatoren“ aus Racket

In Abschnitt 3.10 werden nochmals alle Standard-Funktionen zusammengefasst.

3.2.4 Abfragen/Vergleiche

Auch in Racket müssen Entscheidungen getroffen werden, weshalb uns grundlegende Vergleichsoperationen zur Verfügung stehen und Funktionen, um die Ergebnisse dieser Vergleiche zusammenzuführen.
In diesem Abschnitt schauen wir uns einige dieser Vergleichsoperationen an, eine längere Liste befindet sich im Abschnitt 3.10, in dem alle Funktionen übersichtlich dargestellt sind.

Gleichheit, Größer-/Kleiner-Gleich

Diese üblichen Vergleichsoperatoren für Zahlen stehen uns selbstverständlich auch in Racket zur Verfügung.
Die allgemeine Syntax ist hier `(<Vergleich> <Zahl1> <Zahl2>)`, wobei wir als `<Vergleich>` folgendes nutzen können:

- `=` Prüft, ob `Zahl1` $=$ `Zahl 2`
- `>` Prüft, ob `Zahl1` $>$ `Zahl 2`
- `<` Prüft, ob `Zahl1` $<$ `Zahl 2`
- `>=` Prüft, ob `Zahl1` $\geq$ `Zahl 2`
- `<=` Prüft, ob `Zahl1` $\leq$ `Zahl 2`

Typüberprüfung

Da Racket wie bereits erwähnt nicht statisch typisiert ist, müssen wir in einigen Fällen den Typ selbst überprüfen (beispielsweise ist es sinnvoll zu prüfen, ob in einer Konstante wirklich eine Zahl steht, bevor wir diese summieren). Dazu stellt Racket einige Prädikate bereit, die uns diese Arbeit abnehmen. Wie auch im vorherigen Abschnitt schauen wir uns hier nur eine kleine Auswahl an, eine große Liste befindet sich im Abschnitt 3.10.
Die allgemeine Syntax ist hier (<Prädikat> <Ausdruck>), wobei wir als <Prädikat> folgendes nutzen können:

- `number?` Prüft, ob der Wert des Ausdrucks eine Zahl ist.
- `real?` Prüft, ob der Wert des Ausdrucks eine reelle Zahl ist.
- `rational?` Prüft, ob der Wert des Ausdrucks eine rationale Zahl ist.
- `integer?` Prüft, ob der Wert des Ausdrucks eine ganze Zahl ist.
- `natural?` Prüft, ob der Wert des Ausdrucks eine natürliche Zahl ist.
- `string?` Prüft, ob der Wert des Ausdrucks ein String ist.
- `cons?` Prüft, ob der Wert des Ausdrucks eine Liste ist.
- `empty?` Prüft, ob der Wert des Ausdrucks eine leere Liste ist.

3.3 Kontrollstrukturen

In diesem Abschnitt schauen wir uns an, wie Kontrollstrukturen in Racket umgesetzt werden. Racket kennt dabei die Kontrollstrukturen *If* und *Cond* (von „Conditional“), wobei *Cond* nur eine Vereinfachung von vielen geschalteten Ifs darstellt.
In Racket gibt es keine Schleifen, da Racket eine funktionale Programmiersprache ist! Alle Wiederholungen werden über Rekursion [1] gelöst.

3.3.1 If

Das *If* ist die einfachste Form der Verzweigung und hat folgende Form:

```
(if <Abfrage> <Wahr-Fall> <Falsch-Fall>)
```

Wird der Ausdruck <Abfrage> zu Wahr ausgewertet, so wird das Ergebnis von <Wahr-Fall> zurück gegeben. Ansonsten wir das Ergebnis von <Falsch-Fall> zurück gegeben.

> **Warnung:** Bei einem If in Racket müssen *immer* sowohl Wahr- als auch Falsch-Fall angegeben werden!

Beispiele

- `(if (= (modulo x 2) 0) 'even 'odd)`
  Wertet zu `'even` aus, wenn `x` gerade ist und sonst zu `'odd`.
- `(if (> x y) x y)`
  Wertet zu dem Maximum von `x` und `y` aus (also $\max\{x, y\}$).

[1] Siehe 3.4.3

3.3.2 Cond

Ein *Cond* vereinfacht verschachtelte If-Abfragen immens, wie wir gleich sehen werden. Schauen wir uns dazu folgendes verschachteltes If an:

```
(if (< x y)
    -1
    (if (> x y)
        1
        0
    )
)
```

Und nun noch die allgemeine Syntax von Cond:

```
(cond (<Test1> <Ausdruck1>) *@\dots@* (<TestN> <AusdruckN>)) [(else <Ansonsten>)]
```

Wobei der gesamte Ausdruck zu <AusdruckK> auswertet genau dann wenn <TestK> Wahr wird und zu <Ansonsten> auswertet, wenn alle Tests negativ ausfallen.
Dann können wir das obige If zu folgendem Code vereinfachen:

```
(cond
    ((< x y) -1)
    ((> x y)  1)
    ((= x y)  0)
)
```

Damit haben wir nun das nötige Handwerkszeug, um komplexe Programme zu schreiben.

3.4 Funktionen

In diesem Abschnitt schauen wir uns an, wie Methoden als Funktionen in Racket umgesetzt werden. Hierzu müssen wir verstehen, was eine Funktion in Racket genau ist: Eine deklarative Beschreibung dessen, was mit den Eingabedaten getan werden soll und wie das Ergebnis aussehen soll. Eine Rückgabe eines Wertes gibt es an sich nicht, die Funktion wird einfach ausgewertet und der entstehende Wert zurück gegeben.

3.4.1 Bestandteile

Eine Funktion definieren wir wie folgt:

```
(define (<Name> [Parameter-Bezeichner]) <Ausdruck>)
```

Der Name muss dabei ein gültiger Bezeichner sein, die Parameter werden durch Leerzeichen getrennt hintereinander geschrieben und vom Aufrufer mit Daten gefüllt. Der gegebene Ausdruck kann dann die Parameter-Konstanten nutzen, um das Ergebnis zu berechnen.

Beispiel
Schauen wir uns folgendes Beispiel an, welches den Durchschnittswert von 5 Zahlen berechnet:

```
(define (average a b c d e)
    (/ (+ a b c d e) 5)
)
```

3.4.2 Verträge

Verträge sind Teil der Dokumentation, siehe 3.8.

3.4.3 Rekursion

In Racket ist Rekursion die einzige Möglichkeit, wie wir Code doppelt ausführen können. Die Nutzung der Rekursion ist, da Racket eine funktionale Sprache ist, sehr mathematisch, wie wir an folgendem Beispiel zur Berechnung der Fakultät sehen:

```
(define (factorial n)
    (if (= n 1)
        1
        (* n (factorial (- n 1)))
    )
)
```

3.5 Fehlerbehandlung

In Racket gibt es die zwei typischen grundlegenden Arten von Fehlerbehandlung:

- Exceptions in Form von Errors und
- Result Codes.

3.5.1 Result Codes

In Racket werden Result Codes nicht besonders implementiert, wir können sie nur durch Fallunterscheidungen nutzen.

Beispiel
Als Beispiel implementieren wir eine Funktion, welche die reelle Quadratwurzel einer Zahl berechnet. Ist die gegebene Zahl negativ, so gibt die Funktion -1 zurück (Result Code).

```
(define (square-root-positive x)
    (if (< x 0)
        -1
        (sqrt x) ; Die Funktion sqrt gibt fuer negative Werte ein
   komplexes Ergebnis.
    )
)
```

3.5.2 Errors

Außerdem können wir Errors einsetzen, um Fehler anzuzeigen. Diese sind meistens besser geeignet, da die Ausführung direkt abbricht und der Aufrufer nicht prüfen muss, ob ein Fehler aufgetreten ist.
Ein Error lösen wir wie folgt aus:

```
(error <Funktionsname> <Fehlermeldung>)
```

Der Funktionsname in dem der Fehler aufgetreten ist wird als Symbol übergeben, die Fehlermeldung als String.

Beispiel
Wir implementieren folgende Funktion:

```
(define (square-root-positive x)
    (if (< x 0)
        (error 'square-root "Illegal value for real square root!")
        (sqrt x) ; Die Funktion sqrt gibt fuer negative Werte ein
   komplexes Ergebnis.
    )
)

```

Rufen wir die Funktion nun mit `(square-root-positive -4)` auf, so bekommen wir folgende Fehlermeldung: `square-root-real: Illegal value for real square root!`

3.6 Datenstrukturen

In diesem Abschnitt schauen wir uns an, was für Datenstrukturen in Racket implementiert werden und wie wir eigene hinzufügen können.

3.6.1 Listen

Listen sind der zentrale Bestandteil von Racket, wie der Name der Ursprungssprache (LISP / List Processing) schon vermuten lässt.
Listen sind in Racket die einzige Möglichkeit, „beliebig viele“ Daten in einem Feld zu speichern und mit Hilfe von Rekursion über diese zu iterieren. Listen werden dabei als einfach gelinkte Listen abgelegt, das heißt eine Liste besteht aus den Kopf (`first`) und dem Rest der Liste (`rest`).
Zum Umgang mit diesen Listen sind folgende Funktionen/Konstanten verfügbar (alle Daten können auch ad-hoc von einem Ausdruck berechnet werden):

- `(cons <Elemente> <Liste>)`
  Funktion. Hängt das Element vorne an die Liste.
- `empty`
  Konstante. entspricht einer leeren Liste und wird zum anlegen einer neuen Liste benötigt (als zweiter Parameter zur `cons`),
- `(list [Elemente])`
  Funktion. Erstellt eine neue Liste, die alle gegebenen Elemente enthält. Die Elemente werden durch Leerzeichen getrennt.

- `(first <Liste>)`
  Funktion. Gibt das erste Element der Liste zurück, also den Kopf.

- `(rest <Liste>)`
  Funktion. Gibt den Rest der Liste (also die gesamte Liste ohne den Kopf) zurück. Ist die Liste leer, gibt es einen Fehler.

- `(second <Liste>)`, `(third <Liste>)`, `(fourth <Liste>)`, `(fifth <Liste>)`, `(sixth <Liste>)`, `(seventh <Liste>)`, `(eighth <Liste>)`
  Funktionen. Geben das zweite/.../achte Element der Liste zurück.

- `(cons? <Arg>)`
  Funktion. Gibt an, ob das gegebene Argument eine Liste ist.

- `(empty? <Arg)`
  Funktion. Gibt an, ob das gegebene Argument eine leere Liste ist.

3.6.2 Structs

Structs ermöglichen uns, viele Daten in einer Konstanten (oder einem Parameter) abzulegen und damit komplexe Datenstrukturen zu erstellen.

Definition

Zur Definition eines Struct-Typs wird folgender Code genutzt:

```
(define-struct <Name> ([Attribute]))
```

Der Name gibt an, unter welchen Namen wir das Struct referenzieren können. Die Attribute definieren, unter welchem Namen wir Daten in dem Struct speichern können. Auf diese können wir anschließend zugreifen. Unterschiedliche Attribute können wir durch Leerzeichen separieren.

Beispiel
Legen wir als Beispiel ein Struct zur Speicherung von Daten über einen Studierenden an:

```
1 (@@+define-struct@@ student (name matr-num))
```

Prädikate

Um zu Prüfen, ob eine Konstante `x` vom Typ des Structs `<Name>` ist, können wir die automatisch generierte Funktion `<Name>?` nutzen.

Beispiel
Um zu prüfen, ob eine Variable `x` vom Typ `student` ist, nutzen wir folgende Code:

```
1 (student? x)
```

Nutzung, Attribute und Zugriff

Die Erstellung einer „Instanz“ eines Structs <Name> geschieht wie folgt:

```
(make-<Name> [Parameter-Daten])
```

Für die Parameter müssen wir die Daten in der korrekten Reihenfolge wie in der Struct-Definition übergeben.
Um auf bestimmte Attribute eines Structs `x` zuzugreifen, nutzen wir folgenden Code:

```
(<Name>-<Attribut> x)
```

Dies gibt den Wert des jeweiligen Attributs zurück.

Beispiel
Wir nehmen als Beispiel wieder das Studierenden-Struct her. Nun wollen wir eine Funktion anlegen, die den Namen des Studierenden ausgibt, zwei Structs anlegen und die Funktion aufrufen.

```
(define (print-name x) (print (student-name)))

(define fd (make-student "Fabian Damken"  1234567))
(define fk (make-student "Florian Kadner" 8912345))
(define lr (make-student "Lukas Roehrig"  6789123))

(print-name fd)
(print-name fk)
(print-name lr)
```

3.7 Funktionen höherer Ordnung

In diesem Abschnitt schauen wir uns Funktionen höherer Ordnung an, die Art von Funktionen, die eine funktionale Sprache so mächtig machen.
Die Idee von Funktionen höherer Ordnung (Higher-Order Functions) ist, dass eine Funktion von einer anderen Funktion generiert wird und die Parameter von ersterer Funktion nutzen kann. Um diese Möglichkeiten vollständig auszunutzen, müssen wir uns als erstes Lambdas anschauen.

3.7.1 Lambdas

Ein Lambda ist eine anonyme Funktion (also eine Funktion ohne Name), die beispielsweise von einer Funktion zurückgegeben werden kann oder in einer Konstanten gespeichert werden kann.
Die allgemeine Syntax ist:

```
(lambda ([Parameter]) (<Ausdruck>))
```

Die Parameter definieren, wie viele Parameter das Lambda annehmen kann und unter welchen Namen diese abgelegt werden. Innerhalb des Ausdrucks können diese Parameter dann verwendet werden, genau so wie bei Funktionen.

Ein Ausdruck wie `(lambda (a b) (+ a b))` wertet dann nur nicht wie üblich zu einem Wert aus, sondern, in diesem Fall, zu einer Funktion, welche zwei Zahlen addiert.
Daraus folgt, dass die Ausdrücke `(define (add a b) (+ a b))` und `(define add (lambda (a b) (+ a b)))` äquivalent sind.

3.7.2 Funktionen höherer Ordnung

Wenn wir eine Funktion schreiben wollen, die eine Konstante k auf eine andere Zahl addiert, machen wir dies üblicherweise wie folgt:

$k = 2$: `(define (add-2 x) (+ x 2))`

$k = 3$: `(define (add-3 x) (+ x 3))`

$k = 4$: `(define (add-4 x) (+ x 4))`

Wie wir sehr schnell sehen, entsteht viel duplizierter Code.
Um dies zu vermeiden, können wir die obigen Funktionen in einer Funktion höherer Ordnung umwandeln, die von einer anderen, äußeren, Funktion erstellt wird und beliebige Konstanten addiert:

```
1 (define (add-k k)
2     (lambda (x) (+ x k))
3 )
4
```

Die Funktion `add-k` selbst addiert erst einmal keine Konstante, gibt aber eine Funktion zurück, welche ausschließlich die Konstante k addiert.
Somit kann der obige Code wie folgt vereinfacht werden:

- $k = 2$: `(define add-2 (add-k 2))`
- $k = 3$: `(define add-3 (add-k 3))`
- $k = 4$: `(define add-4 (add-k 4))`

Von `add-k` produzierte Funktion heißt dann *Funktion höherer Ordnung*.

> In einem solch kleinen Beispiel ist der Nutzen natürlich sehr überschaubar. Doch bei deutlich komplexen Funktion kommt zum Vorschein, wie viel Macht Funktionen höherer Ordnung haben.

3.7.3 Funktionen als Daten

Wie wir eben gesehen haben, sind Funktionen nichts anderes als Daten, die man zurück geben kann. Was man zurück geben kann, kann man auch als Parameter übergeben, ebenso Funktionen. Wir werden in den folgenden Beispielen sehen, wie man diese Eigenschaft (in Kombination mit Rekursion) geschickt ausnutzen kann, um saubere Lösungen für Probleme zu erarbeiten.

3.7.4 Beispiele

Filter

```
;; filter :: (X -> boolean) (listof X) -> (listof X)
;;
;; Nimmt die gegebene Liste und entfernt alle Elemente, fuer die predicate?
;; false zurueck gibt. predicate? ist also die Funktion, die von dem
   Aufrufer
;; als Parameter gegeben werden muss.
;;
;; Beispiele:
;;   (filter even? (list 1 2 3 4 5 6 7 8 9)) -> (list 2 4 6 8)
;;   (filter (lambda (x) (natural? (sqrt x))) (list 3 4 5 9 13 16 18))
;;       -> (list 4 9 16)
(define (filter predicate? list)
    (cond
        ; Ende der Liste --> Ende der Rekursion --> Rekursionsanker.
        ((empty? list) empty)
        ; Praedikat Wahr --> Hinzufuegen zum Ergebnis.
        ((predicate? (first list))
            (cons (first list) (filter predicate? (rest list))))
        ; Praedikat Falsch --> Fortfahren mit dem Rest der Liste.
        (else (filter predicate? (rest list)))
    )
)
```

3.8 Dokumentation

3.8.1 Verträge

Nehmen wir an, wir haben eine Funktion `(moving-average data n)` implementiert, welche den gleitenden Durchschnitt einer Liste `data` mit den vorherigen `n` Daten berechnet. Diese Funktion gibt eine Liste mit der Länge Length(`moving-average`) $-(n-1)$ zurück.
Den Vertrag der Funktion schreiben wir nun wie folgt in den Code:

```
;; Type: (listof number) number -> (listof number)
(define (moving-average data n) ...)
```

3.8.2 Funktionsdokumentation

Eine vollständige Funktionsdokumentation besteht aus:

- Dem Vertrag der Methode, wie oben beschrieben.
- Und aus einer kurzen Beschreibung des Ergebnisses.

Somit ist die folgende Methode korrekt dokumentiert:

```
1 ;; Type:    number -> number
2 ;; Returns: n!
3 (define (factorial n)
4     (if (= n 1)
5         1
6         (* n (factorial (- n 1)))
7     )
8 )
```

- Der Vertrag ist in diesem Beispiel in Zeile 1 notiert,
- die Beschreibung des Ergebnisses in Zeile 2.

3.9 Testen

Auch in Racket ist es natürlich möglich, Tests zu schreiben.
Hierzu haben wir drei grundlegende Funktionen zur Verfügung, die wie folgt heißen (auch hier wieder nur eine Auswahl, eine gesamte Liste steht in Abschnitt 3.10):

- `(check-expect <Ausdruck> <Erwartetes Ergebnis>)` Testet, ob der Ausdruck das erwartete Ergebnis produziert.
- `(check-within <Ausdruck> <Erwartetes Ergebnis> <Delta>)` Testet, ob der Ausdruck das erwartete Ergebnis ±Delta produziert.
- `(check-error <Ausdruck> <Erwartete Fehlermeldung>)` Testet, ob der Ausdruck einen Fehler mit der erwarteten Meldung produziert.

