Programmierung 1

Vorlesungsskript 1.2
HS 2023

PD Dr. Kaspar Riesen
Institut für Informatik
Universität Bern

kaspar.riesen@unibe.ch

Inhaltsverzeichnis

0 Administration und Organisation **1**

0.1 Einbettung und Inhalte . . . 1

0.2 Namen und Adressen . . . 1

0.3 Material und Literatur . . . 2

0.4 Zeiten und Orte . . . 3

0.5 Selbststudium und Übungen . . . 4

0.6 Prüfung . . . 5

1 Einführung und Motivation **6**

1.1 Weshalb Java? . . . 6

1.2 Ein Erstes Java Programm . . . 9

1.2.1 Die Methode `main` . . . 10

1.2.2 Kommentare . . . 12

1.2.3 Bezeichner und Reservierte Wörter . . . 13

1.3 Programmentwicklung . . . 15

1.3.1 Kompilieren und Interpretieren . . . 15

1.3.2 Entwicklungsumgebungen . . . 16

1.3.3 Programmierfehler . . . 18

1.4 Objektorientiertes Programmieren . . . 20

1.4.1 Variablen und Methoden . . . 21

1.4.2 Objekt vs. Klasse . . . 21

1.4.3 Kapselung und Vererbung . . . 23

2 Daten und Ausdrücke **26**

2.1 Zeichenketten . . . 26

2.1.1 Die Methoden `print` und `println` . . . 26

2.1.2 Konkatenation von Zeichenketten . . . 28

2.1.3 Escape Sequenzen . . . 30

2.2 Variablen und Konstanten . . . 32

2.3 Primitive Datentypen . . . 36
2.3.1 Datenkonvertierung . . . 37
2.4 Eingaben vom Benutzer Entgegennehmen – *Teaser* zur Klasse `Scanner` . . 39
2.5 Arithmetische Ausdrücke . . . 40
2.6 Boolesche Ausdrücke und die `if`-Anweisung . . . 45
2.6.1 Boolesche Ausdrücke . . . 47
2.6.2 Die `if-else` Anweisung . . . 48
2.6.3 Blockanweisungen . . . 50

3 Klassen des Java API Verwenden (Teil 1) 51
3.1 Objekte aus Klassen Instanziieren . . . 51
3.1.1 Methoden von Instanziierten Objekten Verwenden . . . 55
3.2 Das Java API . . . 56
3.3 Die Klasse `String` . . . 58
3.4 Die Klasse `Scanner` . . . 61
3.5 Die Klasse `Random` . . . 64
3.6 Die Klasse `DecimalFormat` . . . 66
3.7 Kopieren von Objekten – Aliase . . . 69

4 Eigene Klassen Programmieren (Teil 1) 73
4.1 Einführung . . . 73
4.2 Aufbau einer Java Klasse . . . 74
4.3 Instanzvariablen und Sichtbarkeitsmodifikatoren . . . 79
4.4 Methoden Aufrufen und dem Kontrollfluss Folgen . . . 83
4.5 Der Methodenkopf . . . 84
4.6 Konstruktoren . . . 85
4.7 Parameter für Methoden . . . 87
4.8 Die `return` Anweisung . . . 89
4.9 *Getter* und *Setter* . . . 91
4.10 UML Klassendiagramme . . . 94
4.11 Ein Weiteres Beispiel einer Eigenen Klasse . . . 95

5 Klassen des Java API Verwenden (Teil 2) 100
5.1 Die Klasse `Math` . . . 100
5.2 Wrapper Klassen . . . 103
5.3 Die `while`-Anweisung . . . 106
5.3.1 Endlosschleifen . . . 110
5.3.2 Verschachtelte Schleifen . . . 110
5.4 Sammlung von Objekten . . . 113

5.4.1 Exkurs: Generics . . . 114
5.4.2 Die Klasse `ArrayList` . . . 116
5.4.3 Iteratoren . . . 121

6 Bedingungen und Schleifen **122**
6.1 Die `switch`-Anweisung . . . 123
6.2 Der *Conditional Operator* . . . 128
6.3 Die `do`-Anweisung . . . 129
6.4 Die `for`-Anweisung . . . 130
6.4.1 Die `for-each` Schleife . . . 134
6.5 Daten Vergleichen . . . 135
6.5.1 Vergleich von Gleitkommazahlen . . . 135
6.5.2 Vergleich von Zeichen . . . 136
6.5.3 Vergleich von Zeichenketten . . . 137
6.5.4 Vergleich von Objekten . . . 139

7 Arrays **142**
7.1 Arrays Deklarieren und Verwenden . . . 142
7.1.1 Grenzen Finden . . . 146
7.2 Arrays für Objekte . . . 148
7.3 Programmparameter . . . 155
7.4 Variable Länge von Parameterlisten . . . 157
7.5 Zweidimensionale Arrays . . . 160
7.6 Vergleich von Arrays und `ArrayList` . . . 163

8 Eigene Klassen Programmieren (Teil 2) **165**
8.1 Klassen und Objekte Identifizieren . . . 166
8.2 Enum-Typen . . . 167
8.3 Klassen mit Statischen Variablen/Methoden . . . 172
8.3.1 Statische Variablen . . . 172
8.3.2 Statische Methoden . . . 174
8.4 Beziehungen zwischen Klassen . . . 178
8.4.1 Abhängigkeit . . . 178
8.4.2 Aggregation . . . 182

9 Schnittstellen und Vererbung **186**
9.1 Schnittstellen . . . 186
9.1.1 Schnittstellen des Java APIs . . . 192
9.2 Vererbung . . . 195

9.2.1 Der Konstruktor einer Subklasse 199
9.2.2 Überschreiben von Geerbten Methoden 200
9.2.3 Klassenhierarchien 202
9.2.4 Polymorphismus 208