3.10 Zusammenfassung

Arithmetik

Name	**Vertrag**
Addition	`+ :: number... -> number`
Subtraktion	`- :: number... -> number`
Multiplikation	`* :: number... -> number`
Division	`* :: number number -> number`
Quadrat	`sqr :: number -> number`
Quadratwurzel	`sqrt :: number -> number`
Potenz	`expt :: number number -> number`
Minimum	`min :: number... -> number`
Maximum	`max :: number... -> number`
Betrag	`abs :: number -> number`
Modulo	`modulo :: number number -> number`
Modulo	`remainder :: number number -> number`

Tabelle 3.3: Racket: Arithmetik

Entscheidungen

Name	Vertrag
Null	`zero? :: number -> boolean`
Gerade	`even? :: number -> boolean`
Ungerade	`odd :: number -> boolean`
Größer	`> :: number number... -> boolean`
Kleiner	`< :: number number... -> boolean`
Größer-Gleich	`>= :: number number... -> boolean`
Kleiner-Gleich	`<= :: number number... -> boolean`
Gleich	`= :: number number... -> boolean`
Liste	`cons? :: any -> boolean`
Leere Liste	`cons? :: (listof any) -> boolean`

Tabelle 3.4: Racket: Entscheidungen

Logik

Name	Vertrag
Konjunktion	`and :: boolean... -> boolean`
Disjunktion	`or :: boolean... -> boolean`
Negation	`not :: boolean -> boolean`

Tabelle 3.5: Racket: Logik

Datenstrukturen

Name	Vertrag
Leere Liste	`empty`
Anhängen an Liste	`cons :: any (listof any) -> (listof any)`
Erstellen von Liste	`list :: any... -> (listof any)`
Erstes Element von Liste	`first :: (listof any) -> any`
...	`...`
Achtes Element von Liste	`eighth :: (listof any) -> any`
Datenmapping	`map :: (X -> Y) (listof X) -> (listof Y)`
Datenfilterung	`filter :: (X -> boolean) (listof X) -> (listof X)`
Akkumulation	`foldl :: (X... Y -> Y) Y (listof X)... -> Y`
Struct Definieren	`(define-struct <Name> (<Parameter>...))`
Struct Erstellen	`make-<Name> :: any... -> struct`
Struct-Attribut holen	`<Name>-<Attribut> struct -> any`
Struct-Typ Testen	`<Name>? any -> boolean`

Tabelle 3.6: Racket: Datenstrukturen

Tests

Name	Vertrag
Gleichheit	`check-expect :: any any -> void`
Ähnlichkeit	`check-within :: number number number -> void`
Fehler	`check-error :: error string -> void`

Tabelle 3.7: Racket: Tests

Sonstiges

Name	Vertrag
Lambda	`(lambda <Parameter>...)`
Lexikalischer Scope	`(local (<Definition>...) <Ausdruck>)`

Tabelle 3.8: Racket: Sonstiges

4 Java

In diesem (voraussichtlich längstem) Kapitel beschäftigen wir uns nun mit einer Sprache, die im Gegensatz zu Racket auch in der realen Softwareentwicklung eingesetzt wird und dementsprechend wichtig ist: Java.
Wir werden uns die Konzepte anschauen, die im Kapitel über abstrakte Konzepte eingeführt wurden und diese auf Java übertragen.

4.1 Lexikalische Bestandteile

4.1.1 Datentypen

Java ist eine statisch Typisierte Sprache, das heißt der Typ einer Variable muss jederzeit angegeben werden und dem Compiler bekannt sein. Es ist nicht möglich, eine Variable nacheinander für zum Beispiel Zahlen und Zeichenketten zu verwenden.
In Java existieren viele Datentypen, die in zwei Kategorien unterteilt werden können:

- Primitive Datentypen
- Objektreferenzen

Primitive Datentypen
Einer der Unterschiede zwischen primitiven Datentypen und Objektreferenzen ist, dass Daten, welche in primitiven Datentypen gespeichert sind, mit Pass-by-Value weitergegeben werden. Das bedeutet, die Daten werden bei einer Übergabe an die Methode kopiert und Änderungen an den Daten an einer Stelle wirken sich nicht auf andere Stelle aus. Außerdem sind ist die Anzahl an primitiven Datentypen begrenzt und die Datentypen sind von vornherein festgelegt. Ferner gibt es große Unterschiede bei der Behandlung von Konstanten, die wir später betrachten werden. Als ersten Anhaltspunkt eignet sich, dass primitive Datentypen mit einem kleinen Buchstaben und Objektreferenztypen mit einem großen Buchstaben beginnen.
Es existieren folgende primitive Datentypen:

Schlüsselwort	Typ	Beschreibung	Wertebereich
`byte`	Ganzzahl	Vorzeichenbehaftet, 8 Bit	$-(2^7)$ bis 2^7-1
`short`	Ganzzahl	Vorzeichenbehaftet, 16 Bit	$-(2^{15})$ bis $2^{15}-1$
`int`	Ganzzahl	Vorzeichenbehaftet, 32 Bit	$-(2^{31})$ bis $2^{31}-1$
`long`	Ganzzahl	Vorzeichenbehaftet, 64 Bit	$-(2^{63})$ bis $2^{63}-1$
`float`	Fließkommazahl	einfache Genauigkeit	$1,4 \cdot 10^{-45}$ bis $\approx 3,4 \cdot 10^{38}$
`double`	Fließkommazahl	doppelte Genauigkeit	$4,9 \cdot 10^{324}$ bis $\approx 1,8 \cdot 10^{308}$
`char`	Charakter	Unicode-Code, 16 Bit	0 bis $2^{16}-1$
`boolean`	Wahrheitswert		`true/false`

Tabelle 4.1: Liste der primitiven Datentypen in Java

Hierbei fällt auf, dass es in Java keinen eingebauten Datentyp für vorzeichenfreie Zahlen („unsigned“) gibt. Dies kann bei der Verarbeitung von Binärdaten (beispielsweise bei Netzwerkkommunikation) zu Fehlern führen.

Objektreferenzen

Neben primitiven Datentypen gibt es noch die Objektreferenzen. Diese verhalten sich anders als primitive Datentypen, wobei uns vor allem die folgenden Unterschiede auffallen:

- Es gibt als einziges „echtes“ Literal den Wert `null`, der aussagt, dass die Objektreferenz kein Objekt referenziert. Jegliche Methodenaufrufe auf dieser Referenz brechen mit einer `NullPointerException` ab. Deshalb muss vor jedem Zugriff auf eine solche Referenz geprüft werden, ob sie ungleich `null` ist.
- Werden Objektreferenzen als Parameter übergeben, so wird hierbei ausschließlich die Referenz übergeben und auf das gleiche Objekt referenziert (Pass-by-Reference). Das bedeutet, eine Änderung an dem Objekt an einer Stelle kann sich an beliebig vielen anderen Stellen auswirken.

Wir werden uns Objektreferenzen im Abschnitt 4.6 über objektorientierte Programmierung in Java nochmals genauer anschauen.

Sonderfall `String`

Ein `String` ist eine Objektreferenz, kann allerdings in manchen Bereichen als ein primitiver Datentyp angesehen werden. Beispielsweise existieren, wir wir weiter unten noch sehen werden, Literale für diesen Datentyp, welche implizit ein Objekt erzeugen. Auch scheint ein String mit Pass-by-Value übergeben zu werden, da ein String nicht veränderbar ist.
Insgesamt existieren folgende Unterschiede:

- Ein String ist *immutable*, das heißt nicht veränderlich.
- Dadurch scheint es, als wird ein String Pass-by-Value übergeben.
- Bei Konstanten wird ein String als primitiver Typ angesehen und gleich behandelt.
- Es gibt eine syntaktische Form, String-Literale auszudrücken.
- Trotz allem ist es möglich, einem String den Wert `null` zuzuweisen. Dies ist gleichermaßen praktisch wie nervig.

4.1.2 Literale

In Java gibt es Schreibweisen für Literale für alle Datentypen, wobei die Erstellung von Objekten einen Sonderfall darstellt und nicht vollständig als Literal bezeichnet werden kann (es können zwar alle Argumente fest im Code stehen, das Objekt selbst wird allerdings erst zur Laufzeit erstellt).
In der folgenden Tabelle sind sämtliche syntaktische Methoden zu Definition von Literalen gelistet:

Datentyp	Schreibweise
`byte`	`123, -123`
`short`	`1234, -1234`
`int`	`12345, -12345`
`long`	`123456L, 123456L`
`float`	`12.34F, 0.34F/.34F`
`double`	`123.456, 0.456/.456`
`char`	`'a'`
`boolean`	`true, false`
`String`	`"Hello, World!"`
`Object`	`null`

- `String` ist hier kein primitiver Datentyp, das heißt mit einem String-Literal wird auch immer ein neues Objekt erzeugt.

- Das Literal `null` für `Object` ist allgemein anwendbar, wenn mit Objekten gearbeitet wird. Allerdings kann dies zu unerwarteten *`NullPointerExceptions`* führen, welche wir später noch eingehend betrachten werden.

Bei Literalen von Zahlen gibt es außerdem folgende Besonderheiten:

- Bei einem `float`-Literal muss ein „`F`“ am Ende des Literals angehängt werden, damit das Literal als `float` und nicht als `double` interpretiert wird. Die Groß-/Kleinschreibung ist irrelevant.

- Bei einem `long`-Literal kann ein „`L`“ am Ende des Literals angehängt werden, damit das Literal als `long` und nicht als `int` interpretiert wird. Die Groß-/Kleinschreibung ist irrelevant, aufgrund der Ähnlichkeit von „`l`“ und „`1`“ wir allerdings ein großes „`L`“ empfohlen.

- Bei allen Ganzzahlen (`byte`, `short`, `int`, `long`) können die Zahlen mit den Zahlensystemen Binär, Oktal, Dezimal und Hexadezimal eingegeben werden, wobei Dezimal sinnvollerweise der Standard ist. Zur Nutzung hiervon müssen den Werten bestimmte Zeichenketten vorangestellt werden. Dies sind `0b` für Binär, `0` für Oktal, nichts für Dezimal und `0x` für Hexadezimal.
  Das heißt, die folgenden Literale sind äquivalent:
  - `0b101010`
  - `052`
  - `42`
  - `0x2A`

  Wobei auch hier die Groß-/Kleinschreibung irrelevant ist, für den Prefix allerdings die Kleinschreibung und für die Zahl die Großschreibung empfohlen wird.

> **Warnung:** Wird bei Zahlen eine `0` vorangestellt, wird die Zahl Oktal interpretiert! Das heißt es gilt `010` $\neq$ `10`.

Escape-Sequenzen

Escape-Sequenzen werden innerhalb eines Strings mit einem Backslash (\) eingeleitet und bestehen in den meisten Fällen auf einem Zeichen.
In Java sind folgende Escape-Sequenzen verfügbar:

Escape-Sequenz	Repräsentiertes Zeichen
`\t`	Tab.
`\b`	Backspace.
`\n`	New line.
`\r`	Carriage Return.
`\f`	Formfeed.
`\'`	Single quote.
`\"`	Double quote.
`\\`	Backslash.

Tabelle 4.2: Java: Escape-Sequenzen

4.1.3 Schlüsselwörter

In Java existieren folgende Schlüsselwörter (kursiv geschriebene Themen werden wir nicht ausführlicher betrachten):

`abstract` Markiert eine...
- Klasse: das heißt, diese kann abstrakte Methoden enthalten.
- Methode: die von Unterklassen implementiert werden muss.

`continue` Fährt in einer Schleife mit dem nächsten Element fort.

`for` Leitet eine for-Schleife ein.

`new` Operator zur Erstellung eines neuen Objektes einer Klasse.

`switch` Leitet eine switch-Anweisung ein.

`assert` Legt bestimmte Bedingungen fest, die für Parameter gelten müssen. Gelten diese nicht, wird ein Fehler ausgelöst.

`default`
- Default-Fall in einer switch-Anweisung.
- Definition einer Default-Methode innerhalb eines Interfaces.
- *Definition des Default-Wertes einer Methode in einer Annotation*

`if` Leitet eine if-Verzweigung ein.

`package` Definition des Packages einer Klasse.

`synchronized` Markiert eine Methode oder einen Codeblock als synchron, das heißt es kann maximal ein Thread zur gleichen Zeit die Methode „betreten“.

`boolean` Datentyp.

`do` Leitet eine do-while-Schleife ein.

`goto` Reserviert. Löst ausschließlich einen Compilefehler aus.

`private` Markiert eine Klasse, einen Konstruktor, eine Methode oder ein Attribut als privat.

`this` Referenz auf die Instanz des aktuellen Objektes.

`break` Bricht die Ausführung einer Schleife ab.

`double` Datentyp.

`implements` Implementiert ein Interface.

`protected` Markiert eine Klasse, einen Konstruktor, eine Methode oder ein Attribut als protected.

`throw` Wirft eine Instanz einer Exception.

`byte` Datentyp.

`else` Leitet einen else-Block ein.

`import`		Importiert eine Klasse/Methode aus einem anderen Paket.
`public`		Markiert eine Klasse, einen Konstruktor, eine Methode oder ein Attribut als public.
`throws`		Definiert, dass ein Konstruktor/eine Methode eine bestimmte Exception werfen kann.
`case`		Leitet einen Fall eines switch-Ausdruckes ein.
`enum`		Leitet die Definition eines Enums ein.
`instanceof`		Operator zum Prüfen, ob eine Instanz eine Instanz einer anderen Klasse ist.
`return`		Gibt einen Wert zurück und bricht die Ausführung der Methode/des Konstruktors ab.
`transient`		*Definiert, dass ein bestimmte Attribut einer Instanz nicht mit serialisiert wird.*
`catch`		Leitet einen catch-Block ein.
`extends`	Klasse	Erweitert eine bestehende (möglicherweise abstrakte) Klasse.
	Interface	Erweitert ein bestehendes Interface.
`int`		Datentyp.
`short`		Datentyp.
`try`		Leitet einen try-Block ein.
`char`		Datentyp.
`final`	Klasse	Die Klasse ist nicht vererbbar.
	Methode	Die Methode ist nicht überschreibbar.
	Variable	Die Variable ist nur einmal zuweisbar.
`interface`		Leitet die Definition eines Interfaces ein.
`static`		Markiert eine innere Klasse, eine Methode oder ein Attribut als statisch.
`void`		„Datentyp“ als Platzhalter für „Nichts“.
`class`		Leitet die Definition einer Klasse ein.
`finally`		Leitet einen finally-Block ein.
`long`		Datentyp.
`strictfp`		*Legt fest, dass innerhalb einer Methode/einer Klasse nur strikte mathematische Operationen verwendet werden, sodass diese nicht optimiert werden sollen (es wird sich strikt an den Standard gehalten).*
`volatile`		*Markiert ein Attribut, sodass Modifikationen an diesem atomar durchgeführt werden und für andere Threads direkt sichtbar sind.*
`const`		Reserviert. Löst ausschließlich einen Compilefehler aus.

`float` Datentyp.

`native` *Markiert die Implementierung einer Methode als nativ, das heißt, die Implementierung liegt in nativem Code (C/C++) vor. Siehe JNI (Java Native Interface).*

`super` Referenz auf die Instanz der Oberklasse des aktuellen Objektes.

`while` Leitet eine while-Schleife ein.

Die genaue Bedeutung der obigen Schlüsselwörter werden wir in den jeweiligen Kapiteln genauer betrachten.

4.1.4 Bezeichner und Konventionen

In Java können Zeichenketten als Bezeichner gelten, wenn sie folgenden Bedingungen genügen:

- Sie bestehen nur aus `a` bis `z`, `0` bis `9`, `_` oder `$`.
- Sie beginnen nur mit `a` bis `z`, `_` oder `$`.

Zur Benennung sind außerdem folgende Konventionen zu empfehlen:

- Namen von Klassen beginnen mit einem Großbuchstaben.
- Namen von Methoden/Parametern/Variablen/etc. beginnen mit einem Kleinbuchstaben.
- Namen von Klassen sollen Subjekte und Objekte sein. Beispiel: „`User`“
- Namen von Methoden sollen mit einem Verb beginnen. Beispiel: „`generateAccessToken`“

> **Info:** Die oben Bedingungen, wann eine Zeichenkette als Bezeichner dienen kann, stellen Vereinfachungen dar. Streng genommen können alle Zeichen verwendet werden, für die die Methoden `Character.isJavaIdentifierStart(char)` bzw. `Character.isJavaIdentifierPart(char)` den Wert `true` ergeben. Damit *wären* auch Bezeichner wie „$\Delta\Psi$“ möglich.

4.1.5 Operatoren

In Java existieren die folgenden Operatoren, die genaue Bedeutung werden wir im Abschnitt 4.2.4 behandeln:

Kategorie	Ausprägungen
Arithmetische Verknüpfungen	`*`, `/`, `%`, `+`, `-`
Unäre Arithmetik	`expr++`, `expr--`, `++expr`, `--expr`, `+expr`, `-expr`
Logik	`!`, `&&`, `||`, `^`, `?:`
Bitweise Logik	`~`, `&`, `|`, `^`
Verschiebung (Shift)	`<<`, `>>`, `>>>`
Vergleiche	`<`, `>`, `<=`, `>=`, `==`, `!=`, `instanceof`
Zuweisungen	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`

Tabelle 4.3: Java: Operatoren

4.1.6 Strukturierung des Codes, Packages und Imports

Kommentare

Es gibt drei verschiedene Arten von Kommentaren in Java:

- Einzeilige Kommentare
  Der Kommentar ist nur in der aktuellen Zeile gültig.
- Blockkommentare
  Der Kommentar ist gültig, bis der Kommentar explizit beendet wird (auch über Zeilenumbrüche hinweg).
- Javadoc
  Dies ist eine besondere Form eines Blockkommentars, der so nur vor einer Methode, einem Feld oder einem Typ stehen kann (und, um es ganz genau zu nehmen, vor der `package`-Deklaration in einer Datei `package-info.java`). Diese besondere Form von Kommentaren wird genutzt, um den Code zu Dokumentieren und anschließend eine HTML-Dokumentation daraus zu generieren. Wir werden dies im Abschnitt **??** über Javadoc intensiver anschauen.

Whitespaces

Jegliche Whitespaces (Tab, Zeilenumbruch, Leerzeichen) sind in Java optional und dienen nur der Strukturierung des Codes. Trotz dass dies überflüssig ist, empfiehlt es sich, wenn wir unseren Code einrücken, gezielt Zeilenumbrüche setzen und so unseren Code lesbar machen.

Klammerung

In Java werden jegliche verfügbaren Klammern genutzt (`()`, , `[]`, `<>`). Sie haben die folgenden Zwecke:

`()` Klammerung von Ausdrücken (um die Operatorenpräzedenz festzulegen), Aufrufen von Methoden/Konstruktoren, Notwendig bei If-Ausdrücken, Switch-Case und Schleifen.

`{}` Kennzeichnung von Codeblöcken (Klassen-/Methodendefinition, If-Ausdrücke, Schleifen, Switch-Case, ...).

`[]` Zugriff auf die Elemente eines Arrays und Erstellung von Arrays.

<> Vergleichsoperatoren und Generics.

Packages und Imports

Als übergeordnete Strukturierung unserer Klassen existiert das Konstrukt von *Packages*, mit denen Klassen gruppiert und somit logisch zusammengefasst werden können. Beispielsweise können wir alle Klassen, die mit dem Zugriff auf eine Datenbank zu tun haben in ein Package `database` legen, und die Klassen, die mit dem User Interface zu tun haben in ein Package |ui|. Der Name eines Packages muss ein gültiger Bezeichner sein. Um eine Klasse in einem Package abzulegen, muss die Klasse zum einen mit der Zeile `package /*Packagename */;` starten und außerdem im korrekten Ordner liegen (das heißt eine Klasse im Package `ui` muss in einem Ordner `ui` liegen).

Damit entsteht eine logische Trennung und das Projekt wird übersichtlicher. Der *voll-qualifizierte Klassen* ist dann der Name der Klasse mit dem vorangestellten Package-Namen. Hierdurch werden Kollisionen in der Klassenbenennung vermieden. Beispiel: Unsere Klasse `Connection` liegt im Package `database`. Dann ist der voll-qualifizierte Name dieser Klasse `database.Connection`. Nutzen wir innerhalb einer Klasse eine andere Klasse, die nicht im selben Package und nicht im Package `java.lang` liegt, so müssen wir entweder den voll-qualifizierten Klassennamen bei jeder Verwendung angeben oder die Klasse mit dem Ausdruck `import /*Voll-qualifizierter Klassenname */;` importieren. Dann können wir die Klasse überall nutzen, als wäre sie im gleichen Package. Standardmäßig sind alle Klassen aus `java.lang` importiert.
Um Packages von Unterpackages zu trennen, können wir Punkte innerhalb des Package-Namen nutzen. Somit können wir beispielsweise Klassen, die mit der Datenbank und konkret mit MySQL zu tun haben, in ein Package `database.mysql` legen.

Konvention
Zur Vermeidung von Kollisionen ist es üblich, allen Packages innerhalb eines Projektes den umgekehrten Namen der Domain voranzustellen, für die das Projekt entwickelt wird (Bindestriche oder andere nicht-Java-konforme Zeichen werden dabei durch einen Unterstrich ersetzt, siehe §3.8 Java-Spezifikation). Danach folgt der Projektname.
Bei der Entwicklung von Nabla für das Fachgebiet Algorithmik am Fachbereich Informatik an der TU Darmstadt sollte also der Packagename `de.tu_darmstadt.informatik.algo.nabla` vorangestellt werden.

4.2 Anweisungen

Schauen wir uns nun an, wie man Dinge in Java tut, also wie wir Anweisungen und Ausdrücke formulieren können.

4.2.1 Variablen

Die allgemeine Syntax zur Deklaration einer Variablen ist:

```
<modifier> <typ> <name>;
```

Dabei ist <modifier> eine Reihe von Schlüsselwörtern, welche das Verhalten der Variablen modifizieren (genannt „Modifier“). Diese werden wir uns weiter unten genau anschauen. <typ> ist der Datentyp der Variablen (dies kann ein primitiver Datentyp aber auch ein Referenztyp sein). Der Name der Variablen wird mit <name> festgelegt.

Modifier

Für eine lokale Variable (das heißt eine Variable innerhalb eines Codeblocks oder als Parameter) existiert ausschließlich folgender Modifier:

`final` Sorgt dafür, dass die Variable nur einmal zugewiesen werden kann (zum Beispiel direkt nach oder noch während der Deklaration). Wenn möglich sollte eine Variable immer als `final` markiert werden, um versehentliches Überschreiben des Wertes zu verhindern.

Handelt es sich bei der Variablen um eine Instanz- oder Klassenvariable, sind zusätzlich folgende Modifier verfügbar:

`volatile` Bei der Zuweisung der Variablen geschieht die Zuweisung *atomar*. Dieser Modifier kann nicht mit `final` modifiziert werden.

`transient` Bei der Serialisierung einer Instanzvariablen wird dieses Feld nicht serialisiert.

- Sämtliche Sichtbarkeitsmodifizierer (siehe 4.6.1).

Alle Modifier können wir mit kleinen Einschränkungen beliebig kombinieren.
Beispiel: Eine Definition einer privaten Klassenvariable `timestamp`, die atomar Zugewiesen werden soll und nicht mit serialisiert werden soll sieht so aus:

```
private static transient volatile long timestamp;
```

Null- und Defaultwerte

Klassenvariablen, die nicht `final` sind, werden bestimmte Default-Werte zugewiesen (sofern die Variable nicht während der Deklaration direkt zugewiesen wird):

Typ	Default-Wert
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0F`
`double`	`0.0`
`boolean`	`false`
`char`	`'\000'` (Null-Byte)
`Object` und Unterklassen	`null`

Tabelle 4.4: Java: Defaultwerte

4.2.2 Zuweisungen

Um eine Variable zuzuweisen, wird folgender Ausdruck verwendet:

```
<variable> = <ausdruck>;
```

Dabei ist der linke Teil `<variable>` der Name der Variablen, welcher der Wert des Ausdrucks `<ausdruck>` zugewiesen wird. Der Ausdruck kann dabei beliebig komplex sein.
Wie können den Wert auch zeitgleich mit der Deklaration zuweisen, die Syntax ist dann wie folgt:

```
<modifier> <typ> <name> = <ausdruck>;
```

Eine Besonderheit ist hier, dass der Ausdruck einer normalen Zuweisung den Wert der Zuweisung zurück gibt (das heißt es gilt `(<variable> = <ausdruck>) == <ausdruck>`).

4.2.3 Methodenaufrufe

Der allgemeine Ausdruck, um eine Methode in Java aufzurufen ist:

```
<objekt>.<methodenname>([parameter], [parameter], ...)
```

Der Methodenname muss immer gegeben sein, ebenso wie das Objekt (beziehungsweise bei einer statischen Methode die Klasse), welches/welche das Objekt enthält. Die Parameter müssen gegeben sein, wenn die aufgerufene Methode dies fordert, es gibt aber auch Methoden, die keine Parameter erfordern.
Wir können die Rückgabe der Methode auch einer Variablen zuweisen, die Syntax ist dann wie folgt:

```
<variable> = <objekt>.<methodenname>([parameter], [parameter], ...)
```

Dies ist nur möglich, wenn die Methode einen Rückgabetyp hat, das heißt der Rückgabetyp nicht `void` ist.

4.2.4 Operatoren

Arithmetik-Operatoren

Es existieren die folgenden arithmetischen Operatoren, die allesamt alle primitiven und numerischen Datentypen (`byte`, `short`, `int`, `long`, `float`, `double`) annehmen:

Operator	Syntax	Beschreibung
`++`	`a++`, `++a`	a wird um 1 *inkrementiert*.
`--`	`a--`, `a--`	a wird um 1 *dekrementiert*.
`*`	`a * b`	a und b werden *multipliziert*.
`/`	`a / b`	a wird durch b *dividiert*.
`%`	`a % b`	Es wird a **mod** b berechnet (d.h. $a - \lfloor \frac{a}{b} \rfloor b$) (*Modulo*).
`+`	`a + b`	a und b werden *addiert*.
`-`	`a - b`	b wird von a *subtrahiert*.
`-`	`-a`	Negiert das Vorzeichen von a.

Bei den Inkrementierungs-/Dekrementierungs-Operatoren ist der Unterschied zwischen den Syntaxen `a++` und `++a` (beziehungsweise `a--` und `--a`), dass das Ergebnis von ersterem Ausdruck den Wert von a vor der Inkrementierung/Dekrementierung und `++a`/`--a` den Wert nach der Inkrementierung/Dekrementierung als Ergebnis liefert (Postfix vs. Prefix Operatoren). Das bedeutet, dass `a++ == a`, `a-- == a`, `++a == a + 1` und `--a == a - 1` gelten.

Kommazahlen und Division

Eine Division wird immer als *Ganzzahldivision* durchgeführt, wenn nicht mindestens einer der Parameter eine Fließkommazahl ist. Das bedeutet, dass Nachkommastellen nur berechnet werden, wenn mindestens einer der Parameter ein `float` oder `double` ist.
Eine Ganzzahldivision von a und b entspricht $\lfloor \frac{a}{b} \rfloor$, dass heißt, die Nachkommastellen werden abgeschnitten.

Logik- und Vergleichs-Operatoren

Es existieren die folgenden logischen Operatoren und Vergleichsoperatoren, die alle als Ergebnis ein `boolean` zurück geben.

Operator	Syntax	Parametertyp	Beschreibung
<	`a < b`	primitive Zahl	Ist a kleiner b?
>	`a > b`	primitive Zahl	Ist a größer b?
<=	`a <= b`	primitive Zahl	Ist a kleiner-gleich b?
>=	`a >= b`	primitive Zahl	Ist a größer-gleich b?
==	`a == b`	Beliebig	Ist a identisch zu b?
!=	`a != b`	Beliebig	Ist a nicht identisch zu b?
&&	`a && b`	Wahrheitswert	Verknüpft a und b mit einem logischem UND.
||	`a || b`	Wahrheitswert	Verknüpft a und b mit einem logischem ODER.
^	`a ^ b`	Wahrheitswert	Verknüpft a und b mit einem logischem XOR.
!	`!a`	Wahrheitswert	Negiert den Wahrheitswert von a

Identisch bedeutet für Zahlen, dass diese bis auf die letzte Nachkommastelle gleich sind. Für Objekte bedeutet dies, dass es ein und das selbe Objekt sind (das heißt, dass die Speicheradresse identisch ist). Eine Änderung auf a ändert somit auch b, wenn `a == b` gilt (nur bei Objekten!). Aufgrund dessen ist es auch nicht möglich, Strings mit == zu vergleichen, da dies bei Benutzereingaben oder ähnlichem immer `false` liefern würde, da die Objekte nur den gleichen Inhalt haben und nicht identisch sind (siehe auch 4.6.2).

Bitlogik-Operatoren

Die bitlogischen Operatoren können auf primitive Ganzzahlen (`byte`, `short`, `int`, `long`) angewendet werden. Diese wenden die üblichen logischen Verknüpfungen auf Bit-Ebene an, dass heißt, die Zahl wird in Binärdarstellung überführt und die Verknüpfung der Reihe nach auf jedes Bit einzeln angewendet (bei ungleich großen Datentypen werden die fehlenden Stellen bei dem kleineren mit Nullen aufgefüllt). Der Rückgabetyp entspricht immer dem größeren Datentyp. Es existieren die folgenden Operatoren:

Operator	Syntax	Beschreibung
<<	`a << b`	Verschiebt die Bits von a um b Stellen nach links.
>>	`a >> b`	Verschiebt die Bits von a um b Stellen nach rechts.
>>>	`a >>> b`	Verschiebt die Bits von a um b Stellen nach rechts und behält das Vorzeichen bei.
&	`a & b`	Verknüpft die Bits von a und b mit einer UND-Verknüpfung.
^	`a ^ b`	Verknüpft die Bits von a und b mit einer XOR-Verknüpfung.
|	`a | b`	Verknüpft die Bits von a und b mit einer ODER-Verknüpfung.
~	`~a`	Negiert die Bits von a.

Spezielle Operatoren

Zusätzlich zu den oben genannten Operatoren gibt es noch die Operatoren `new`, `instanceof` und der Ternäre Operator, die etwas anders funktionieren.

`new` Mit diesem Operator können neue Instanzen (Objekte) einer Klasse erstellt werden und die allgemeine Syntax lautet `new <klasse>([parameter], [parameter], ...);` diesen Operator werden wird im Abschnitt 4.6.1 genauer betrachten.

`instanceof` Mit diesem Operator kann geprüft werden, ob ein Objekt eine Instanz einer bestimmten Klasse darstellt, die allgemeine Syntax hierfür lautet `<objekt> instanceof <klasse>`. Beispielsweise wäre für eine Variable

`Number x = 1.2` der Ausdruck `x instanceof Double` wahr, der Ausdruck `x instanceof Integer` jedoch falsch.

Ternärer Operator Mit diesem Operator können, ähnlich wie bei einem If, Fallunterscheidungen vorgenommen werden. Die allgemeine Syntax lautet `<test> ? <wahr-fall> : <sonst-fall>`. Dabei wird zuerst der Test ausgewertet, ist dieser Wahr, so wird das Ergebnis von dem wahr-Fall zurück gegeben, sonst das Ergebnis von dem sonst-Fall. Dabei muss der Test zu einem Wahrheitswert auswerten und die beiden Fälle zu dem gleichen Typ, beziehungsweise einem kompatiblen Typ für den äußeren Ausdruck.

Bindungsstärke der Operatoren

Die Bindungsstärke der Operatoren in Java gliedert sich wie folgt, wobei die oberste Zeile die stärkste Bindungsstärke hat und mehrere Elemente auf einer Zeile die gleiche Bindungsstärke:

1. `expr++`, `expr-`
2. `++expr`, `-expr`, `+expr`, `-expr`, ∼, `!`
3. `*`, `/`, `%`
4. `+`, `-`
5. `<<`, `>>`, `>>>`
6. `<`, `>`, `<=`, `>=`, `instanceof`
7. `==`, `!=`
8. `&`
9. `^`
10. `|`
11. `&&`
12. `||`
13. `? :`
14. `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`

Klammerung

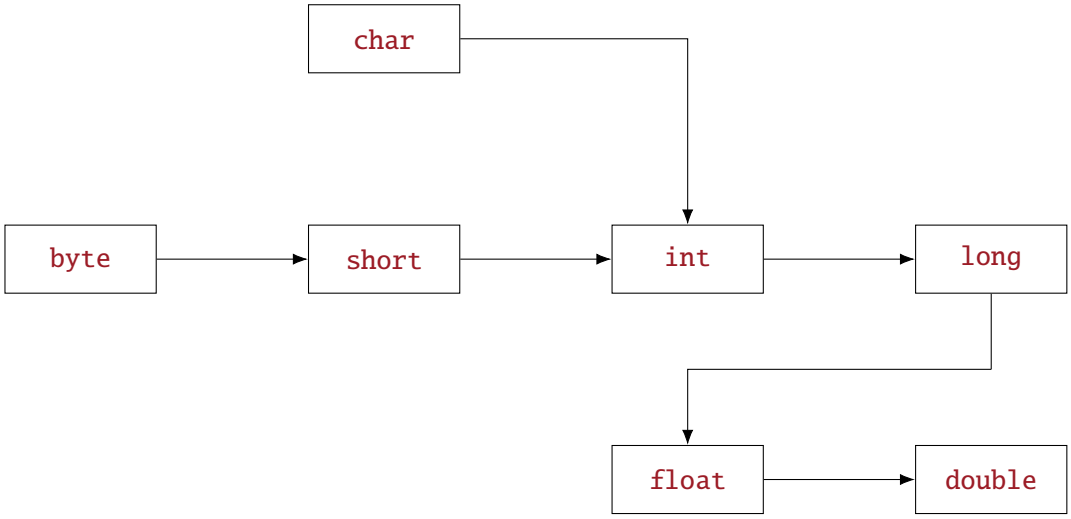
Um die Bindungsstärke von Operatoren zu beeinflussen, können Ausdrücke wie in der Mathematik geklammert werden, wobei die innerste Klammer immer zuerst ausgewertet wird. Hierfür dürfen ausschließlich runde Klammern (`(`, `)`) genutzt werden.

4.2.5 Implizite und Explizite Typenkonversion (Casts)

Schauen wir uns zuerst einmal an, was wir unter einer Typenkonversion verstehen: Wenn wir eine Variable `int a = 41` haben, können wir diese Problemlos einer anderen Variable mit dem Datentyp `long` zuweisen (`long b = a`). Hier liegt uns eine *implizite Typenkonversion* vor, bei der der Datentyp `int` zu einem `long` umgewandelt wird. Wir gehen nun getrennt auf primitive Typenkonversionen, Wrapper-Typen und Objektkonversionen ein.

Primitive Typen

Eine primitive Typenkonversion haben wir bereits gesehen. Eine implizite Typenkonversion ist immer dann möglich, wenn der neue Datentyp eine größere oder gleiche Datenmenge halten kann wie der alte Datentyp (das heißt es ist zum Beispiel nicht implizit möglich, eine Fließkommazahl in eine Ganzzahl zu konvertieren).



Der Pfeil $A \rightarrow B$ bedeutet, dass A implizit in B konvertiert werden kann. Der Rückweg ist ausgeschlossen. Außerdem ist die Konvertierung transitiv, dass bedeutet, wenn $A \rightarrow B$ und $B \rightarrow C$, dann geht auch $A \rightarrow C$.

Eine explizite Konvertierung wird vorgenommen, indem der neue Typ in Klammern vor die Variable (oder den Ausdruck) des alten Typs gesetzt wird:

```
(<neuer-typ>)<ausdruck>
```

Beispielsweise Wertet der Ausdruck `1/2.0` zu einem `double` aus und das Ergebnis muss explizit in ein `int` konvertiert werden: `(int)(1 / 2.0)`. Das Ergebnis wäre in diesem Falle `0`, da bei einer Typenkonvertierung von einer Fließkommazahl in eine Ganzzahl die Nachkommastellen abgeschnitten werden.

Wrappertypen

Wie wir im Abschnitt zu Generics (4.7.1) sehen werden, sind primitive Typen nicht immer hilfreich. Manchmal möchten wir auch Zahlen oder ähnliches in Objekten speichern können. Hier kommen die sogenannten *Wrappertypen* ins Spiel, die ebenso wir Strings immutable, das heißt nicht veränderlich, sind.

Wrappertypen sind Klassen, die eine primitive Variable speichern und diese bei Bedarf zur Verfügung stellt. Die Verwendung dieser Wrapper Typen erfolgt durch *Autoboxing* transparent, das heißt, eine Variable wird automatisch in einem Wrappertyp gespeichert und gelesen.

Die Namen der Wrappertypen entsprechen zu großen Teilen dem Namen des primitiven Typs mit einem großem Anfangsbuchstaben (die Klassen liegen allesamt in dem Package `java.lang`):

Primitiver Typ	Wrappertyp
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Autoboxing

Weisen wir einer Variable `Object obj` einen primitiven Wert (zum Beispiel `1.2`) zu, so wird dieser primitive Typ automatisch in den entsprechenden Wrappertyp konvertiert und der Variable zugewiesen. Ebenfalls wird an Stellen, an denen primitive Typen gebraucht werden (zum Beispiel in arithmetischen Operationen oder Vergleichen) der Wrappertyp zurück in einen primitiven Wert gewandelt.