10 Algorithmen und Methoden (Teil 1) 211
10.1 Methoden Planen 211
10.1.1 Teilen-und-Beherrschen 212
10.1.2 Parameter an Methoden Übergeben 216
10.1.3 Methoden Überladen 219
10.2 Algorithmen in Methoden Umsetzen 222
10.2.1 Mit Algorithmen Sortieren 222
10.2.2 Mit Algorithmen Suchen 225
10.3 Testen von Algorithmen 229

11 Sammlungen von Daten – Das *Collection-Framework* 231
11.1 Trennung von Schnittstelle und Implementierung 232
11.1.1 Listen auf Basis von Arrays 232
11.1.2 Listen auf Basis von Verkettung 234
11.1.3 Abstrakte Datentypen (ADT) 240
11.1.4 Weitere Implementierungsideen für Listen 241
11.2 Das *Collection-Framework* von Java 243
11.2.1 Listen: Die Klassen `ArrayList` und `LinkedList` 245
11.2.2 Stapel: Die Klasse `Stack` 246
11.2.3 Schlangen: Die Schnittstellen `Queue` und `Dequeue` 248
11.2.4 Algorithmen im *Collection-Framework* 250

12 Algorithmen und Methoden (Teil 2) 252
12.1 Rekursive Denkweise 252
12.1.1 Unendliche Rekursion 253
12.1.2 Rekursion in der Mathematik 254
12.2 Rekursives Programmieren 254
12.2.1 Rekursion vs. Iteration 257
12.2.2 Direkte vs. Indirekte Rekursion 258
12.3 Rekursive Algorithmen 259
12.3.1 Rekursiv Sortieren 259
12.3.2 Ein Labyrinth Durchqueren 265

13 Laufzeitfehler **271**
13.1 Nicht Behandeln einer `Exception` 272
13.2 Die `try-catch` Anweisung 274
13.3 Weitergeben einer `Exception` 280
13.4 Eine Eigene `Exception` Programmieren und Werfen 285
13.5 Ein- und Ausgabefehler 287

Kapitel 0

Administration und Organisation

0.1 Einbettung und Inhalte

Programmierung 1 (*P1*) ist eine Vorlesung im *Bachelorstudiengang Informatik* (1. Semester; 5 ECTS) mit folgenden Schwerpunkten:

- Einführung in die Programmierung
- Syntax der Programmiersprache Java
- Prinzipien der objektorientierten Programmierung
  - Klassen
  - Objekte
  - Kapselung
  - Vererbung und Schnittstellen
  - Polymorphismus
- Problemlösung mit Programmen
- Programme entwerfen, implementieren und testen

0.2 Namen und Adressen

- **Adresse**: Institut für Informatik (INF), Neubrückstrasse 10 (siehe Abb. 1) `https://www.inf.unibe.ch`
- **Dozent**: Kaspar Riesen; `kaspar.riesen@unibe.ch`
- **Übungsleitung**:

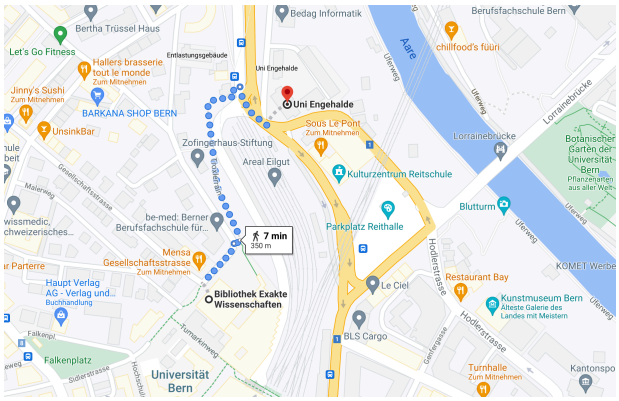

Abbildung 1: Lageplan des Institutes.

  - Mathias Fuchs `mathias.fuchs@unibe.ch`
  - Cedric Wolfisberg `cedric.wolfisberg@students.unibe.ch`
  - Cedric Sigrist `cedric.sigrist@students.unibe.ch`
  - Raphael Waller `raphael.waller@students.unibe.ch`

- **Workstations**: ExWi A94, A95
- **Systemadministration (LINUX, UNIX)**: Peppo Brambilla; `sysop.inf@unibe.ch`
- **Institutssekretariat**: Priska Grunder; `priska.grunder@unibe.ch`
- **Prüfungssekretariat**: Bettina Choffat; `studienberatung.inf@unibe.ch`

0.3 Material und Literatur

Wir unterhalten eine Kurs-Homepage auf ILIAS: `http://www.ilias.unibe.ch`

- Diese Seite enthält sämtliche Materialien (Skript, Übungen, Musterlösungen, Quellcode der Beispiele, etc.) und detaillierte Informationen über die Vorlesung, den Übungsbetrieb und die Prüfung.
- Wir unterhalten auf der Seite auch ein Forum für Fragen oder Anliegen.
- Bitte besuchen Sie diese Seite regelmässig, um über die Vorlesung und den Übungsbetrieb auf dem Laufenden zu bleiben.

Es existieren Dutzende von Lehrbüchern zum Thema *Programmieren mit Java*, zum Beispiel:

- Ch. Ullenboom: Java ist auch eine Insel. 15. Auflage, Rheinwerk Verlag
- J. Goll, C. Heinisch: Java als erste Programmiersprache. 8. Auflage, Springer-Vieweg
- K. Arnold, J. Gosling, D. Holmes: Die Programmiersprache Java (deutsche Übersetzung), Addison Wesley
- Ä. Plüss: Java - exemplarisch. Oldenbourg Wissenschaftsverlag
- B. Evans, D. Flanagan. Java in a Nutshell, 7th Edition, O'Reilly
- u.v.a.