Beispiel:

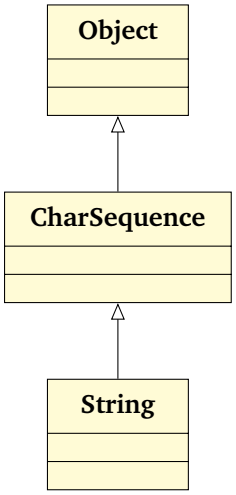
```
1 double primitive = 1.2;
2 int wholeNumber = (int) x;
3 Double wrapper = primitive;   // Autoboxing.
4 if (wrapper > wholeNumber) {  // Autounboxing.
5     ...
6 }
```

Warnung: Im Gegensatz zu primitiven Typen können Variablen von Wrappertypen `null` sein. Wird versucht, Autounboxing auf `null`-Werten anzuwenden, so wird eine `NullPointerException` geworfen.

Objekte („Downcast“)

Auch bei Objekten müssen wir manchmal eine Typenkonvertierung vornehmen. Implizite Typenkonvertierungen sind hier genau dann möglich, wenn der neue Typ eine Oberklasse des alten Typs ist. Eine explizite Typenkonvertierung wird benötigt, wenn in der Klassenhierarchie „nach unten“ gegangen werden soll (dies wird *Downcast* genannt). Eine explizite Typenkonvertierung findet wie bei primitiven Typen statt indem der neue Typ in Klammern vor den Ausdruck geschrieben wird.

Schauen wir uns dies am Beispiel eines Strings an:



Eine implizite Typenkonvertierung ist nun immer nach oben in der Hierarchie möglich (also `String` → `CharSequence` → `Object`).
Beispiel:

```
String s = "Hello, World!";
Object o = s;                 // Implizite Typenkonvertierung.
String casted = (String) o;   // Explizite Typenkonvertierung (Downcast).
```

4.3 Kontrollstrukturen

4.3.1 Verzweigungen

In Java gibt es als grundlegende Art der Verzweigung nur das einfache If. Ein Switch, welches wir uns später anschauen werden, baut sehr direkt auf einem If auf reduziert größtenteils die Tipparbeit.

If

In `if` hat in Java die folgende Form:

```
if (/* Test */) {
    /* then-Fall */
} else if (/* else-Test */) {
    /* else-then-Fall */
} else {
    /* else-Fall */
}
```

Abbildung 4.1: Java: `if`-Verzweigung

Dabei kann es beliebig viele Else-Ifs geben, oder diese können ganz weg gelassen werden. Ebenfalls kann der Else-Fall weggelassen werden, einzig und allein der Then-Fall ist nötig (dieser kann theoretisch auch leer sein, dies ergibt aber in den meisten Fällen keinen Sinn).
Somit ist die einfache Form des `if`s:

```java
if (/* Test */) {
    /* then-Fall */
}
```

Abbildung 4.2: Java: Einfache `if`-Verzweigung

Alle Tests (die Bedingungen für das If) *müssen* zu einem `boolean` auswerten. Alle anderen Datentypen werden nicht akzeptiert und der Code wird nicht kompilieren.

Switch

Ein `switch` stellt eine Vereinfachung von vielen If-Else-Verzweigungen dar, welche alle die gleiche Operation (beispielsweise das Vergleichen von zwei Objekten) ausführen.

Schauen wir uns als Motivation den folgenden Code an, welcher ausgibt, wie viele Primzahlen `p` mit `p <= x <= 10` existieren.

```java
if (x == 1) {
    prime = 0;
} else if (x == 2) {
    prime = 1;
} else if (x == 3) {
    prime = 2;
} else if (x == 4) {
    prime = 2;
} else if (x == 5) {
    prime = 3;
} else if (x == 6) {
    prime = 3;
} else if (x == 7) {
    prime = 4;
} else if (x == 8) {
    prime = 4;
} else if (x == 9) {
    prime = 4;
} else if (x == 10) {
    prime = 4;
} else {
    // Error.
}
```

Abbildung 4.3: Java: `switch` Motivation

Mit Hilfe eines Switches können wir den Code nun äquivalent umformen:

```java
switch (x) {
case 1:
    prime = 0;
    break;
case 2:
    prime = 1;
    break;
case 3:
    prime = 2;
    break;
case 4:
    prime = 2;
    break;
case 5:
    prime = 3;
    break;
case 6:
    prime = 3;
    break;
case 7:
    prime = 4;
    break;
case 8:
    prime = 4;
    break;
case 9:
    prime = 4;
    break;
case 10:
    prime = 4;
    break;
default:
    // Error.
    break;
}
```

Abbildung 4.4: Java: `switch`

Der Code ist äquivalent zu der If-Else-Kaskade und ist einfacher zu verstehen. Allerdings müssen wir in jedem Case (ein einzelner Fall in dem Switch-Konstrukt) ein `break` platzieren, welches die Ausführung des Switches abbricht. Wird das `break` weggelassen, so *fällt die Ausführung durch*, das heißt es wird einfach mit dem nächsten Case fortgefahren.
Um den Nutzen hiervon verstehen zu können müssen wir uns darüber im klaren sein, wie ein Switch ausgewertet wird:

- Im ersten Schritt wird die *Vergleichsvariable* in den Klammern hinter dem Schlüsselwort `switch` ausgewertet. Diese Variable darf nur zu einem String, einem primitiven Wert oder dem Wert eines Enums auswerten.
- Anschließend wird der erhaltene Wert mit dem Wert eines jeden Cases verglichen. Hierfür ist es nötig, das hinter dem Schlüsselwort `case` ausschließlich Literale oder Konstanten mit dem

gleichen Typ stehen. Es kann niemals zwei Cases mit dem gleichen Wert (auch genannt *Label*) geben!

- Nun wird zur ersten Zeile des Cases gesprungen, dessen Wert gleich dem Vergleichswert ist. Existiert kein solcher Case, so wird zu dem Default-Case gesprungen, welcher aber nicht existieren muss. Wird kein Code zur Ausführung gefunden, so wird das gesamte Switch übersprungen.
- Sämtlicher nachfolgender Code wird nun ausgeführt, bis ein `break` gefunden wird. Dann wird aus dem Switch gesprungen un nach dem Switch fortgefahren. Außerdem beenden Returns und Exceptions wie üblich die Ausführung.

Das heißt: Beim Start des Cases wird zu einer Stelle im Code gesprungen und dieser so lange ausgeführt, bis die Ausführung *explizit* beendet wird oder kein Code mehr im Switch existiert.
Mit dieser Kenntnis kann obiger Code des Switches deutlich vereinfacht werden, indem wir einfach bei jeder Primzahl den Zähler um eins erhöhen:

```
1  prime = 0;
2  switch (x) {
3      case 10:
4      case 9:
5      case 8:
6      case 7:
7          prime++;
8      case 6:
9      case 5:
10         prime++;
11     case 4:
12     case 3:
13         prime++;
14     case 2:
15         prime++;
16     case 1:
17         break;
18     default:
19         // Error.
20         break;
21 }
```

Abbildung 4.5: Java: `switch` mit Fall-Thru

> **Warnung:** Ein Case in einem Switch öffnet keinen neuen Scope! Somit können Variablen nur einmal genutzt werden oder es muss ein Block um den Case geschrieben werden.

4.3.2 Schleifen

While-Schleife

In Java sieht die zuvor vorgestellte While-Schleife wie folgt aus:

```java
while (/* Test */) {
    /* Code */
}
```

Abbildung 4.6: Java: `while`-Schleife

Wie auch schon beim If gesehen, darf auch hier der Test nur zu einem `boolean` und nicht zu anderen Datentypen ausgewertet werden. Der Test wird *vor* jedem Schleifendurchlauf ausgeführt und bricht ab, sobald er zu `false` auswertet.

Do-While-Schleife

Als Spezialfall einer While-Schleife gibt es in Java die Do-While-Schleife, welche immer *mindestens einmal* ausgeführt wird:

```java
do {
    /* Code */
} while (/* Test */);
```

Abbildung 4.7: Java: `do-while` Schleife

Wie bei allen Schleifen und Bedingungen darf auch hier der Test nur zu einem `boolean` auswerten. Im Gegensatz zur While-Schleife wird der Test hier allerdings *nach* jedem Schleifendurchlauf ausgeführt, wodurch die Schleife immer mindestens einmal ausgeführt wird.

For-Schleife

Da oftmals über Elemente einer Liste oder eines Arrays iteriert wird und der Code hierfür immer gleich ist (eine Zählvariable wird in jedem Schritt hochgezählt):

```java
int i = 0;
while (i < array.length) {
    /* Code */

    i++;
}
```

Abbildung 4.8: Java: `for each`-Schleife Motivation

kann dieser Code zu folgendem, äquivalentem, Code umgewandelt werden:

```java
for (int i = 0; i < array.length; i++) {
    /* Code */
}
```

Abbildung 4.9: Java: `for each`-Schleife Motivation

womit Code eingespart wird und die Iteration deutlich übersichtlicher ist.

Die einzelnen Bestandteile des Schleifenkopfes, welche mit Semikola getrennt werden müssen, haben folgende Namen und Funktionen:

`int i = 0` *Initialisierung* - Der Code an dieser Stelle wird *einmalig vor* Durchlauf der Schleife ausgeführt.

`i < array.length` *Test* - Ein zu `boolean` auswertender Ausdruck, welcher *vor jedem* Durchlauf ausgewertet wird. Wird die Bedingung zu `false` ausgewertet, wird die Schleife beendet.

`i++` *Schritt* - Der Code an dieser Stelle wird `nach jedem` Durchlauf ausgeführt.

Ferner können alle Bestandteile der For-Schleife ausgelassen werden (unter Beibehaltung der Semikola!), wobei Initialisierung und Schritt einfach nicht ausgeführt werden und die Bedingung immer zu `true` auswertet. Somit sind „`while (true) { }`“ und „`for (;;) { }`“ äquivalent.

„Erweiterte“ For-Schleife

Statt wie üblich über Arrays und Listen zu iterieren:

```
1 for (int i = 0; i < array.length; i++) {
2     Object element = array[i];
3
4     /* Code */
5 }
```

Abbildung 4.10: Java: Erweiterte For-Schleife Motivation

kann seit Java 5 auch die *erweiterte For-Schleife* verwendet werden, um über Arrays oder Instanzen von `java.util.Iterable` iterieren (alle Standard-Klassen für Listen implementieren dieses Interface):

```
1 for (Object element : array) {
2     /* Code */
3 }
```

Abbildung 4.11: Java: Erweiterte For-Schleife

Dabei wird immer über den konkreten Typ, der im Array (oder der Liste) gespeichert ist, iteriert. Anders ausgedrückt: Der Code im Schleifenkörper wird für jedes Element des Arrays oder der Liste ausgeführt.

`break`, `continue`

Um eine Schleifenausführung vorzeitig auszuführen, gibt es die folgenden Schlüsselwörter:

`break` Bricht die gesamte Schleifenausführung der innerstmöglichen Schleife ab.

`continue` Fährt mit der nächsten Iteration der innerstmöglichen Schleife fort.

Beispiel

Schauen wir uns abschließend folgendes schwachsinniges Beispiel an, um die Funktionalität von `break` und `continue` zu verdeutlichen. Der folgende Code summiert die ungeraden Elemente eines Arrays, wobei keine weiteren Element aufsummiert werden, sobald die Summe einmal `10` überschritten hat.

```
1  int sumOdd = 0;
2  for (int x : array) {
3      if (x % 2 == 0) {
4          // Element is even --> continue with next element.
5          continue;
6      }
7
8      if (sumOdd > 10) {
9          // Sum is over 10 --> stop loop.
10         break;
11     }
12
13     sumOdd += x;
14 }
15 System.out.println(sumOdd);
```

Abbildung 4.12: Java: `break`, `continue` Beispiel

Mit den Werten `array = new int[] { 1, 2, 3, 4, 5, 7, 8, 9 }` wird `16` ausgegeben, wobei der Code wie folgt ausgeführt wird:

```
(Zeile =  1; sumOdd =  0)

(Zeile =  2; sumOdd =  0; x = 1)
(Zeile =  3; sumOdd =  0; x = 1; (x % 2 == 0) = false)
(Zeile =  8; sumOdd =  0; x = 1; (sumOdd > 10) = false)
(Zeile = 13; sumOdd =  1; x = 1)

(Zeile =  2; sumOdd =  1; x = 2)
(Zeile =  3; sumOdd =  1; x = 2; (x % 2 == 0) = true)

(Zeile =  2; sumOdd =  1; x = 3)
(Zeile =  3; sumOdd =  1; x = 3; (x % 2 == 0) = false)
(Zeile =  8; sumOdd =  1; x = 3; (sumOdd > 10) = false)
(Zeile = 13; sumOdd =  4; x = 3)

(Zeile =  2; sumOdd =  4; x = 4)
(Zeile =  3; sumOdd =  4; x = 4; (x % 2 == 0) = true)

(Zeile =  2; sumOdd =  4; x = 5)
(Zeile =  3; sumOdd =  4; x = 5; (x % 2 == 0) = false)
(Zeile =  8; sumOdd =  4; x = 5; (sumOdd > 10) = false)
(Zeile = 13; sumOdd =  9; x = 5)

(Zeile =  2; sumOdd =  9; x = 6)
(Zeile =  3; sumOdd =  9; x = 6; (x % 2 == 0) = true)

(Zeile =  2; sumOdd =  9; x = 7)
(Zeile =  3; sumOdd =  9; x = 7; (x % 2 == 0) = false)
(Zeile =  8; sumOdd =  9; x = 7; (sumOdd > 10) = false)
(Zeile = 13; sumOdd = 16; x = 7)

(Zeile =  2; sumOdd = 16; x = 8)
(Zeile =  3; sumOdd = 16; x = 8; (x % 2 == 0) = true)

(Zeile =  2; sumOdd = 16; x = 9)
(Zeile =  3; sumOdd = 16; x = 9; (x % 2 == 0) = false)
(Zeile =  8; sumOdd = 16; x = 9; (sumOdd > 10) = true)

(Zeile = 15; sumOdd = 16)
```

Abbildung 4.13: Java: `break`, `continue` Beispielausführung

4.4 Methoden

In Java sind Methoden immer an ein Objekt oder eine Klasse gebunden. Die Unterschiede hierzwischen werden wir uns später im Abschnitt 4.6 zu objektorientierter Programmierung in Java anschauen. In diesem Kapitel werden wir annehmen, dass sich alle Methoden in einer Klasse befinden, eine Instanz der Klasse vorliegt und die Methoden an diese Instanz gebunden sind.
Betrachten wir zur Einführung die folgende Methode:

```java
int add(int a, int b) {
    return a + b;
}
```

die die Summe der Zahlen `a` und `b` berechnet.
Dabei entspricht *int add(int a, int b)* dem Kopf der Methode und alles in den geschweiften Klammern (also `return a + b;`) dem Körper der Methode.

Der Methodenkopf

Ein Methodenkopf hat folgenden allgemeinen Aufbau:

`[MODIFIER] [GENERICS] <RÜCKGABETYP> <METHODENNAME>([PARAMETER])`

dabei ist die Angabe von Modifizierern (*Modifier*), Generics und Parametern optional, wobei beliebig viele Parameter angegeben werden können. Die Klammern hinter dem Methodennamen müssen dennoch vorhanden sein, auch wenn keine Parameter angegeben werden.

Modifizierer

Die Modifizierer, die an einer Methode angegeben werden können, werden wir uns im Kapitel über objektorientierte Programmierung genauer anschauen, da die in Java vorhanden Modifizierer nur in diesem Kontext Sinn ergeben.
Erweitertes Wissen: Eine Ausnahme stellt der Modifizierer `strictfp` dar, der der JVM aufträgt, arithmetische Operationen exakt wie in der Spezifikation der JVM vorzunehmen und nicht zu optimieren.

Generics

Siehe 4.7.1.

Rückgabetyp

Hiermit geben wir den Typ an, den das Ergebnis unserer Methode hat. Dies kann ein primitiver Datentyp oder eine Klasse sein. Wird nichts zurückgegeben, muss `void` angegeben werden, was so viel wie „nichts“ heißt.

Methodenname

Dies ist der Name der Methode, mit dem wir selbige referenzieren können. Der Name muss sich an die Grundregeln von Bezeichnern in Java halten (siehe 4.1.4).

Parameter

Die ist eine Komma-separierte Liste von Parametern, die unsere Funktion erwartet.
Einer dieser Parameter ist dabei aufgebaut wir eine normale Variablendeklaration, das heißt `[final] <DATENTYP> <NAME>`. Ein hier verwendeter Name kann im Körper der Methode nicht erneut für Variablen genutzt werden, der Zugriff auf den Wert des Parameters erfolgt, als wäre dieser eine ganz normale Variable. Ein in der Parameterliste angegebenes `final` verhält sich entsprechend.
Varargs: Varargs sind eine spezielle Form der Parameter, die dem Aufrufer erlauben, beliebig viele Parameter zu übergeben.
Betrachten wir hierzu folgendes Beispiel, um beliebig viele Zahlen zu addieren:

```java
int add(int[] numbers) {
    int result = 0;
    for (int x : numbers) {
        result += x;
    }
    return result;
}
```

Ein Aufrufer müsste die Funktion zum Beispiel so aufrufen: `add(new int[] { 1, 2, 3, 4, 5 })`, das heißt der müsste erst ein Array erstellen und dieses der Funktion übergeben. Dies stellt einen erheblichen Schreibaufwand dar.
Hätten wir stattdessen unsere Funktion wie folgt mit Varargs gestaltet, vereinfacht sich der Aufruf, wie wir gleich sehen werden:

```java
int add(int... numbers) {
    int result = 0;
    for (int x : numbers) {
        result += x;
    }
    return result;
}
```

Nun vereinfacht sich der funktional identische Aufruf zu `add(1, 2, 3, 4, 5)`.
Wir sehen auch, dass sich an dem Körper unserer Funktion nichts geändert hat, einzig und allein die manuelle Erstellung des Arrays verschwindet. Konkret heißt dies, dass Java uns die Arbeit abnimmt, das Array manuell zu erstellen, sondern dies im Hintergrund erledigt. Wenn der Aufrufer unbedingt will, kann er dennoch ein einfaches Array übergeben.

> **Warnung:** Es ist nicht möglich, nach einem Vararg-Parameter noch weitere Parameter anzugeben, da Java sonst nicht wüsste, welche Parameter noch zum Varargs gehören und welche nicht. Vor einem Vararg-Parameter ist dies problemlos möglich.

Signatur

Die Signatur einer Methode muss innerhalb einer Klasse eindeutig sein. Zu der Signatur einer Methoden gehören

- der Methodenname und
- die Typen der Parameter.

Somit sind bei den folgenden Methoden:

1. `int add(int a, int b)`
2. `float add(int a, int b)`
3. `float add(float a, float b)`

die Methoden 1 und 2 der Signatur nach identisch, die 3. Methode hingegen verschieden. Somit dürften in einer Klasse nur folgende Kombinationen vorkommen:

$$\emptyset, \quad \{1\}, \quad \{1,3\}, \quad \{2\}, \quad \{2,3\}$$

Formale Parameter vs. Aktualparameter

Damit es bei Unterhaltungen über Methoden nicht zu Verwirrungen kommt, schauen wir uns noch die Begriffe *Formale Parameter* und *Aktualparameter* an:

Formale Parameter Formale Parameter sind jene, welche bei der Definition einer Methode angegeben werden.

Aktualparameter Aktualparameter sind die Parameter, welche einer Methode bei einem Aufruf übergeben werden.

Beispiel:

```
1 void doIt(int a, int b) { /* ... */ }
2
3 int g = 9.81;
4 doIt(1, g)
```

In diesem Beispiel sind `a` und `b` die formalen Parameter und `1` und `g` die Aktualparameter.

Veträge in Form von Javadoc

Verträge sind Teil der Dokumentation, siehe **??**.

Überladen

Wie wir bereits im Abschnitt über Signaturen gesehen haben, muss ausschließlich die Signatur einer Methode in einer Klasse eindeutig sein, nicht der Name der Methode.
Wird ein Methodenname mehrmals in einer Klasse verwendet, so wird diese Methode *überladen*. Dies ist sinnvoll, wenn eine Methode beispielsweise unterschiedliche Parametertypen annehmen kann, um mit ihnen zu arbeiten, ein jeweils eigener Methodenname aber keinen Sinn ergibt.
Beispiel:
Wir haben eine Klasse mit folgenden Methoden:

- `String valueOf(int a)`
- `String valueOf(float a)`

dann ist die Methode `valueOf(...)` *überladen*. Bei einem Aufruf `valueOf(42)` wird die erste, bei einem Aufruf `valueOf(4.2F)` die zweite Methode verwendet.

Problematiken

Manchmal kann der Compiler nicht entscheiden, welche der überladenen Methoden aufgerufen werden soll. In diesem Fall muss der Typ eines Parameters genauer spezifiziert werden, damit der Code kompiliert.
Beispiel:
Wir haben eine Klasse mit folgenden Methoden:

- `String valueOf(Integer a)`
- `String valueOf(Long a)`

Bei einem Aufruf `valueOf(null)` könnten beide Methoden aufgerufen werden, der Typ passt bei beiden. Somit kann der Compiler den Code so nicht kompilieren und bricht ab. Zur Lösung müssen wir den Typ genauer spezifizieren, zum Beispiel durch einen Downcast: `valueOf((Integer)null)`. Nun wird die erwartete Methode aufgerufen.

4.4.1 Rückgabe von Werten

Wie wir bereits gesehen haben, wird bei der Definition einer Methode ebenfalls ein Rückgabetyp definiert. Definieren wir einen Rückgabetyp, so müssen wir auch einen entsprechenden Wert zurück gegen. Dies wird mit dem Schlüsselwort `return` vollführt. Die allgemeine Syntax ist hierbei `return [AUSDRUCK];`, wobei der Ausdruck durch einen solchen ersetzt werden muss, der zu dem Rückgabetyp auswertet. Der Ausdruck `return;` kann auch verwendet werden, wenn der Rückgabetyp `void` ist. Hiermit kann die Besonderheit ausgenutzt werden, dass eine Methode sofort zurück geht, sobald ein Return ausgeführt wurde. Damit ist es zum Beispiel möglich, am Anfang der Methode einige Einschränkungen der Parameter zu prüfen und erst fortzufahren, sobald die Bedingungen erfüllt sind:

```
1  void doIt(Integer a, Integer b) {
2      if (a == null || b == null) {
3      return;  // Vorzeitiges Return.
4      }
5      if (a < 0 || b < 0) {
6          return;  // Vorzeitiges Return.
7      }
8
9      ...
10 }
```

Sonderfall `finally`
Wird ein Return innerhalb eines Try-Finally verwendet, so wird nach der Ausführung des Returns zuerst noch der `finally`-Block ausgeführt und erst danach zurück zum Aufrufer gesprungen. Befindet sich innerhalb des Finallys wiederum ein Return, so wird der Wert des letzten Returns zurück gegeben.
Beispiel:

```
1  int add(int a, int b) {
2      try {
3          return a + b;
4      } finally {
5          return a - b;
6      }
7  }
```

Wird die Methode mit `add(2, 1)` aufgerufen, so werden die Zeilen in der Reihenfolge (3, 5) ausgeführt und der Wert 1 zurück gegeben.

4.5 Scoping

In Java wird immer dann ein neuer Scope geöffnet, wenn eine geschweifte Klammer auf geht und ein Scope geschlossen, wenn die geschweifte Klammer zu geht. Das ist zum Beispiel bei Methoden und Schleifen der Fall.
Das bedeutet: Eine Variable, die wir innerhalb einer Methode definieren (inklusive der Parameter) ist von außerhalb nicht zugreifbar. Das gleiche gilt für Variablen, die innerhalb eines If-Blocks oder dem Körper einer Schleife definiert wurden.
Beispiel:

```
1  int multiply(int a, int b) {
2      int result = 0;
3
4      int bAbs = Math.abs(b);
5      for (int i = 0; i < Math.abs(b); i++) {
6          result += a;
7      }
8
9      if (b < 0) {
10         return -result;
11     } else {
12         return result;
13     }
14 }
```

Auf die Variablen `a`, `b` und `result` kann von außerhalb der Methode nicht zugegriffen werden. Ebenfalls kann nicht auf `i` zugegriffen werden, dies ist aber schon außerhalb der Schleife (also in Zeile 1 bis 4 und 8 bis 14) nicht möglich.
Der Wert der Variable `result` wird zurück gegeben, womit der Aufrufer Zugriff auf den Wert der Variable hat, aber ohne Kenntnis darüber, wie das Ergebnis zustande gekommen ist.

4.6 Klassen und objektorientierte Programmierung

4.6.1 Klassen, Objekte und Methoden

Eine Klasse hat folgende allgemeine Syntax:

```
1 <SICHTBARKEIT> [final] class <NAME> {
2     <KLASSENKOERPER>
3 }
```

Die Sichtbarkeit kann dabei Public, Private, Protected oder Default sein. Was genau diese Sichtbarkeiten bedeuten, werden wir im Abschnitt 4.6.1 genauer betrachten. Der Name der Klasse muss ein gültiger Bezeichner sein und beginnt üblicherweise mit einem Großbuchstaben. Außerdem kann eine Klasse als `final` markiert werden, wodurch keine Unterklassen mehr gebildet werden können.
Der Körper der Klasse enthält den gesamten Klasseninhalt, das heißt Methoden, Attribute, Konstruktoren, Initialisierer, ...
Dabei werden die Variablendeklarationen sowie die Methodendefinitionen einfach nacheinander in die Klasse geschrieben und es kann von jeder Instanzmethode auf jede Variable zugegriffen werden.

Beispiel:

```
1  import java.util.Date;
2
3  public class Human {
4      private String hairColor;  // Instanzvariable.
5      private int birthYear;     // Instanzvariable.
6
7      // Konstruktor.
8      public Human(String hairColor, int birthYear) {
9          this.hairColor = hairColor;
10         this.birthYear = birthYear;
11     }
12
13     // Methode.
14     public int calculateAge() {
15         return birthYear - new Date().getYear() - 1900;
16     }
17
18     // Methode.
19     public String getHairColor() {
20         return hairColor;
21     }
22
23     // Methode.
24     public int getBirthYear() {
25         return birthYear;
26     }
27 }
```

Die Definition einer Variablen innerhalb einer Klasse hat die gleiche Syntax wie die Definition einer lokalen Variable (siehe 4.2.1) mit dem Zusatz, dass ein Sichbarkeitsmodifizierer (Public, Protected, Private oder Default) gesetzt werden kann. Außerdem kann ein solches „Feld“ als *statisch* markiert werden, näheres werden wir im Abschnitt 4.6.1 betrachten.
Eine Methoden wird ebenfalls wie in 4.4 beschrieben definiert, kann ebenfalls einen Sichbarkeitsmodifizierer aufweisen und als *statisch* markiert werden.

Statische Methoden und statische Attribute

Um ein Attribut oder eine Methode als *statisch* zu markieren, wird ihr der Modifizierer `static` vorangestellt. Dadurch wird das Feld/die Methode zu einem statischen Feld/einer statischen Methode und kann auch ohne eine Instanz der Klasse genutzt werden.
Außerdem ist ein statisches Feld nicht Instanzabhängig, das bedeutet, alle Instanzen haben den selben Wert, wenn sie auf ein statisches Feld zugreifen.

Einschränkungen

Da eine statische Methode nicht von einer Instanz abhängig ist, kann eine statische Methode auch nicht auf Instanzfelder (also nicht-statische Felder) zugreifen. Hierzu muss immer eine Instanz verfügbar sein.

Sichtbarkeit

Klassen sowie alle Attribute und alle Methoden (im Folgenden „Elemente“) können folgende Sichtbarkeitsmodifizierer enthalten:

`public` Das Element ist für jede Klasse und jedes Package sichtbar.

`private` Das Element ist nur für die eigene Klasse sichtbar (bei Klassen ist dies nur möglich, wenn die Klasse geschachtelt ist, siehe 4.6.7).

`protected` Das Element ist nur für Klassen aus demselben Package und jede Unterklasse sichtbar.

Default Das Element ist nur für die Klassen aus demselben Package sichtbar. Dieser Modifizierer wird angewandt, indem der Modifizierer weggelassen wird.

„Sichtbar“ bedeutet in diesem Falle, dass das Attribut lesbar und, sofern es nicht `final` ist, auch lesbar ist.

Abgrenzung: Objektvariable ⟷ Objektkonstante ⟷ Klassenvariable ⟷ Klassenkonstante

Objektvariable Eine Instanzabhängige, nicht-statische Variable, die nicht mit `final` markiert ist. Eine Objektvariable wird mit dem Default-Wert des jeweiligen Datentyps belegt bis sie zugewiesen wird.

Objektkonstante Eine Instanzabhängige, nicht-statische Variable, die mit `final` markiert ist. Eine Objektkonstante muss direkt bei der Deklaration, spätestens aber bei der Ausführung des Konstruktors initialisiert werden.

Klassenvariable Eine Instanzunabhängige, statische Variable, die nicht mit `final` markiert ist. Eine Klassenvariable wird mit dem Default-Wert des jeweiligen Datentyps belegt bis sie zugewiesen wird.

Klassenkonstante Eine Instanzunabhängige, statische Variable, die mit `final` markiert ist. Eine Klassenkonstante muss direkt bei der Deklaration, spätestens aber bei der Ausführung des Static-Initializers initialisiert werden.

Name	`static`	`final`
Objektvariable	☐	☐
Objektkonstante	☐	☒
Klassenvariable	☒	☐
Klassenkonstante	☒	☒

Abgrenzung: Objektmethode ⟷ Klassenmethode

Objektmethode Eine Instanzabhängige, nicht-statische Methode, die auf Instanzvariablen und -konstanten zugreifen kann.

Klassenmethode Eine Instanzunabhängige, statische Methode, die nicht auf Instanzvariablen und -konstanten zugreife kann.

Name	`static`
Objektmethode	☐
Klassenmethode	☒

Konstruktoren

Eine Konstruktor wird aufgerufen, wenn eine Instanz einer Klasse mit `new` erstellt wird (zum Beispiel `new Human("red", 1997)`). Ein Konstruktor muss in jeder Klasse vorhanden sein und ist eine spezielle Methode, die keinen Rückgabetyp aufweist und Objektkonstanten initialisieren kann/muss. Ein Konstruktor hat die folgende allgemeine Syntax:

```
1 <SICHTBARKEIT> <KLASSENNAME>([PARAMETER]) {
2     ...
3 }
```

Die Sichtbarkeit ist dabei wie oben, jedoch müssen wir beachtet, dass eine Sichtbarkeit von `private` dazu führt, dass keine Unterklasse der Klasse mehr erstellt werden kann, da mit jedem Konstruktoraufruf auch der Konstruktor der Oberklasse aufgerufen wird. Wird kein expliziter Konstruktor angegeben, dann wird eine Standard-Konstruktor erstellt, welcher keine Parameter annimmt und keine weiteren Initialisierungsschritte für die Klasse vornimmt. Geben wir explizit einen Konstruktor an, so wird kein Standard-Konstruktor erstellt und wir müssten einen solchen Konstruktor selber erstellen, wenn er benötigt wird.
Bei der Erstellung eines Objektes wird zuerst der Konstruktor der Oberklasse aufgerufen, bis `Object` erreicht wird. anschließend werden alle Konstruktoren von oben nach unten ausgeführt. Definiert eine Oberklasse keinen Standard-Konstruktor, dann muss der Aufruf des Oberklassenkonstruktors manuell geschehen, was wir uns gleich genauer anschauen werden (Schlüsselwort `super`).
Es ist möglich, Konstruktoren ebenso wie Methoden zu überladen. Das bedeutet, es ist möglich, mehrere Konstruktoren in einer Klasse zu haben, welche die Klasse alle leicht unterschiedlich initialisieren oder auch nur Daten umwandeln. Hierzu ist es manchmal nötig, einen anderen Konstruktor aufzurufen.
Dies ist mit dem Schlüsselwort `this` möglich, was wir uns am besten an einem Beispiel anschauen:

```
1  ...
2
3  public Human(String hairColor, int birthYear) {
4      this.hairColor = hairColor;
5      this.birthYear = birthYear;
6  }
7
8  public Human(String hairColor, int ageInYears) {
9      this(hairColor, new Date().getYear() + 1900 - ageInYears);
10 }
11
12 ...
```

Hier wird in Zeile 9 der andere Konstruktor innerhalb der Klasse aufgerufen, welcher das eigentliche Initialisieren vornimmt. Es kann maximal einen `this`-Aufruf geben und dieser muss die erste Anweisung im Konstruktor geben. Gibt es nur einen Konstruktor oder wird kein anderer aufgerufen, so können wir den Aufruf weg lassen.
Um einen expliziten Konstruktor der Oberklasse aufzurufen, wird das Schlüsselwort `super` verwendet. Dies funktioniert ähnlich wie das Schlüsselwort `this`, allerdings wird ein Konstruktor der Oberklasse aufgerufen.

Warnung: Sowohl `this(...)` als auch `super(...)` müssen die erste Anweisung im Konstruktor sein. Somit ist es nicht möglich, beide Schlüsselworte zeitgleich zu verwendet!

Initializer-Block

Der *Initializer-Block* ist ein einfacher Block innerhalb der Klasse, der ohne Namen/Parametern/etc. in der Klasse steht:

```
1 public HelloInitializer {
2     {
3         // Initializer-Block.
4     }
5 }
```

Sämtlicher Code innerhalb dieses Blockes wird vor dem Durchlauf von jedem Konstruktoraufruf ausgeführt. Der Inhalt des Blocks wird also quasi an den Anfang, aber unter die `this`/`super`-Aufrufe, aller Konstruktoren kopiert, aber nur einmal pro Aufruf ausgeführt.

Static-Initializer-Block

Der *Static-Initializer-Block* ist ebenso wie der Initializer-Block ein einfach Block, der allerdings mit dem Schlüsselwort `static` markiert wird:

```
1 public HelloStaticInitializer {
2     static {
3         // Static-Initializer-Block.
4     }
5 }
```

Der Code in diesem Block wird ausgeführt, sobald die Klasse geladen wird (das heißt vor sämtlichem anderen Code innerhalb der Klasse).

Ausführungsreihenfolge

Schauen wir uns die Ausführungsreihenfolge von Methoden, Konstruktoren, Initializer-Blöcken und so weiter noch einmal an einem Beispiel an:

```
class ParentClass {
    static {
        // 1
    }

    {
        // 3
    }

    ParentClass(String s) {
        // 4
    }

    void doIt() {
        // 8
    }
}

class ChildClass extends ParentClass {
    static {
        // 2
    }

    {
        // 5
    }

    ChildClass(String s) {
        super(s);

        // 6
    }

    ChildClass() {
        this("Hello, World!");

        // 7
    }

    @Override
    void doIt() {
        super.doIt();

        // 9
    }
}
```

Bei einem Aufruf `new ChildClass().doIt()` wird der Code nun in der oben angegebenen Reihenfolge ausgeführt. Die genaue Bedeutung des `@Override` werden wir im Abschnitt 4.6.3 kennen lernen.

4.6.2 Referenzen

In Java greifen wir auf Objekte durch sogenannte *Referenzen* zu. Diese können wir uns als „Zeiger“ vorstellen, die auf das eigentliche Objekt zeigen. Um ein neues Objekt einer Klasse `A` anzulegen und die Referenz auf dieses neu angelegte Objekt in eine Variable a vom Typ `A` zu speichern, nutzen wir den Operator `new`:

```
A a = new A();
```

Mit dem `new`-Operator und den geschweiften Klammern wird der Konstruktor der Klasse `A` aufgerufen.

Literal `null`

Wollen wir in einer Referenzvariablen (noch) kein Objekt referenzieren oder die Referenz auf ein vorhandenes Objekt entfernen, so können wir diese Variable auf den Wert `null` setzen. Somit zeigt die Variable ins nichts (Null Pointer) und es kann durch diese auf kein Objekt zugegriffen werden.

> **Warnung:** Wir müssen vor jedem Zugriff (Methodenaufruf, Attributzugriff) auf eine Variable prüfen, dass diese nicht `null` ist. Greifen wir auf eine Null-Referenz zu, so wird eine `NullPointerException` (NPE) geworfen und die Ausführung bricht ab.

Vergleich zu primitiven Daten

Bei primitiven Daten werden direkt die Daten in der Variable abgelegt und keine Referenz auf diese. Was dies für einen Unterschied macht werden wir im Abschnitt 4.6.2 genauer sehen. Ebenfalls kann eine primitive Variable nicht `null` sein, was an vielen Stellen ein Vorteil sein kann, den wir uns auch zu Nutzen machen sollten. Des weiteren werden die Datentypen von primitiven Datentypen im Allgemeinen klein geschrieben, wobei Klassen mit einem Großbuchstaben beginnen.

Sonderfall `String`

Einen Sonderfall stellt die Klasse `String` dar, da diese sich an einigen (nicht allen!) Stellen wie ein primitiver Datentyp verhält. Dies liegt an folgenden Punkten:

- Ein String ist *immutable*, das heißt nicht veränderlich. Wurde einmal ein String-Objekt angelegt, so kann dieses nicht mehr verändert werden.
- String-Objekte werden implizit erstellt, wenn eine Zeichenkette in Anführungszeichen gesetzt wird. Das bedeutet, statt `new String(new char[] { 'a', 'b', 'c'})` können wir `"abc"` schreiben.