Die Vorlesung basiert hauptsächlich auf folgendem Buch:

- John Lewis, William Loftus: Java Software Solutions. 9th Edition, Pearson, Addison Wesley

Die Anschaffung eines Buches ist *nicht* Pflicht.

0.4 Zeiten und Orte

Die Präsenzveranstaltung setzt sich aus drei Teilen zusammen:

- **Vorlesung (Freitag, 13 bis max. 15; ExWi A006)**: In der Vorlesung wird der im jeweiligen Kapitel und *Video-Lecture* eingeführte Stoff anhand von Beispielen und Aufgaben gefestigt und vertieft. Zudem besteht die Möglichkeit, Fragen zur *Vorlesung* direkt mit dem Dozenten zu besprechen.
- **Übungsstunde (direkt im Anschluss an die Vorlesung; ExWi A006)**: In der Übungsstunde werden die Übungen vor- und nachbesprochen und es besteht die Möglichkeit, Fragen zu den *Übungen* direkt mit den Assistierenden zu besprechen.
- **Praktikum (Mittwoch, 12 bis 14; ExWi A097)**: Im Praktikum werden keine neuen Inhalte präsentiert. Der Besuch ist freiwillig. Es besteht die Möglichkeit, Fragen zu stellen und die Aufgaben mit Hilfe der Assistierenden zu lösen[1].

Wichtiger Hinweis: Die Vorlesung bzw. die Übungstunde finden in einigen Wochen nicht statt. In Abb. 2 finden Sie einen detaillierten Zeitplan für das gesamte Semester.

[1] Das Praktikum startet erst in Woche 2.

Semesterplan Programmieren 1 HS 2023					
Zeitpunkte		Vorlesungsbetrieb		Übungsbetrieb	
Woche	Datum	Vorbereitung für die Vorlesung	Inhalte der Vorlesung	Inhalte der Übungsstunde	Abgaben
1	22.09.23	Kapitel 0 und Videolecture 1	Kapitel 1	Allgemeine Infos zur Abgabe, Bewertung und dem Testat Infos zu den nötigen Installationen	
2	29.09.23	Kapitel 2 und Videolecture 2	Keine Vorlesung: Übungsstunde beginnt um 13.15 Uhr	Vorbesprechung Serie 1	
3	06.10.23	Kapitel 3 und Videolecture 3	Weitere Erläuterungen, Beispiele und Aufgaben zum jeweiligen Kapitel der Woche	Nachbesprechung Serie 1 Vorbesprechung Serie 2	Serie 1
4	13.10.23	Kapitel 4 und Videolecture 4		Nachbesprechung Serie 2 Vorbesprechung Serie 3	Serie 2
5	20.10.23	Kapitel 5 und Videolecture 5		Keine Übungsstunde	
6	27.10.23	Kapitel 6 und Videolecture 6		Nachbesprechung Serie 3 Vorbesprechung Serie 4	Serie 3
7	03.11.23	Kapitel 7 und Videolecture 7		Nachbesprechung Serie 4 Vorbesprechung Serie 5	Serie 4
8	10.11.23	Kapitel 8 und Videolecture 8		Keine Übungsstunde	
9	17.11.23	Kapitel 9 und Videolecture 9		Nachbesprechung Serie 5 Vorbesprechung Serie 6	Serie 5
10	24.11.23	Kapitel 10 und Videolecture 10		Keine Übungsstunde	
11	01.12.23	Kapitel 11 und Videolecture 11		Nachbesprechung Serie 6 Vorbesprechung Serie 7	Serie 6
12	08.12.23	Kapitel 12 und Videolecture 12		Nachbesprechung Serie 7 Vorbesprechung Serie 8	Serie 7
13	15.12.23	Kapitel 13 und Videolecture 13		Nachbesprechung Serie 8 Vorbesprechung Serie Bonus	Serie 8
14	22.12.23	Keine Vorbereitung	Repetition und Fragestunde	Nachbesprechung Serie Bonus	Serie Bonus

Abbildung 2: Semesterplan (ACHTUNG: Aktuellste Version ist immer auf ILIAS).

0.5 Selbststudium und Übungen

Das Selbststudium umfasst zwei Teile:

- Wir erwarten, dass Sie gut vorbereitet in der Vorlesung erscheinen. Das Selbststudium umfasst insbesondere, dass Sie das entsprechende Kapitel aus dem Skript *vor* dem Unterricht studieren, sich ggf. die zugehörige *Video-Lecture* ansehen und sich ggf. Fragen notieren.

- In den Übungsstunden werden 8 + 1 Übungen vergeben, welche Sie in Zweiergruppen bis zum angegebenen Termin bearbeiten und einreichen. Für einige Übungen haben Sie eine Woche Zeit, für andere zwei Wochen. Das Abgeben der Übungen ist Voraussetzung für die Zulassung zur Prüfung (*Testat)*. Details zum Übungsbetrieb und zum Testat werden in der Übungsstunde bekannt gegeben.

Wichtiger Hinweis: Programmieren lernen hat eine gewisse Ähnlichkeit mit dem Erlernen eines Musikinstrumentes. Wenn Sie mehrere Bücher zum Thema Klavier oder Saxophon gelesen haben und/oder wenn Sie stundenlang Menschen beobachtet haben, die ein Instrument spielen, können Sie dann dieses Instrument spielen? Wahrscheinlich nicht. Genauso verhält es sich beim Programmieren – dieses Skript gründlich zu studieren und/oder einem Kollegen oder einer Kollegin über die Schultern zu schauen, während er oder sie programmiert, wird Ihnen bestenfalls

ermöglichen, kleinere Aufgaben oder Übungen zu meistern. Aber richtig zu programmieren kann man nur lernen, indem man selber viel und vor allem regelmässig programmiert.