> **Warnung:** Trotz dieser Gleichheiten können wir Strings nicht mit == vergleichen, sondern müssen die Methode `equals(..)` verwenden! Hierzu siehe 4.6.2.

Zuweisen vs. Kopieren

Bei der Zuweisung von Referenzen mit dem Gleichheitszeichen wird ausschließlich die Referenz auf das Objekt kopiert und nicht der Inhalt des Objektes. Das heißt für uns, eine Modifikation an einer der Variablen spiegelt sich bei der anderen wieder, da die Variablen die gleiche Referenz enthalten und somit auf das selbe Objekt verweisen. Wenn wir eine Methode aufrufen und eine Referenz als Parameter übergeben, müssen wir genau dies beachten: Wir geben ausschließlich die Referenz weiter und diese verweist auf das selbe Objekt. Dies kann zu ungewollten Seiteneffekten führen.
Schauen wir uns dies an einem Beispiel an:

```
1  public class Foo {
2      public int bar = 1;
3  }
4
5  public class Main {
6      public void baz() {
7          Foo a = new Foo();
8          Foo b = a;
9          Foo c = b;
10
11         // 1:  a.bar == b.bar == c.bar == 1
12
13         a.bar = 2;
14
15         // 2:  a.bar == b.bar == c.bar == 2
16
17         b.bar = 3;
18
19         // 3:  a.bar == b.bar == c.bar == 3
20
21         c.bar = 4;
22
23         // 4:  a.bar == b.bar == c.bar == 4
24
25         boo(c);
26
27         // 5:  a.bar == b.bar == c.bar == 5
28     }
29
30     private void boo(Foo foo) {
31         foo.bar = 5;
32     }
33 }
```

Wir sehen, dass sich sämtliche Änderungen auf `a`, `b`, `c` jeweils auf den anderen Variablen widerspiegeln. Auch spiegeln sich die Änderungen der Methode `boo(..)` auf den entsprechenden Objekten wider.
Ist dies nicht das gewünschte Verhalten, so müssen wir das Objekt kopieren, das heißt wir müssen eine neue Instanz der entsprechende Klasse anlegen und den Wert jedes einzelnen Feldes kopieren. Hierzu gibt es mehrere Ansätze:

- Copy-Konstruktor. Wir erstellen einen Konstruktor, der als Parameter eine Instanz der Klasse selber annimmt und aus dieser die Daten kopiert.
- Copy-Methode. Wir erstellen eine Objektmethode, die eine neue Instanz der Klasse erstellt und die Daten in die entsprechende Klasse kopiert.

Eine solche Implementierung kann zum Beispiel so aussehen:

```
1  public class Foo {
2      public int bar = 1;
3
4      // Standard-Konstruktor
5      public Foo() { }
6
7      // Copy-Konstruktor
8      public Foo(Foo foo) {
9          this.bar = foo.bar;
10     }
11
12     // Copy-Methode
13     public Foo copy() {
14         Foo foo = new Foo();
15         foo.bar = this.bar;
16         return foo;
17     }
18 }
```

Mit diesen beiden Verfahren lässt sich ein Objekt nun kopieren.

Test auf Gleichheit und Identität

Um Objekte auf Identität/Gleichheit zu prüfen, existiert einerseits der Operator == und die Methode `equals(..)`. Während bei primitiven Datentypen ausschließlich der Operator == angewandt werden kann, macht dies bei Objekten einen großen Unterschied:

- Der Operator == prüft auf *Identität*, das heißt es wird geprüft, ob zwei Objekte auf dem selben Speicher liegen. Wertet der Ausdruck `a == b` zu `true` aus, so ist `a` das Selbe wie `b`.
- Die Methode `equals(..)` prüft auf *Gleichheit*, das heißt es wird geprüft, ob der Inhalt von zwei Objekten gleich ist. Wertet der Ausdruck `a.equals(b)` zu `true` aus, so ist `a` das Gleiche wie `b`.

> **Warnung:** Da die Methode `equals(..)` auf einem Objekt aufgerufen werden muss, kann es zu Fehler führen, wenn dieses Objekt `null` ist. Als Abhilfe kann hierbei auf die Hilfsmethode `Objects.equals(Object, Object)` zurückgegriffen werden, die genau diesen Fall überprüft.

Die Methode `equals(..)` ist in `Object` implementiert, greift in der Standard-Implementierung aber nur auf den == Operator zurück. Das bedeutet, wir müssen die Methode bei der Erstellung einer Klasse überschreiben, damit diese korrekt arbeitet.

Warnung: Überschreiben wir `equals(..)`, dann müssen wir auch `hashCode()` überschreiben, sodass die Gleichheit zweier Objekte die Gleichheit deren Hashcode impliziert. Was genau die Methode `hashCode()` tut, werden wir hier nicht behandeln, es sei aber erwähnt. In üblichen Tools wie Eclipse oder Lombok lassen sich die beiden Methoden leicht generieren.

Schauen wir uns diese ganze Theorie nochmal an einem Beispiel an:

```
1  public class Foo {
2      private final int bar;
3
4      public Foo(int bar) {
5          this.bar = bar;
6      }
7
8      public boolean equals(Object that) {
9          if (that == null) {
10             return false;
11         }
12         // bar hat einen primitiven Datentyp, weshalb wie hier den ==
               Operator
13         // einsetzen.
14         return this.bar == that.bar;
15     }
16
17     public int hashCode() {
18         return this.bar;
19     }
20 }
21
22 public class Main {
23     public void baz() {
24         Foo a = new Foo(1);
25         Foo b = new Foo(1);
26         Foo c = new Foo(2);
27
28         a == b;  // false
29         a == c;  // false
30         b == c;  // false
31
32         a.equals(b);  // true
33         a.equals(c);  // false
34         b.equals(c);  // false
35     }
36 }
```

Für eine korrekt Implementierung von `equals(..)` und `hashCode()` muss folgende gelten:

$$
\begin{aligned}
\texttt{a.equals(b)} \iff \texttt{a.hashCode()} &== \texttt{b.hashCode()} \\
\texttt{!a.equals(null)} & \\
\texttt{a} == \texttt{b} \implies \texttt{a.equals(b)} & \quad \text{(Konsistenz)} \\
\texttt{a.equals(a)} & \quad \text{(Reflexivität)} \\
\texttt{a.equals(b)} \iff \texttt{b.equals(a)} & \quad \text{(Symmetrie)} \\
\texttt{a.equals(b)} \land \texttt{b.equals(c)} \implies \texttt{a.equals(c)} & \quad \text{(Transitivität)}
\end{aligned}
$$

Downcasts

Der Typ einer Variable muss immer eine Oberklasse der zuzuweisenden Typs sein. Das bedeutet, eine `ArrayList` kann einer Variable des Typs `List` zugewiesen werden, aber nicht anders herum. Um eine möglichst hohe Wiederverwendbarkeit zu gewährleisten, sollten wir bei der Auswahl der Variablentypen darauf achten, den Typ so hoch wie möglich in der Klassenhierarchie zu wählen. Haben wir beispielsweise eine Variable mit dem Typ `ArrayList`, so können dieser ausschließlich `ArrayLists` zugewiesen werden. Haben wir jedoch eine Variable vom Typ `List`, so können wir dieser auch eine `LinkedList` zuweisen (statischer vs. dynamischer Typ, siehe 4.6.6).
Um diese allgemeine Typen wieder in Untertypen zu konvertieren, werden *Downcasts* benötigt. Siehe hierzu Abschnitt 4.2.5 über Downcasts.

4.6.3 Vererbung

Java unterstützt die die meisten Sprachen keine Mehrfachvererbung, dafür können jedoch beliebig viele *Interfaces* implementiert werden, in denen Methoden deklariert werden können, die von der Klasse zu implementieren sind. Wir werden uns in diesem Abschnitt jedoch nur um die eigentlich Vererbung kümmern.
Die Syntax um eine Klasse von einer anderen Klasse ergeben zu lassen ist wie folgt:

```
1 class <KLASSE> extends <OBERKLASSE> {
2     ...
3 }
```

Die einzige Voraussetzung ist, dass die Oberklasse nicht Final sein darf und mindestens einen sichtbaren Konstruktor besitzt. Jede Klasse, die keine explizite Oberklasse hat, erbt von der Klasse `Object`. Hierdurch ist `Object` die Oberklasse von allen existierenden Klassen und sämtliche auf `Object` definierten Methoden sind in allen Klassen verfügbar.

Methoden

Definiert die Oberklasse einige Methoden, so können diese in der Unterklasse *überschrieben* werden. Das heißt, die Implementierung einer Methode wird ausgetauscht. Wird eine Methode der gleichen Klasse aufgerufen, so wird immer die Implementierung ausgeführt, die am weitesten unten in der Klassenhierarchie ist. Das heißt, eine Klasse kann sich nicht darauf verlassen, das auch garantiert der korrekte Code ausgeführt wird. Manchmal möchten wir jedoch verhindern, dass eine bestimmte Methode überschrieben wird, wenn diese zum Beispiel sicherheitskritisch ist. Wir können dies verhindern, indem wir der Methode den Modifizierer `final` hinzufügen. Wollen wir explizit die

Implementierung der Oberklasse aufrufen, so müssen wir dies mit dem Schlüsselwort `super` verdeutlichen. Bei einem Aufruf mit `this` kann das `this` auch weggelassen werden, sofern der Aufruf eindeutig ist.
Schauen wir uns all diese Konzepte an einem Beispiel an:

```
1  class Creature {
2      public void doIt() {
3          // 1
4
5          doAnother();
6
7          // 4
8      }
9
10     protected void doAnother() {
11         // 2
12     }
13 }
14
15 class Human extends Creature {
16     @Override
17     protected void doAnother() {
18         super.doAnother();
19
20         // 3
21     }
22 }
```

Rufen wir die Methode `doIt()` nun mit `new Human().doIt()` auf, so wird der Code in der oben angegebenen Reihenfolge ausgeführt.

Zeile 5 Hier rufen wir nicht Zeile 10, sondern Zeile 18 auf, da wir eine Instanz der Klasse `Creature` erstellt haben.

Zeile 17 Hier rufen wir explizit die Implementierung in der Oberklasse, also Zeile 10 auf.

Mit der Annotation `@Override` markieren wir, dass eine Methode eine andere Methode überschreiben soll. Diese Annotation ist optional, hilft aber beim Lesen des Codes und sorgt außerdem dafür, dass ein Compiler-Fehler auftritt, sollte keine Methode überschrieben worden sein. Dies hilft uns dabei, Schreibfehler (zum Beispiel im Methodennamen) frühzeitig zu erkennen.

Variation von Sichtbarkeit, Sichtbarkeit und Exceptions

Bei dem Überschreiben von Methoden müssen wir darauf achten, dass der Name der Methode exakt gleich ist und die Typen der Parameter gleich strukturiert sind, das heißt die gleiche Reihenfolge und Anzahl haben und immer den gleichen oder einen weiter gefassten Typ vorweisen. Ähnliches gilt für den Rückgabetyp, der entweder gleich dem Rückgabetyp der Überschriebenen Methode sein muss oder konkreter. Das selbe gilt für Exceptions, die in der `throws`-Klausel gelistet sind. Die Sichtbarkeit einer Methode dürfen wie ebenfalls nicht weiter einschränken, allerdings dürfen wir sie erhöhen.
Zusammengefasst:

- Der Name der überschriebenen Methode muss exakt gleich sein.
- Die Anzahl der Parameter muss gleich sein.
- Der Typ eines jeden Parameters gleich dem Typ des Parameter der überschriebenen Methode sein.
- Ist der Rückgabetyp der überschriebenen Methode `void`, so muss der Rückgabetyp der überschreibenden Methode ebenfalls `void` sein.
- Der Rückgabetyp der überschreibenden Methode muss ein Untertyp oder der gleiche Typ wie der Rückgabetyp der überschriebenen Methode sein.
- Jede deklarierte Exception muss eine Unterexception oder gleich der deklarierten Exception der überschriebenen Methode sein. Ebenfalls darf die überschreibende Methode keine Exceptions deklarieren. Es ist jedoch nicht möglich, neue Exceptions zu deklarieren.
- Die Sichtbarkeit der überschreibenden Methode muss gleich oder weiter gefasst sein als die Sichtbarkeit der überschriebenen Methode.
  `public` > `protected` > `Default` > `private`

Beispiel:

```
1  class Parent {
2      protected CharSequence format(Integer a) { /* ... */ return null; }
3  }
4
5  class Child1 extends Parent {
6      @Override
7      public String format(Integer a) { /* ... */ return null; }
8  }
9
10 class Child2 extends Parent {
11     @Override
12     protected CharSequence format(Integer a) { /* ... */ return null; }
13 }
14
15 class Child3 extends Parent {
16     @Override
17     public Object format(Integer a) { /* ... */ return null; }
18 }
19
20 class Child4 extends Parent {
21     @Override
22     CharSequence format(Integer a) { /* ... */ return null; }
23 }
```

In diesem Beispiel sind die Klassen `Child1` und `Child2` valide, da sie entweder den gleichen oder einen genaueren Rückgabetyp haben und `Child3` falsch, da der Rückgabetyp allgemeiner ist. `Child4` ist nicht valide, da die Sichtbarkeit weiter eingeschränkt wurde.

Attribute

Die Attribute einer Klasse werden ebenfalls mit vererbt, sofern das Feld eine mindestens eine Sichtbarkeit von `protected` hat.
Im Gegensatz zu Methoden wird jedoch trotzdem immer auf das Feld in der aktuellen Klasse zugegriffen und nicht auf ein mögliches weiteres Feld mit dem gleichen Namen in einer Unterklasse, da hier das Prinzip der Polymorphie nicht gilt. Wir könnten jedoch einen sogenannten „Getter“ implementieren, der ausschließlich den Wert eines Feldes zurück gibt und diesen überschreiben:

```
1  class Parent {
2      public String field = "parent";
3
4      public void doIt() {
5          System.out.println(field);
6          System.out.println(getField());
7      }
8
9      public String getField() {
10         return field;
11     }
12 }
13
14 class Child extends Parent {
15     public String field = "child";
16
17     @Override
18     public void doIt() {
19         super.doIt();
20
21         System.out.println(field);
22         System.out.println(super.field);
23         System.out.println(getField());
24         System.out.println(super.getField());
25     }
26
27     public String getField() {
28         return field;
29     }
30 }
```

Wenn wir den obigen Code nun mit `new Child().doIt()` aufrufen, so werden die folgenden Zeilen ausgegeben:

```
parent
child
child
parent
child
parent
```

4.6.4 Abstrakte Klassen

In Java werden abstrakte Methoden sowie abstrakte Klassen mit dem Schlüsselwort `abstract` markiert:

```
1 public abstract class Example {
2     public abstract void doIt();
3 }
```

Wir sehen, dass eine abstrakte Methode keine Implementierung besitzt und der Methodenkörper inklusive der Klammern durch ein Semikolon ersetzt wird. Wenn wir von einer abstrakten Klasse erben, müssen wir in dieser entweder alle abstrakten Klassen der Oberklasse(-n) implementieren oder die Klasse selbst als abstrakt markieren:

```
1 public class ConcreteExample extends Example {
2     @Override
3     public void doIt() { /* ... */ }
4 }
5
6 public abstract class ExtendedExample extends Example { /* ... */ }
```

Unsere Klasse `ConcreteExample` implementiert hier die Methode `doIt()`, die Klasse `ExtendedExample` muss als abstrakt markiert werden, da sie die Methode nicht implementiert.

4.6.5 Interfaces

Wenn wir eine abstrakte Klasse schreiben, die ausschließlich abstrakte Methoden definiert, so sollten wir uns fragen, ob ein Interface an dieser Stelle nicht angemessener wäre. Da ein Interface ausschließlich abstrakte `public`-Methoden implementieren kann, können diese beiden Modifizierer weg gelassen werden. Ein Interface wird mit dem Schlüsselwort `interface` eingeleitet.

```
1 public interface Example {
2     void doIt();
3 }
```

Die Methode `doIt()` ist aus den oben beschriebenen Gründen implizit `public` und abstrakt.
Im Gegensatz zu der Vererbung von Klassen sprechen wir bei Interfaces von „Implementierung“, das heißt eine Klasse *implementiert* ein Interface und erbt nicht von diesen. Auch kennzeichnen wir eine solche Implementierung anders, nämlich mit der folgenden allgemeinen Syntax:

```
1 <SICHTBARKEIT> class <KLASSENNAME> implements <INTERFACENAME>,
     <INTERFACENAME>, ... {
2     ...
3 }
```

Wir sehen hier auch schon, dass eine Klasse mehrere Interfaces implementieren kann. Dies ermöglicht uns, auch einfache Funktionen (zum Beispiel Vergleichbarkeit von Objekten) in Interfaces zu definieren und an vielen Stellen nutzen zu können. Ein Beispiel ist hierbei das Interface `java.util.Comparable`,

welches eine Methode `int compareTo(Object other)` definiert, um das aktuelle Objekt mit einem anderen Objekt zu vergleichen und eine Ordnungsrelation über diesen zu erstellen.
Ein Interface kann außerdem andere Interfaces erweitern, das heißt Methoden zu der Definition hinzufügen. Dies können wir mit dem Schlüsselwort `extends` bei der Interfacedefinition angeben:

```
<SICHTBARKEIT> interface <INTERFACENAME> extends <INTERFACENAME>,
   <INTERFACENAME>, ... {
    ...
}
```

Auch hier können wir mehrere Interfaces erweitern.

Default-Methoden

Seit Java 8 ist es möglich, in Interfaces Default-Implementierungen für Methoden anzugeben, die dann in den implementierenden Klassen überschrieben werden können. Der Unterschied zu einer abstrakten Klasse besteht dabei darin, dass eine Klasse weiterhin mehrere Interfaces implementieren kann und somit auch Funktionalitäten von Interfaces erben kann.
Die Syntax für Default-Methoden schauen wir uns an einem kleinen Beispiel an:

```
public interface ExampleA {
    void doIt(String str);

    default void doIt(String[] strs) {
        for (String str : strs) {
            doIt(str);
        }
    }
}

public interface ExampleB {
    default void doIt(String[] strs) {
        for (String str : strs) {
            System.out.println(str);
        }
    }
}
```

Hier haben wir zwei Interfaces erstellt, die beide eine Default-Implementierung für `doIt(String[])` vorschreiben. Wenn wir nun eine Klasse erstellen, die Interface `ExampleA` implementiert, so muss diese Klasse nur die Methode `doIt(String)` implementieren und erbt die Funktionalität `doIt(String[])` von dem Interface.
Dies führt uns allerdings zu einem großen Problem: Erstellen wir eine andere Klasse, sie sowohl `ExampleA` als auch `ExampleB` implementiert, so erbt die Klasse zwei Implementierungen für `doIt(String[])` und Java weiß nicht, welche der Implementierungen ausgeführt werden soll. Um dieses Problem zu umgehen kann Code mit diesem Phänomen nicht kompilieren und die Methode muss in der Klasse überschrieben werden.