Im vorliegenden Skript wird die Theorie mit zahlreichen, illustrierenden Beispielen unterstützt. Es lohnt sich, diese Beispiele nicht nur zu studieren, sondern diese jeweils selber zu programmieren. Zudem gibt es für die meisten Kapitel Übungsserien mit Theorieaufgaben und Java Übungen. Wenn Sie vorhaben, das Programmieren tatsächlich zu erlernen, dann müssen Sie *alle* Aufgaben und *alle* Java Übungen sorgfältig und falls nötig *mehrfach* durcharbeiten.

0.6 Prüfung

- **Zulassung:** Testat (siehe Informationen zum Übungsbetrieb)
- **Termin:** Mitte Januar 2024 (definitiver Termin wird auf KSL publiziert)
- **Ort:** Gebäude ExWi A6 und ggf. weitere Räume
- **An- und Abmeldung:** Eine An- bzw. Abmeldung über KSL ist erforderlich.
- **Voraussichtlicher Ablauf:** Schriftliche Prüfung während 60 Minuten (die Prüfung findet *nicht* am Computer statt). Alle Kandidat*innen müssen spätestens 15 Minuten vor Prüfungsbeginn im zugewiesenen Hörsaal anwesend sein. Bitte bringen Sie Ihre *Unicard* oder einen anderen Ausweis, A4-Schreibpapier und Schreibmaterial mit.
- **Prüfungsstoff:** Inhalte der gesamten Vorlesung und aller Übungsstunden
- **Erlaubte Hilfsmittel:** Ausgedrucktes Skript, Übungen, Unterlagen der Vorlesung, eigene Notizen
- **Nicht erlaubt:** Bücher, Notebook oder andere elektronische Geräte (Smartphone, Tablet, Taschenrechner, etc.)
- **Wiederholungsprüfung:** Die Wiederholungsprüfung findet *während* des Frühlingssemesters statt. Das Datum und die Modalitäten werden auf ILIAS kommuniziert.

Kapitel 1

Einführung und Motivation

1.1 Weshalb Java?

Programmieren hat viel mit Lösen von Problemen zu tun. Das allgemeine Ziel der Programmierung ist es, zu gegebenen Problemen ein oder mehrere Programme zu entwickeln, die auf Computern ausführbar sind und dabei die Probleme korrekt, vollständig und möglichst effizient lösen. Die Komplexität der Probleme, die mit Programmen gelöst werden sollen, kann dabei stark variieren: Von sehr einfachen Problemen wie z.B. der Addition zweier Zahlen, bis hin zu sehr komplexen Problemen, wie z.B. der Steuerung von Passagierflugzeugen.

Möchte man ein Programm schreiben, das ein Computer ausführen kann, so muss dies in einer Sprache geschehen, welche eine Maschine verstehen kann – wir benötigen also eine *Programmiersprache*. Programmiersprachen geben bestimmte Wörter, Symbole und Regeln vor, die genau bestimmen, wie eine Programmiererin[1] gültige *Programmieranweisungen* (engl. *Programming Statements*) definieren kann. Diese Anweisungen werden dann während der Ausführung des Programmes durch den Computer in einer von der Programmiererin festgelegten Reihenfolge ausgeführt.

Es existieren Dutzende von Programmiersprachen, die z.T. für unterschiedlichste Zwecke definiert worden sind und die unterschiedliche *Paradigmen* verfolgen. Man unterscheidet u.a. folgende Paradigmen:

- *Imperative Programmierung*: Das Programm besteht aus einer Folge von Be-

[1] Um den Lesefluss nicht zu beeinträchtigen, nutzen wir in diesem Skript meist nur die weibliche Form zur Bezeichnung eines Menschen, der Computerprogramme schreibt: *die Programmiererin*. Die männliche Form – also *der Programmierer* – ist aber immer mitgemeint. Umgekehrt nutzen wir meist nur die männliche Form zur Bezeichnung eines Menschen, der mit einem Computerprogramm interagiert: *der Benutzer* – und auch hier gilt: Die weibliche Form ist immer mitgemeint.

fehlen, die vorgeben, in welcher Reihenfolge was vom Computer getan werden soll (Beispiele: *Pascal*, *Fortran*, *Ada*, *C*, etc.)

- *Prozedurale Programmierung*: Die prozedurale Programmierung ergänzt das imperative Paradigma um den Ansatz, ein Programm in überschaubare Teile zu zerlegen. Je nach Programmiersprache werden diese Teile *Prozedur* oder *Funktion* genannt (praktisch alle aktuellen imperativen Programmiersprachen – wie zum Beispiel *C* – verwenden den prozeduralen Ansatz).

- *Deklarative Programmierung*: Im Gegensatz zum imperativen Paradigma, bei dem ein Lösungsweg programmiert wird, wird in diesem Sprachparadigma nur angegeben, welches das gewünschte Ergebnis ist. Zu den deklarativen Programmiersprachen gehören funktionale Sprachen (bspw. *Haskell*), logische Sprachen (bspw. *Prolog*) oder auch mengen-orientierte Abfragesprachen für Datenbanken (bspw. *SQL*)

- *Objektorientierte Programmierung:* Programme werden durch das Zusammenspiel kooperierender Objekte erstellt. Ein Objekt besitzt bestimmte Attribute (Eigenschaften) und Methoden (Funktionalitäten) und ist in der Lage, von anderen Objekten Nachrichten zu empfangen beziehungsweise Nachrichten an diese zu senden (Beispiele: *Smalltalk*, *C++*, *Eiffel*, *Java*, etc.).