Funktionale Interfaces

Ein Interface, welches nur eine Methode definiert wird *funktionales Interface* genannt. Außerdem sollte ein solches Interface mit der Annotation `@FunctionalInterface` versehen werden, welche den Compiler dazu bringt, zu Prüfen, ob in dem Interface wirklich nur eine Methode definiert wird. Der Begriff des funktionalen Interfaces wurde in Java 8 mit der Einführung von Lambdas eingeführt. was dies genau zu bedeuten hat werden wir uns im Abschnitt 4.6.8 genauer anschauen.

Interfaces in `java.util.function`
In dem Package `java.util.function` sind einige funktionale Interfaces definiert, welche an vielen Stellen genutzt werden können, an denen Funktionen als Parameter angenommen werden und ähnliches (siehe auch hierzu 4.6.8).
Es existieren die folgenden Interfaces und die entsprechenden Methoden:

Interface	Methode	Beschreibung
`BiConsumer<T, U>`		

4.6.6 Polymorphie und späte Bindung

Statischer vs. Dynamischer Typ

Um Polymorphie zu verstehen, müssen wir uns zuerst klar machen, was der Unterschied zwischen einem statischen und einem dynamischen Typ ist:

- Der statische Typ einer Variable ist der Typ, den wir bei der Deklaration der Variable angegeben haben.
- Der dynamische Typ einer Variable ist der Typ, der „am Ende drin steckt“, also die Klasse, deren Instanz in der Variable gespeichert ist.

Beispiel:

```
List list = new ArrayList();
```

In diesem Beispiel ist `List` der statische und `ArrayList` der dynamische Typ.

Polymorphie und späte Bindung

Während des Compile-Vorgangs ist es nur möglich, Methoden aufzurufen, die auf dem statischen Typ definiert worden sind. Es wird allerdings erst zur Laufzeit entschieden, welche Implementierung aufgerufen wird. Hierbei wird nicht die Implementierung in dem statischen Typ, sondern die Implementierung in dem dynamischen Typ aufgerufen. Dieses Phänomen wird *späte Bindung* genannt. Auch innerhalb von Klassen bei Aufrufen von Methoden der selben Instanz ist die Polymorphie anzutreffen, dies haben wir bereits im Abschnitt zu dem Überschreiben von Methoden (4.6.3) gesehen.

4.6.7 Verschachtelte Klassen

Üblicherweise legen wir für jede Klasse eine neue Datei an. Es ist aber auch möglich, innerhalb einer Klasse weitere Klassen zu definieren, was folgende Vorteile hat:

- Die sogenannte *innere Klasse* kann auch auf private Attribute der äußeren Klasse zugreifen.
- Die innere Klasse kann von der Instanz der äußeren Klasse abhängig sein, sodass auch auf Instanzvariablen der äußeren Klasse zugegriffen werden kann.

Oftmals nutzen wir innere Klassen aber auch einfach dazu, Dateien für sehr kleine Klassen zu vermeiden, die wir ausschließlich in der äußeren Klasse benötigen.
Eine innere Klasse kann die gleichen Sichtbarkeiten haben wie Attribute und Methoden, also `private`, Default, `protected` und `public`. Die Definition der inneren Klasse wird dabei einfach in den Körper der äußeren Klasse geschrieben:

```
1 public class OuterClass {
2     <SICHTBARKEIT> [static] class <KLASSENNAME> {
3         ...
4     }
5 }
```

Wir sehen hier auch schon, dass eine innere Klasse statisch sein kann.

Statische verschachtelte Klassen

Eine statische verschachtelte Klasse benötigt für ihre Existenz keine Instanz der äußeren Klasse, das heißt wir können ohne Referenz auf eine Instanz auf die äußere Klasse auf die innere Zugreifen. Schauen wir uns dies an einem Beispiel an:

```
1 public class OuterClass {
2     public static class StaticInnerClass {
3         ...
4     }
5 }
```

Eine Instanz der inneren Klasse können wir nun mit `new OuterClass.StaticInnerClass()` erstellen.

Innere Klassen

Definieren wir eine innere Klasse nicht als statisch, so wird eine Instanz der äußeren Klasse benötigt, um eine Instanz der inneren Klasse zu erstellen. Schauen wir uns auch dies wieder an einem Beispiel an:

```
1 public class OuterClass {
2     public class InnerClass {
3         ...
4     }
5 }
```

Um nun eine Instanz der inneren Klasse zu erstellen, benötigen wir eine Instanz der äußeren Klasse. Ist diese in einer Variable `outer` gespeichert, so können wir die Instanz der inneren Klasse mit `outer.new InnerClass()` erstellen. Um dies in einem Schritt zu tun können wir auch `new OutClass().new InnerClass()` verwenden.

Merkwürdige Konstrukte

Die Kombination aus statischen und nicht-statischen inneren Klassen kann zu merkwürdigen Konstrukten führen. Nehmen wir an, wir haben eine äußere Klasse `Outer`, eine normale innere Klasse `Inner` und eine statische innere Klasse `StaticInner`, wobei letztere eine Unterklasse von `Inner` ist. Dann müssen wir den Konstruktor wie folgt gestalten, damit der Code kompiliert:

```
1  public class Outer {
2      public class Inner { }
3
4      public static class StaticInner extends Inner {
5          public StaticInner(Outer outer) {
6              outer.super();
7          }
8      }
9  }
```

Um eine Instanz von `StaticInner` er erhalten, müssen wir dem Konstruktor von `StaticInner` also eine Instanz der äußeren Klasse übergeben: `new Outer.StaticInner(new Outer())`.

Anonyme Innere Klassen

Bisher haben wir nur Klassen kennengelernt, die einen Namen haben. Manchmal kann es jedoch sinnvoll sein, eine Klasse ad-hoc zu definieren und ihr keinen Namen geben zu müssen. Eine anonyme innere Klasse ist also eine Klasse, die wir mitten im Code definieren und implementieren. Dies wird meist genutzt, um einfache Interfaces zu implementieren und diese Implementierung direkt an eine Methode weiterzugeben.
Schauen wir uns ein Beispiel an:

```
1  @FunctionalInterface
2  public interface SimpleInterface {
3      void doIt();
4  }
5
6  public class Main {
7      public void foo() {
8          bar(new SimpleInterface() {
9              @Override
10             public void doIt() {
11                 // Implementierung der anonymen Klasse.
12             }
13         });
14     }
15
16     private void bar(SimpleInterface si) {
17         si.doIt();
18     }
19 }
```

Hier erstellen wir in Zeile 7 eine Instanz des Interfaces `SimpleInterface`. Da es aber nicht möglich ist, direkt eine Instanz eines Interfaces oder einer abstrakten Klasse zu erzeugen, müssen wir das Interface ad-hoc implementieren. Dies tun wir, in dem wir nach dem Konstruktoraufruf einen Codeblock einfügen, der die entsprechenden Methode implementiert.
Dieses Verfahren können wir übrigens auf alle Klasse, auch auf nicht-abstrakte, anwenden und damit ad-hoc Implementierungen austauschen, ohne eine separate benannte Klasse zu erstellen.

4.6.8 Lambda-Ausdrücke

Wie wir im vorherigen Abschnitt gesehen haben, können wir anonyme Klassen nutzen, um ad-hoc Interfaces zu implementieren. Da aber gerade für Interfaces mit nur einer Methode (genannt funktionale Interfaces) viel überflüssiger Code geschrieben wird, können wir Lambda-Ausdrücke nutzen, um den Code zu vereinfachen. Lambda-Ausdrücke generieren dabei mit einer speziellen Syntax ebenfalls anonyme Klassen, dies wird aber vor uns versteckt und geschieht im Hintergrund.
Schauen wir uns den Code von oben nun nochmals mit einem Lambda-Ausdruck statt der anonymen Klasse an:

```
1  @FunctionalInterface
2  public interface SimpleInterface {
3      void doIt();
4  }
5
6  public class Main {
7      public void foo() {
8          bar(() -> {
9              // Implementierung der anonymen Klasse.
10         });
11     }
12
13     private void bar(SimpleInterface si) {
14         si.doIt();
15     }
16 }
```

Diese Code ist äquivalent zu obigem Code. Die in Zeile 7 genutzte Syntax führt einen Lambda-Ausdruck ein, welcher den Code in Zeile 8 als Implementierung der Methode `doIt()` enthält. Das bedeutet, der Lambda-Ausdruck erstellt eine anonyme Klasse und fügt den Code als Implementierung der einzigen Methode des Interfaces ein.
Ein Lambda-Ausdruck kann auch Parameter annehmen, diese müssen wir dann in den runden Klammern vor dem Pfeil angeben (der Typ der Parameter ist optional, da der Compiler diesen aus der Methodendefinition lesen kann). Gibt es keine Parameter, so müssen die Klammern leer sein und bei nur einem Parameter können die Klammern weg gelassen werden. Innerhalb des Codeblocks nach dem Pfeil muss ein Return stehen, sofern die zu implementierende Methode einen Rückgabetyp ungleich `void` definiert. Besteht der Codeblock nur aus einer Zeile, so können wir den gesamten Codeblock sowie das Return entfernen.

Methoden-Referenzen

Wollen wir die Implementierung eines Lambdas einfach an eine andere Methode delegieren, so können wir eine Methoden-Referenz erstellen, die eine Implementierung referenziert. Die Syntax für eine

solche Methodenreferenz ist `<OBJEKT>::<METHODENNAME>`, wobei das Objekt das Objekt ist, auf dem die Methode aufgerufen werden soll (bei statischen Methoden ist dies die Klasse) und der Methodenname der Name der Methode ist, die aufgerufen werden soll. Beispielsweise ist `System.out::println` eine Methoden-Referenz auf die Methode `println` auf dem Objekt `System.out`.

Die Parameter sowie der Rückgabetyp der referenzierten Methode müssen dabei zu den Parameterdefinitionen in dem zu implementierenden Interface kompatibel sein.

Abschlussbeispiel

Schauen wir uns nochmals alle Vereinfachungsschritte an einem Beispiel an:

```java
@FunctionalInterface
public interface IntegerComparator {
    // a < b  -->  -1;   a > b  -->  1;   a == b  -->  0
    int compare(int a, int b);
}

public class Main {
    public void bar() {
        // Ausgangsstand.
        baz(new IntegerComparator() {
            @Override
            public int compare(int a, int b) {
                if (a < b) {
                    return -1;
                }
                if (a > b) {
                    return 1;
                }
                return 0;
            }
        });

        // Nutzung des ternaeren Operators.
        baz(new IntegerComparator() {
            @Override
            public int compare(int a, int b) {
                return (a < b ? -1 : (a > b ? 1 : 0));
            }
        });

        // Einfuehrung eines Lambdas.
        baz((int a, int b) -> {
            return (a < b ? -1 : (a > b ? 1 : 0));
        });

        // Wegnahme der Parametertypen.
        baz((a, b) -> {
            return (a < b ? -1 : (a > b ? 1 : 0));
        });

        // Wegnahme des Codeblocks.
        baz((a, b) -> (a < b ? -1 : (a > b ? 1 : 0)));

        // Referenzierung der compare-Methode von Integer.
        baz(Integer::compare);
    }

    public void baz(IntegerComparator comparator) {
        comparator.compare(1, 2);
    }
}
```

Dabei ist folgendes äquivalent:

- Zeilen 9 bis 20
- Zeilen 23 bis 28
- Zeilen 31 bis 33
- Zeilen 36 bis 38
- Zeile 41
- Zeile 44

4.6.9 Enumerations

Eine besondere Art der Klasse stellen *Enumerations* dar, also Aufzählungen. Ein Enum ist eine einfache Klasse, die einige Konstanten definiert.
Hiermit können wir beispielsweise eine Aufzählung `Tier` := {`Hund`, `Strauss`, `Fisch`} definieren:

```
1  public enum Tier {
2      HUND,
3      STRAUSS,
4      FISCH
5  }
```

In einem Enum schreiben wir üblicherweise alle Einträge in Capslock und trennen einzelne Wort mit einem Unterstrich. Diese Konvention kommt daher, dass die Elemente einer Aufzählung Instanzen der Klasse der Aufzählung (also in diesem Fall der Klasse `Tier`) sind. Es sind also statische Klassenattribute der Klasse `Tier`.
Das führt uns auch dazu, dass Elemente eines Enums Eigenschaften haben können, die die Elemente einem Konstruktor übergeben können, den wir definieren müssen:

```
1  public class Tier {
2      HUND(4),
3      STRAUSS(2),
4      FISCH(0);
5
6      private final int legs;
7
8      private Tier(int legs) {
9          this.legs = legs;
10     }
11 }
```

Die Parameter, die wir in den Klammern hinter dem Namen der Elemente angeben werden also an den Konstruktor weitergegeben. Des weiteren muss der Konstruktor eines Enums privat sein, damit keine weiteren Elemente erstellt werden können.
Zusätzlich können wir in dem Körper des Enums normale sowie abstrakte Methoden definieren, die dann (wie bei anonymen inneren Klasse) an dem entsprechenden Element implementiert werden müssen:

```java
public class Tier {
    HUND(4) {
        @Override
        public void move() {
            // Laufen...
        }
    },
    STRAUSS(2) {
        @Override
        public void move() {
            // Laufen...
        }
    },
    FISCH(0) {
        @Override
        public void move() {
            // Schwimmen...
        }
    };

    private final int legs;

    private Tier(int legs) {
        this.legs = legs;
    }

    public abstract void move();
}
```

Abbildung 4.14: Enum: Abstrakte Methoden

Klasse `java.lang.Enum<E>`

Wie wir bereits gesehen haben sind die Elemente eines Enums Instanzen der definierenden Klasse. Tatsächlich übersetzt der Compiler ein Enum zu einer Klasse, die von der Klasse `java.lang.Enum<E>` erbt und damit einige Methoden erbt:

- `compareTo(other: E): int`
  Vergleicht des aktuellen mit der Ordinalzahl des übergebenen Elements (in obigem Beispiel gilt `HUND` < `STRAUSS` < `FISCH`).

- `equals(other: Object): boolean`
  Prüft das aktuelle und das übergebene Objekt auf Gleichheit.

- `name(): String`
  Gibt den Namen des aktuellen Elements zurück.

- `ordinal(): int`

Gibt die Ordinalzahl (das heißt den Index innerhalb des Enums) zurück. In obigem Beispiel gilt:

$$x.\texttt{ordinal()} = \begin{cases} 0 & \text{falls } \texttt{x} = \texttt{HUND} \\ 1 & \text{falls } \texttt{x} = \texttt{STRAUSS} \\ 2 & \text{falls } \texttt{x} = \texttt{FISCH} \end{cases} \tag{4.1}$$

- `toString(): String`
  Gibt den Namen des aktuellen Elements zurück.

- `valueOf(name: String)`
  Statische Methode auf der entsprechenden Enum-Klasse. Gibt das passende Element zu dem übergebenem Namen zurück. Existiert kein solches Element, so wird `null` zurück gegeben. In obigem Beispiel gilt somit:

$$\texttt{Tier.valueOf}(x) = \begin{cases} \texttt{HUND} & \text{falls } \texttt{x} = \texttt{"HUND"} \\ \texttt{STRAUSS} & \text{falls } \texttt{x} = \texttt{"STRAUSS"} \\ \texttt{FISCH} & \text{falls } \texttt{x} = \texttt{"FISCH"} \end{cases} \tag{4.2}$$

- `values(): E[]`
  Statische Methode auf der entsprechenden Enum-Klasse. Gibt ein Array mit allen Elementen des Enums zurück.

Das bedeutet, die Klasse in Abbildung 4.14 wird durch den Compiler in etwa in folgende Klasse übersetzt:

```java
public abstract class Tier extends Enum<Tier> {
    public static final Tier HUND = new Tier(4) {
        public void move() { /* Laufen... */ }
        public int ordinal() {  return 0;  }
        public String name() {  return "HUND";  }
    };
    public static final Tier STRAUSS = new Tier(2) {
        public void move() { /* Laufen... */ }
        public int ordinal() {  return 1;  }
        public String name() {  return "STRAUSS";  }
    };
    public static final Tier FISCH = new Tier(0) {
        public void move() { /* Schwimmen... */ }
        public int ordinal() {  return 2;  }
        public String name() {  return "FISCH";  }
    };

    private final int legs;

    private Tier(int legs) {
        this.legs = legs;
    }

    public abstract void move();

    public abstract int ordinal();

    public abstract String name();

    public static Tier valueOf(String name) {
        for (Tier tier : values()) {
            if (tier.name().equals(name)) {
                return tier;
            }
        }
        return null;
    }

    public static Tier[] values() {
        return new Tier[] { HUND, STRAUSS, FISCH };
    }
}
```

Vererbung

Wie wir bereits gesehen haben können wir in Enums durchaus normale sowie abstrakte Methoden definieren. Außerdem können wir in einem Enum durchaus beliebig viele Interfaces implementieren, allerdings keine Klassen erweitern. Auch eine Enum-Klasse selbst ist `final`, dass heißt es können keine Unterklassen von Enums gebildet werden.

Methoden, die in den Interfaces definiert sind, müssen entweder in der Enum-Klasse selbst oder ähnlich wie die abstrakten Methoden bei jedem Element einzeln implementiert werden. Es ist auch möglich, in der Enum-Klasse eine „Default“-Implementierung zu erstellen und diese in den einzelnen Elementen zu überschreiben.

4.7 Generische Programmierung

Wir werden in diesem Kontext nur Typparameter (Generics) betrachten, welche in Java eine große Rolle spielen, wenn es um generische Programmierung geht.

4.7.1 Generics

In diesem Abschnitt betrachtet wir Generics, ein Konzept zur generischen Programmierung und Typsicherheit, welches mit der Java-Version 1.5 eingeführt wurde. Wie wir bereits wissen, ist Java eine statisch typisierte Programmiersprache, das heißt der korrekte Typ muss schon zur Compilezeit feststehen.

Motivation
Betrachtet wir zur Motivation eine Klasse `List`, welche die folgenden Methoden aufweist:

`add(element: Object)`: void	Fügt ein Element zu der Liste hinzu.
`get(index: int)`: Object	Gibt das Element an der Position `index` zurück. Existiert kein solches Element, wird `null` zurück gegeben.
`size(): int`	Gibt die Anzahl der Element der Liste zurück.

Die Klasse selbst stellt eine Auflistung von Elementen dar und kann keine `null`-Elemente enthalten. Auffällig ist, dass sämtliche Element als Object-Referenz vorliegen. Es gibt für eine Methode, welche eine solche Liste annimmt, gibt es keine Möglichkeit, den Typ der Parameter einzuschränken. Schauen wir uns hierzu eine Methode an, welche alle Elemente aufsummieren soll. Natürlich ergibt eine solche Operation nur auf Zahlen Sinn, weshalb davon ausgegangen wird, dass sämtliche Elemente der Liste vom Typ `Integer` sind:

```
1 public Integer sum(List list) {
2     Integer result = 0;
3     for (int i = 0; i < list.size(); i++) {
4         result += (Integer) list.get(i);
5     }
6     return result;
7 }
```

Abbildung 4.15: Java: Generics Motivation

Hier tut sich folgendes Problem auf: Wenn der Aufrufer falsche Elemente in die Liste getan hat, fliegt uns der Algorithmus an der markierten Stelle um die Ohren: Es wird eine `ClassCastException`.
Nun haben wir zwei Möglichkeiten:

1. Den korrekten Aufruf dem Aufrufer überlassen und annehmen, dass wir korrekte Daten bekommen.

2. Die Korrektheit des Aufrufs vor der eigentlichen Ausführung überprüfen.
3. Die Typprüfung des Compilers nutzen, damit wir erst gar keine falschen Typen bekommen können.

Möglichkeit 1 mag bei solch kleinen Beispielen noch funktionieren, aber spätestens, wenn wir die Liste an andere Methoden weitergeben und nicht selbst für den Fehler sorgen, wird die Fehlersuche schrecklich. Ebenso ergeht es Möglichkeit 2, denn bei tief verschachtelten Klassenstrukturen bleibt das Prüfen der Korrektheit nicht so trivial wie in unserem Beispiel.

> **Info:** Generell gilt: Fail Fast, am besten noch zur Compilezeit. Dies vereinfacht die Fehlersuche erheblich. Fehler, die erst in den Tiefen eines Programms passieren, sind schwer zu finden!

Somit bleibt uns als einzig gute Möglichkeit die Typprüfung des Compilers zu nutzen, welche auch genau dazu dient. Es wäre also eine Möglichkeit, eine Unterklasse `IntegerList` von `List` zu erstellen, die nur `Integers` akzeptiert. Das dies eine schlechte Idee ist, würden wir spätestens nach der zehnten, bis auf den Typ identischen, Klasse merken. Wäre es nicht toll, alles nur einmal schreiben zu müssen und den Typ nachher setzen zu können? Hier helfen uns die zuvor genannten Generics: Diese ermöglichen genau das.
Klassen, welche Generics nutzen, sind für viele Typen geschrieben und ermöglichen die Angabe des expliziten Typs in Spitzen Klammern hinter dem Klassennamen. Des weiteren ist es auch auf Methodenebene möglich, für viele Typen generisch zu Programmieren und bei der Nutzung der Methode den eigentlichen Typ festzulegen. Wir werden uns diese beiden Fälle in den folgenden beiden Abschnitten zu generischen Klassen und Methoden anschauen. Um einen ersten Eindruck zu kriegen, was uns Generics ermöglichen, hier ein Beispiel, welche die obige Methode ersetzen kann und eine größere Typsicherheit ermöglicht:

```
1 public Integer sum(List<Integer> list) {
2     Integer result = 0;
3     for (int i = 0; i < list.size(); i++) {
4         result += list.get(i);
5     }
6     return result;
7 }
```

Abbildung 4.16: Java: Generics Motivation: Nutzung von Generics

Wir haben hier nun die Typen in den Spitzen Klammern hinter `List` festgelegt, wodurch uns die Methode `get(...)` direkt Elemente des Typs `Integer` gibt. Wir können somit einfach die Zahlen aufsummieren. Somit wird ein möglicher Aufrufer nun schon beim Kompilieren merken, dass der Aufruf fehlschlägt. Da wir `null`-Elemente ausgeschlossen haben, wird der Code nun immer funktionieren, sofern dieser erfolgreich Kompiliert. Weshalb wir hier `Integer` und nicht `int` verwenden, werden wir im Abschnitt 4.7.1 betrachten.

Generische Klassen

Da wir in dem vorherigen Abschnitt gesehen haben, wofür Generics an sich gut sind, werden wir uns in diesem Abschnitt anschauen, wie wir selber generische Klassen erstellen können und uns anschauen, wie wir generische Klassen nutzen können.

Wir schauen uns hierzu eine mögliche Implementierung der oben beschriebenen Liste an, welche noch keine Generics nutzt (die Implementierung leitet einfach alle Aufrufe an eine andere, nicht von uns implementierte List weiter):

```
public class List {
    private DelegateList delegateList = new DelegateList();

    public void add(Object element) {
        delegateList.add(element);
    }

    public Object get(int index) {
        return delegateList.get(index);
    }

    public int size() {
        return delegateList.size();
    }
}
```

Abbildung 4.17: Java: Implementation von `List`

Diese Klasse können wir nur nutzen, wie es in Codebeispiel 4.15 gelistet ist. Um die Klasse nun generisch Nutzbar zu machen, müssen wir sogenannte *Typparameter* einführen. Diese werden in spitzen Klammern (<, >) hinter den Klassennamen geschrieben (ähnlich wie bei der Nutzung, nur werden die keine expliziten Klassen genannt, sondern Platzhalter verwendet). Als Name für Typparameter wird meist ein einzelner großer Buchstabe verwendet, um auf den ersten Blick ersichtlich zu machen, dass es sich um einen generischen Typ und nicht um eine existierende Klasse handelt.

Die Klasse sieht, unter Nutzung von Typparametern, nun folgendermaßen aus (der Einfachheit halber nehmen wir an, die Klasse `DelegateList` sei auch generisch):

```
public class List<T> {
    private DelegateList<T> delegateList = new DelegateList<T>();

    public void add(T element) {
        delegateList.add(element);
    }

    public T get(int index) {
        return delegateList.get(index);
    }

    public int size() {
        return delegateList.size();
    }
}
```

Abbildung 4.18: Java: Implementation von `List<T>`

Die Klasse können wir nun deutlich einfacher nutzen, wie es bereits in Codebeispiel 4.22 vorgestellt wurden. Es ist nun möglich, der Klasse beliebige Typen zu übergeben. Wie wir dies weiter Einschränken können, werden wir im Abschnitt 4.7.1 über die Einschränkung der Typparameter behandeln.

Den Typparameter `T` können wir nun überall in der Klasse verwenden, Instanzen des Typs verhalten sich wie Instanzen von `Object`, das heißt es können annähernd keine Operationen auf den Instanzen durchgeführt werden.

Warnung: Die Typparameter einer Klasse können nicht in dem statischen Kontext einer Klasse verwendet werden! Das heißt, die Typparameter können nicht im `static`-Block oder in statischen Methoden verwendet werden.

Info: Typische Namen für die Typparameter sind:

`T` für beliebige Typen.

`K` für Typen, die einen Key darstellen.

`V` für Typen, die einen Wert (Value) darstellen.

Generische Methoden

In vielen Fällen ist es nicht nötig, die gesamte Klasse zu Parametrisieren, sondern es ist möglich oder auch nötig, nur einzelne Methoden generisch zu Halten. Ein gutes Beispiel hierfür sind Methoden in Utility-Klassen, bei denen gar keine Instanz der Klasse erstellt wird und damit die Klassentypparameter nicht nutzbar sind.

Betrachten wir hierzu eine Methode, welche die Textrepräsentation eines jeden Objektes in einer Liste aneinanderhängt.

Wir betrachten zuerst die nicht-generische Methode, welche nur Strings unterstützt:

```
1 public static String concatenate(List<String> list) {
2     String result = "";
3     for (int i = 0; i < list.size(); i++) {
4         result += list.get(i);
5     }
6     return result;
7 }
```

Abbildung 4.19: Java: Generics an Methoden: Motivation

Nun sollen auch andere Typen genutzt werden können, weshalb wir hier Typparameter einführen. Der Typparameter wird wie vorher auch in Spitze klammern gefasst und direkt vor dem Rückgabetyp plaziert. Hierdurch ist es auch möglich, den Typparameter bereits im Rückgabetyp zu verwenden, wie wir später sehen werden. Des weiteren gilt wie bei Klassen, dass für Typparameter ein einzelner, großer Buchstabe verwendet werden sollte.

Schauen wir uns nun also die obige Methode unter Verwendung von Typparametern an:

```java
public static <T> String concatenate(List<T> list) {
    String result = "";
    for (int i = 0; i < list.size(); i++) {
        // Da wir nun nicht mehr direkt auf Strings arbeiten, muessen wir
           uns erst die
        // Textrepraesentation der Elemente holen.
        result += list.get(i).toString();
    }
    return result;
}
```

Abbildung 4.20: Java: Generics an Methoden

Nun ist es also möglich, die obige Methode mit jedem Typ aufzurufen.

Primitive Typen und Generics

Da Generics nur Klassentypen entgegen nehmen können, ist es somit nicht möglich, primitive Datentypen in Generics zu verwenden (also `int`, `float`, ...). Zur Abhilfe dieses Problems gibt es sogenannte Wrapper-Klassen, welche den Wert einer primitiven Variable halten und mit Generics genutzt werden können. Namentlich sind diese:

Wrapper-Klasse	Primitiver Typ
`Byte`	`byte`
`Short`	`short`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`
`Character`	`char`
`Boolean`	`boolean`

Tabelle 4.5: Java: Wrapper-Klassen (im Package `java.lang`)

Alle Wrapper-Klassen haben unter anderem die folgenden Methoden:

`valueOf(...)` Erstellt eine Instanz der Wrapper-Klasse mit dem übergebenen primitiven Wert.

`xxxValue()` Gibt den gespeicherten primitiven Wert zurück. „`xxx`“ muss dabei durch den konkreten primitiven Typ ersetzt werden.

Schauen wir uns unter diesen Voraussetzungen folgenden Code an, der den Durchschnitt zweier benachbarten Zahlen (gleitender Durchschnitt) berechnet:

```java
public List<Double> floatingAverage(List<Integer> list) {
    List<Double> result = new ArrayList<Double>();
    for (int i = 1; i < list.size(); i++) {
        int previous = list.get(i - 1).intValue();
        int current = list.get(i).intValue();
        double average = (previous + current) / 2.0;
        result.add(Double.valueOf(average));
    }
    return result;
}
```

Abbildung 4.21: Java: Generics und primitive Datentypen

Wir sehen, dass sich ein extremer Overhead an Code ergibt, welcher nur zum Konvertieren der Wrapper-Klassen in primitive dient. Da dies sehr nervig sein kann, wurde in Java 5 das sogenannte Autoboxing eingeführt, welches wir nun betrachten.

Autoboxing

Der Name „Autoboxing“ bezeichnet das automatische Konvertieren von primitiven in Wrapper-Klassen und anders herum („Unboxing“). Der primitive Wert wird in eine Wrapper-Klasse „geboxed“.
Die Konvertierung findet genau dann statt, wenn es benötigt wird. Führen wie also eine Rechnung mit Wrapper-Klassen aus, so wird im Hintergrund der primitive Wert geunboxed und mit diesem gerechnet. Weisen wir das Ergebnis einer Wrapper-Klasse zu, so wird das Ergebnis wieder geboxed.

> **Warnung:** Autoboxing kann zu unerwarteten Fehlern führen! Wird eine `null`-Instanz einer Wrapper-Klasse geunboxed, so führt dies zu einer `NullPointerException`!

Schauen wir uns also obigen Beispiel erneut an, diesmal aber mit Autoboxing:

```java
public List<Double> floatingAverage(List<Integer> list) {
    List<Double> result = new ArrayList<Double>();
    for (int i = 1; i < list.size(); i++) {
        int previous = list.get(i - 1);              // Autounboxing
        int current = list.get(i);                   // Autounboxing
        double average = (previous + current) / 2.0;
        result.add(average);                         // Autoboxing
    }
    return result;
}
```

Abbildung 4.22: Java: Autoboxing

Vererbung

In Codebeispiel 4.22 verwenden wir `T`, um Listen jedes Typs annehmen zu können. Wieso können wir hier nicht einfach wie üblich `Object` nehmen? Der Grund ist: Generics unterstützen keine Vererbung, das heißt es wäre nicht möglich, Listen vom Typ `String` zu übergeben, wenn die Methode einen Parameter `List<Object>` erwartet.

Es ist dennoch möglich, ein solches Verhalten hervorzubringen, wie wir im Abschnitt 4.7.1 sehen werden.

Einschränkung der Typparameter und Wildcards

Wie bereits erwähnt ist es nicht möglich, einer Methode mit Parameter `List<Object>` eine Instanz einer Liste `List<String>` zu übergeben.

Wildcard

Eine Möglichkeit ist es wie in 4.20 einen uneingeschränkten Typparameter zu definieren. Eine andere Möglichkeit ist, die Wildcard `?` zu verwenden, welche alle Typen akzeptiert. Somit kann die Methode in 4.20 zu folgender, äquivalenter, Methode umgeschrieben werden:

```
1 public static String concatenate(List<?> list) {
2     String result = "";
3     for (int i = 0; i < list.size(); i++) {
4         result += list.get(i).toString();
5     }
6     return result;
7 }
```

Abbildung 4.23: Java: Generics Wildcard

Das „?“ nimmt dann alle Typen an.

Einschränkungen der Typparameter

Oftmals ist es jedoch nötig, nicht einfach alle Typen zu akzeptieren, sondern die Parameter weiter einzuschränken. Hierzu gibt es folgende essentielle Operatoren:

`extends` Akzeptiert alle Typen, die eine bestimmte Klasse sind oder eine Unterklasse von dieser sind.

`super` Akzeptiert alle Typen, die eine bestimmte Klasse sind oder eine Oberklasse von dieser sind.

Wir werden gleich noch einige Beispiele betrachten, vorher sei jedoch noch gesagt, dass sich die Einschränkungen sowohl auf Typparameter selbst (also auf die Definition von `T`, `K`, ...) anwenden lassen als auch direkt auf Wildcards (also `?`).

`extends`: Erweitern wir unsere Methoden zum Aufsummieren von Zahlen so, dass die Methode alle Zahlentypen akzeptiert und Wildcards nutzt (Achtung: In dieser Implementierung werden einfach alle Nachkommastellen abgeschnitten, sollte eine Fließkommazahl übergeben werden!):

```java
public long sum(List<? extends Number> list) {
    long result = 0;
    for (int i = 0; i < list.size(); i++) {
        result += list.get(i).longValue();
    }
    return result;
}

// Erfolgreiche Aufrufe.
List<Integer> integerList = new ArrayList<Integer>();
sum(integerList);

List<Long>    longList    = new ArrayList<Long>();
sum(longList);

List<Float>   floatList   = new ArrayList<Float>();
sum(floatList);


// Fehlschlagende Aufrufe.
List<String> stringList = new ArrayList<String>();
sum(stringList);
```

Abbildung 4.24: Java: Generic `extends`

Die Methode kann nun mit allen Unterklassen von `Number` aufgerufen werden. Eine Auswahl dieser ist auch im Codebeispiel gegeben.

super: Zur Nutzung von `super` schauen wir uns folgendes Beispiel an, welches die Elemente einer Liste `src` in eine Liste `dest` kopiert und dabei Typsicher verfährt:

```java
public static <T> void copy(List<? extends T> src, List<? super T> dest) {
    for (int i = 0; i < src.size(); i++) {
        dest.add(a.get(i));
    }
}

// Erfolgreiche Aufrufe.
List<String>           src  = new ArrayList<String>();
List<CharSequence> dest = new ArrayList<CharSequence>();
copy(src, dest);


// Fehlschlagende Aufrufe.
List<CharSequence> src  = new ArrayList<CharSequence>();
List<String>           dest = new ArrayList<String>();
copy(src, dest);
```

Abbildung 4.25: Java: Generics `super`

Der zweite Aufruf von `copy(..)` schlägt hierbei fehl, denn weder ist `CharSequence` eine Unterklasse von `String`, noch ist `String` eine Oberklasse von `CharSequence`. Das Gegenteil ist hingegen der Fall, womit der erste Aufruf erfolgreich ist.

Typlöschung (Type Erasure)

In diesem Abschnitt werden wir uns mit der Implementierung von Generics im Compiler beschäftigen, welche zu einigen interessanten Eigenschaften von Generics führt.

Implementiert werden Generics durch Typlöschung, das heißt, die Generics sind zur Laufzeit nicht mehr vorhanden. Stattdessen werden die Aufrufe, bei denen die Typen durch Generics gesteuert werden, durch den höchstmöglichen Typ (meist `Object` wenn kein `extends` genutzt wurde) und Downcasts ersetzt, das heißt, der Code wird zu einem äquivalenten Code umgebaut.

Beispiel

Betrachten wir folgende Methode, welche das maximale Element einer List findet:

```
1  public <T extends Comparable<? super T>> T max(List<T> list) {
2      T max = null;
3      for (int i = 0; i < list.size(); i++) {
4          if (max == null || list.get(i).compareTo(max) > 0) {
5              max = list.get(i);
6          }
7      }
8      return max;
9  }
10
11 // Aufruf.
12 List<Integer> list = new ArrayList<Integer>();
13 list.add(1); list.add(2); list.add(3);
14 Integer max = max(list);                    // max == 3
```

Abbildung 4.26: Java: Generics vor Typlöschung

von dem Compiler durch folgenden Code ersetzt (das Autoboxing wird hier außer Acht gelassen):

```
public Comparable max(List list) {
    Comparable max = null;
    for (int i = 0; i < list.size(); i++) {
        if (max == null || ((Comparable) list.get(i)).compareTo(max) > 0) {
            max = (Comparable) list.get(i);
        }
    }
    return max;
}

// Aufruf.
List list = new ArrayList();
list.add(1); list.add(2); list.add(3);
Integer max = (Integer) max(list);      // max == 3
```

Abbildung 4.27: Java: Generics nach Typlöschung

Limitierungen

Primär durch die Typlöschung verursacht gibt es einige unerwartete Effekte, wenn man mit Generics arbeitet. Wir werden nun die zwei wichtigsten betrachten:

- Es ist nicht (mit vertretbarem Aufwand) möglich, ein Array eines Typparameters erstellen. Dies lässt sich durch die Typlöschung einfach begründen, wenn Code der Form `new T[10]` würde zu `new Object[10]` umgebaut werden, was nicht zu `String[]` oder welchen Typ auch immer gecastet werden kann.
- Es ist außerdem nicht möglich, neue Instanzen eines Typparameters zu erstellen. Dies ist lässt sich auf zwei Faktoren zurück führen:
  1. Es ist zur Compilezeit der generischen Klassen nicht bekannt, welche Parameter der Konstruktor hat.
  2. Durch die Typlöschung würde Code der Form `new T()` zu `new Object()` umgebaut werden, was absolut nutzlos ist und auch nicht weiter gecastet werden kann.

Literaturverzeichnis

[BJ05] Andreas M. Böhm and Bettina Jungkunz. *Grundkurs IT-Berufe: die technischen Grundlagen verstehen und anwenden können*. Vieweg, 2005.