Unterschiedliche Sprachen besitzen jeweils Vor- und Nachteile, die von vielen Faktoren abhängig sind. In diesem Skript werden wir mit Hilfe der Programmiersprache *Java* das Programmieren erlernen (siehe auch Abb. 1.1). Obschon es sehr schwierig bzw. unmöglich ist, zu sagen, welche Sprache die *beste* Programmiersprache ist, darf man behaupten, dass Java einige gewichtige Vorzüge aufweist:

- Java ist **weit verbreitet** und gilt als eine der am schnellsten wachsenden Programmiertechnologien aller Zeiten. 90% der *Fortune 500* Firmen setzen Java ein, es gibt ca. 10 Millionen Java Programmierer und Programmiererinnen und Java läuft auf mehreren Milliarden Geräten weltweit. Gemäss dem *Tiobe-Index* war Java in den letzten 20 Jahren immer in den Top-3 der populärsten Programmiersprachen.

- Java kann **vielfältig** eingesetzt werden (von Programmen für mobile Geräte über Desktop-Anwendungen zu Web- und Unternehmensanwendungen).

- Java ist **plattformunabhängig** (läuft also auf allen gängigen Betriebssystemen).

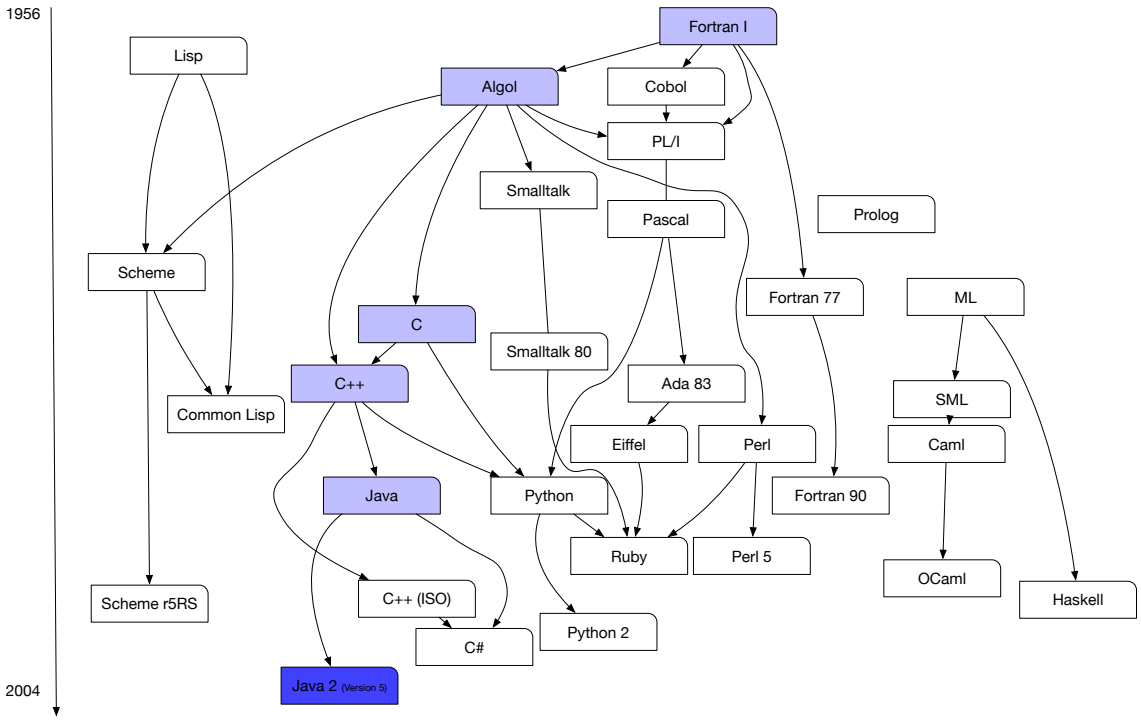

Abbildung 1.1: Vor- und Vorvorgänger von Java.

- Java ist **objektorientiert** (und Objektorientierung ist das aktuell "gültige" Paradigma in der Softwareentwicklung).
- Java wird ständig **weiterentwickelt** und seit der "Erfindung" von Java wurden zahlreiche Änderungen und Erweiterungen an Java vorgenommen. Ursprünglich wurde Java 1995 von *Sun Microsystems* entwickelt – heute gehört Java zu *Oracle*.
- Java erlaubt die **parallele Programmierung** – Teilstücke des Programms können aufgeteilt und nebenläufig ausgeführt werden.
- Java wird von einer umfassenden **Bibliothek** (engl. *Library*) aus zusätzlicher Software begleitet, die wir zur Programmentwicklung verwenden können. Diese bereits erstellten Programme (geschrieben in Java) werden *Java API* oder *Standard Class Library* genannt (API steht für *Application Programmer Interface*).

1.2 Ein Erstes Java Programm

Nachfolgend ist der *Quellcode* (engl. *Sourcecode*) eines ersten, sehr einfachen Java Programmes abgebildet (→ Video K-1.1)[2].

```
public class Quote {

        /**
         * Gibt ein Zitat von Steve Jobs aus
         */
        public static void main(String[] args) {
                System.out.println("Steve Jobs:");
                System.out.println("Es ist besser, ein Pirat zu sein, als der Marine beizutreten.");
        }
}
```

Wird dieses Programm ausgeführt, werden die folgenden zwei Zeilen auf dem Bildschirm[3] eines Computers ausgegeben:

```
Steve Jobs:
Es ist besser, ein Pirat zu sein, als der Marine beizutreten.
```

Dieses Programm ist sehr einfach und noch nicht sehr nützlich. Dennoch illustriert dieses Beispiel die Grundstruktur aller Java Programme schon recht deutlich:

- Alle Java Programme werden mit *Klassen* erstellt. Der Klassenname ist im *Klassenkopf* ersichtlich (nach den zwei Worten `public class`). Diese Klasse heisst `Quote`. Wir können Klassen fast beliebige Namen – sogenannte *Bezeichner* (engl. *Identifier*) – geben. Diese Klasse könnte also z.B. auch `Steve` oder `Zitat` oder `X` heissen (es gibt allerdings ein paar Regeln für Bezeichner, die wir beachten müssen – Details folgen weiter unten).

- Wir werden in dieser Vorlesung jede Klasse in einer separaten Datei speichern – der Dateiname und der Klassenname müssen dabei zwingend übereinstimmen und die Dateiendung muss `.java` sein. Die Datei, welche die Klasse `Quote` enthält, heisst also in diesem Fall `Quote.java`.

- Der *Rumpf* der Klasse `Quote` beginnt mit der öffnenden, geschweiften Klammer { nach dem Klassennamen und endet mit der letzten, schliessenden, geschweiften Klammer }.

[2] Der Quellcode erstreckt sich z.T. über den Rand des Feldes. Das ist Absicht und wird später besprochen.
[3] Etwas genauer: Die Ausgabe der zwei Zeilen erfolgt auf der sogenannten *Konsole* des Computers.

- Im Klassenrumpf können beliebig viele *Methoden* definiert werden. Eine Methode ist eine Gruppierung von Programmieranweisungen, die unter einem bestimmten Bezeichner zusammengefasst werden (dieser Bezeichner steht im *Methodenkopf*). In diesem Beispiel besitzt die Klasse eine einzige Methode, welche `main` heisst und zwei Programmieranweisungen zusammenfasst.
- Wie bei Klassen, werden auch die Methoden mit geschweiften Klammern geöffnet und wieder geschlossen. Innerhalb dieser geschweiften Klammern befindet sich der *Methodenrumpf*.
- Der Rest des Quellcodes sind *Kommentare*. Kommentare beeinflussen das Programm nicht – diese sollen lediglich das Lesen und Verstehen des Quellcodes für Menschen erleichtern.

Zusammenfassend:

- Ein *Programm* setzt sich aus einer oder mehreren Klassen zusammen.
- Eine *Klasse* enthält eine oder mehrere Methoden.
- Eine *Methode* enthält Programmanweisungen.
- Ein *Kommentar* ist auf jeder Ebene möglich (Anweisung, Methode, Klasse).

```
 //  Kommentar zur Klasse
public class MyClass {

          //  Kommentar zur Methode
          public void method() {
                  // Kommentar zur Anweisung
                  System.out.println();
          }
}
```

Hinweis: Leerraum (z.B. Leerschläge oder das Tabulatorzeichen) wird verwendet, um einzelne Wörter und Symbole in Quellcode zu trennen. Extra-Leerraum wird ignoriert. Quellcode sollten zur besseren Lesbarkeit "schön" formatiert werden, wobei insbesondere eine einheitliche Einrückung verwendet werden sollte[4].

1.2.1 Die Methode `main`

Alle Java Programme besitzen eine Methode mit dem Bezeichner `main`. In dieser *Hauptmethode* startet das Programm. Wird ein Programm ausgeführt, so wird –

[4] John F. Woods: *Programmieren Sie immer so, als ob der Typ, der Ihren Quellcode am Ende pflegt, ein gewalttätiger Psychopath ist, der weiss, wo Sie wohnen.*

im Prinzip – jede Programmieranweisungen in der `main` Methode von oben nach unten ausgeführt. Wird das Ende des Rumpfes der `main` Methode erreicht, endet das Programm (man sagt, das Programm *terminiert*). Eine `main` Methode in Java Programmen startet immer mit den Worten

```
public static void
```

Wir werden später auf diese Schlüsselwörter zurückkommen (der Nutzen von `String`, den eckigen Klammern `[]` und `args` wird ebenfalls später erläutert).

Die zwei Zeilen Quellcode in der `main` Methode der Klasse `Quote` rufen eine andere Methode auf, die `println` (ausgesprochen *print line*) heisst. Wir rufen eine Methode auf, wenn wir wollen, dass diese ausgeführt wird. Die `println` Methode ist so programmiert, dass diese die in Klammern mitgegebenen Zeichen auf dem Bildschirm ausgibt (die Zeichen befinden sich hierbei innerhalb doppelter Anführungszeichen). Beachten Sie, dass in Java jede Programmieranweisung mit einem Semikolon `;` abgeschlossen werden muss.

Der Quellcode, der ausgeführt wird, wenn die Methode `println` aufgerufen wird, ist in unserem Programm nicht ersichtlich. Die Methode `println` ist Teil des Java APIs. Jemand anderes hat also die Funktionalität für Ausgaben von Zeichen in Java programmiert und die nötigen Anweisungen unter dem Bezeichner `println` zusammengefasst. Statt selber die Funktionalität in Java zu programmieren, rufen wir also einfach bestehenden Quellcode auf, der die Aufgabe erledigt.

Wenn das obige Programm ausgeführt wird, dann geschieht also folgendes: Die `println` Methode wird zweimal nacheinander mit unterschiedlichen Zeichen aufgerufen. Danach wird das Ende der `main` Methode erreicht und das Programm terminiert.

Methoden können nicht nur beliebig oft, sondern auch in beliebiger Reihenfolge aufgerufen werden.

Beispiel 1 *In folgendem Programm sehen Sie die Klasse* ***`Quotes`****, welche neben der Methode* ***`main`*** *noch zwei weitere Methoden beinhaltet (→ Video K-1.2):*

```
public class Quotes {

        /**
         * Gibt ein Zitat von Bill Gates aus
```

```
         */
        public static void printQuoteOfBill(){
                System.out.println("Bill Gates:");
                System.out.println("Deine Unzufriedenen Kunden sind deine grösste Lernquelle.");
        }

        /**
         * Gibt ein Zitat von Steve Jobs aus
         */
        public static void printQuoteOfSteve(){
                System.out.println("Steve Jobs:");
                System.out.println("Es ist besser, ein Pirat zu sein, als der Marine beizutreten.");
        }

        /**
         * Die main Methode
         */
        public static void main(String[] args) {
                printQuoteOfSteve();
                printQuoteOfBill();
                printQuoteOfSteve();
        }
}
```

Wird dieses Programm ausgeführt, so wird zuerst die Methode `printQuoteOfSteve` *aufgerufen und diese wiederum ruft zweimal die* `println` *Methode auf (zur Erzeugung der Ausgabe des Zitates von Steve Jobs).*

Danach wird die Methode `printQuoteOfBill` *aufgerufen, welche sich um die Ausgabe des Zitates von Bill Gates kümmert. Danach wird nochmals die erste Methode* `printQuoteOfSteve` *aufgerufen und schliesslich wird die letzte Zeile der* `main` *Methode erreicht und das Programm terminiert.*

1.2.2 Kommentare

Typischerweise enthalten Klassen an verschiedenen Stellen Kommentare, welche den Sinn und Zweck des Quellcodes in natürlicher Sprache erläutern. Wenn wir Quellcode lesen, können wir (meistens) verstehen, *was* dieser tut, aber nicht immer, *warum* er dies tut. Kommentare sind für eine Programmiererin die einzige Möglichkeit, ihre Absichten und Überlegungen mitzuteilen.

Kommentare sind essentiell für die Qualität eines Programmes: Quellcode muss oftmals modifiziert oder erweitert werden. Kommentare helfen dabei, den Quellcode (auch Jahre nach dessen Entwicklung) schneller und besser zu verstehen (insbesondere – aber nicht nur – wenn dieser von anderen Menschen geschrieben wurde).

Java erlaubt drei mögliche Formen für Kommentare:

```
// Dieser Kommentar geht bis ans Ende der Zeile

/* Dieser Kommentar kann sich über mehrere Zeilen
   erstrecken und endet erst mit folgenden Endsymbol */

/**
 * Das ist sogenannter javadoc Kommentar, welcher die
 * automatische Generierung von Dokumentationen erlaubt
 */
```

Kommentare sollten prägnant und in ganzen Sätzen formuliert werden, sollten nicht das Offensichtliche kommentieren und müssen eindeutig sein.

1.2.3 Bezeichner und Reservierte Wörter

Die Wörter, die gebraucht werden, um ein Programm zu schreiben, heissen *Bezeichner* (engl *Identifier*). Die elf Bezeichner im Programm `Quote` sind `public`, `class`, `Quote`, `static`, `void`, `main`, `String`, `args`, `System`, `out`, `println` (die Zeichen innerhalb doppelter Anführungszeichen, z.B. `"Steve Jobs"`, sind *keine* Bezeichner sondern *Zeichenketten* – Thema des nächsten Kapitels).

Alle Bezeichner von Java gehören zu einer der folgenden drei Kategorien:

1. Bezeichner, die für bestimmte Zwecke reserviert sind. In unserem Beispiel sind dies `public`, `class`, `static` und `void`. Alle reservierten Wörter erfüllen einen bestimmten Zweck bei der Programmierung und dürfen nicht für andere Zwecke verwendet werden (z.B. dürfen Sie kein reserviertes Wort als Bezeichner für eine Klasse oder Methode wählen). Aktuell sind ca. 50 Wörter in Java reserviert:

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

2. Bezeichner, die sich die Programmiererin *dieses* Programms ausgedacht hat. In unserem Beispiel sind dies `Quote` und `args`. Während die Programmiererin

dieses Programm geschrieben hat, hat sie sich z.B. für `Quote` als Klassennamen entschieden. Der Bezeichner `args` (Kurzform für *Arguments*) wird zwar sehr oft so in der `main` Methode verwendet – die Programmiererin hätte aber auch hierfür einen beliebig anderen Bezeichner wählen können.

3. Bezeichner, die sich eine *andere* Programmiererin ausgedacht hat (`String`, `System`, `out`, `println` und `main` in unserem Beispiel). Diese Bezeichner sind nicht Teil der Programmiersprache Java – diese sind Teil des Java APIs mit Quellcode, den jemand in Java geschrieben hat. Die Autorin dieses Programmcodes hat die Bezeichner gewählt und wir benutzen diese nun.

Ein Bezeichner, den wir selber wählen, kann eine beliebige Kombination aus Buchstaben, Ziffern, dem Unterstrich- und dem Dollarzeichen sein. Bezeichner dürfen dabei beliebig lang sein aber *nicht* mit einer Ziffer beginnen.

Java unterscheidet zwischen Gross- und Kleinschreibung (engl. *case sensitive*). Das heisst, `Minimum`, `minimum` und `MINIMUM` sind in Java verschiedene Bezeichner. Es gibt zahlreiche Konventionen für die Gross- und Kleinschreibung von Bezeichnern (diese Konventionen sind aber *nicht* durch die Programmiersprache Java vorgeschrieben – Konventionen sind Abmachungen zwischen Menschen, die das Lesen und Schreiben von Quellcode vereinfachen sollen).

Wir halten uns in dieser Vorlesung an folgende Konventionen:

- Bezeichner für *Klassen* sind typischerweise Substantive und beginnen mit einem Grossbuchstaben gefolgt von Kleinbuchstaben, wobei der erste Buchstabe jedes internen Wortes gross geschrieben wird (bspw. `StudentActivity` oder `Student`).
- Bezeichner für *Methoden* sind typischerweise Verben, beginnen mit einem Kleinbuchstaben und interne Wörter werden gross geschrieben (bspw. `main`, `sum` oder `findMin`).
- Bezeichner für *Variablen* sollten kurz (z.B. statt `theCurrentProductToBeSent` schreiben wir `product`), aussagekräftig (z.B. `grade` statt `x`) und eindeutig sein (steht der Bezeichner `std` bspw. für `standard` oder `stockDistribution`?).
- Bezeichner für *Konstanten* bestehen ausschliesslich aus Grossbuchstaben und einzelne Wörter sind durch einen Unterstrich getrennt (bspw. `MINIMUM` oder `MAX_CAPACITY`).

Beachten Sie, dass es beim Programmieren weitere Konventionen gibt. Z.B. die Konvention, dass eine Zeile Quellcode nicht mehr als 80 Zeichen umfassen sollte[5].

1.3 Programmentwicklung

1.3.1 Kompilieren und Interpretieren

Maschinensprachen sind Programmiersprachen, in denen die Instruktionen definiert sind, die vom Prozessor eines Computers verstanden und ausgeführt werden können. Jeder Prozessortyp besitzt seine eigene Maschinensprache. Einzelne Anweisungen einer Maschinensprache können nur sehr einfache Aktionen ausführen (z.B. Kopieren eines Wertes in einen Speicherregister oder Vergleichen eines bestimmten Wertes mit anderen Werten). Jeder Befehl der Maschinensprache ist durch einen oder mehrere binäre Zahlenwerte codiert. Eine sinnvolle Folge von solchen binären Zahlencodes bildet den Quellcode, der von einem Computer ausgeführt werden kann. Das Programmieren in Maschinensprache ist natürlich möglich, aber eher schwierig, fehleranfällig und vor allem langwierig.

Um ein Computerprogramm zu programmieren, verwendet man typischerweise eine bestimmte *Hochsprache* (z.B. *Java*, *Ruby* oder *Python*). Das Erstellen und Verstehen von Quellcode, der in einer Hochsprache programmiert ist, ist relativ einfach (auf jeden Fall einfacher als eine sinnvolle Folge von Binärzahlen zu schreiben). Damit aber ein Programm auf einem Computer ausgeführt werden kann, muss dieses zwingend in der entsprechenden Maschinensprache ausgedrückt sein. Das bedeutet, dass Quellcode, welcher in einer anderen Sprache als Maschinensprache verfasst wurde, vor seiner Ausführung zuerst in Maschinensprache übersetzt werden muss.

Kompilierer (engl. *Compiler*) sind Computerprogramme, die einen Quellcode, geschrieben in einer bestimmten Sprache A, in eine Zielsprache B übersetzen können (es braucht unterschiedliche Kompilierer für unterschiedliche Quell- und Zielsprachen). Oftmals ist die Zielsprache die Maschinensprache eines bestimmten Computers. Beachten Sie, dass eine einzige Anweisung in einer Hochsprache (z.B. die Anweisung `println`) vielen – vielleicht Hunderten – von Maschineninstruktionen entsprechen kann. Der Kompilierer erzeugt diese Maschineninstruktionen automatisch aus dem Quellcode.

[5] Diese Grenze von 80 Zeichen ist in obigen Programmen mit einem Rechteck eingezeichnet und wir sehen, dass in beiden Beispielen die Konvention nicht eingehalten wird – wir kommen später auf dieses Problem zurück.

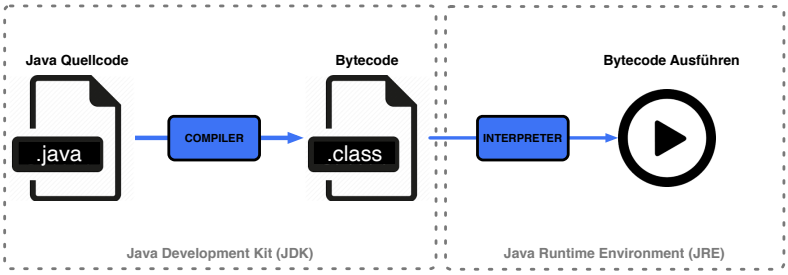

Abbildung 1.2: Java Quellcode wird zuerst mit einem Kompilierer in Bytecode übersetzt. Der Interpreter übersetzt den Bytecode in die passende Maschinensprache und führt die erzeugten Instruktionen gleich aus.

Java unterscheidet sich in der Übersetzung von Quellcode in Maschinensprache von anderen Sprachen. Es findet nämlich keine direkte Übersetzung des Quellcodes in Maschinensprache statt. Der Java Kompilierer übersetzt Java Quellcode in eine spezielle Repräsentation, die *Bytecode* genannt wird. Bytecode entspricht keiner traditionellen Maschinensprache. Mit einem anderen Programm, genannt *Interpreter*, wird der Bytecode schliesslich in Maschinensprache übersetzt. Der Interpreter führt den übersetzten Bytecode – also das Programm – zudem auch gleich auf der Maschine aus (der gesamte Prozess des Kompilierens und des Interpretierens ist in Abb. 8.1 visualisiert).

Dieses etwas komplizierte Vorgehen hat den Vorteil, dass Java *plattformunabhängig* ist. Das heisst, ein kompiliertes Java Programm – also Bytecode – kann auf einer bestimmten Maschine erzeugt worden sein und auf anderen, beliebigen Maschinen ausgeführt werden. Die einzige Einschränkung ist, dass auf der ausführenden Maschine ein Java Interpreter installiert sein muss, der Java Bytecode in die prozessorspezifische Maschinensprache übersetzen und ausführen kann.

Tatsächlich haben die meisten Computer standardmässig einen Java Interpreter installiert – die sogenannte Java Laufzeitumgebung (engl. *Java Runtime Environment (JRE)*). Der Kompilierer für Java Quellcode hingegen muss typischerweise zuerst installiert werden. Dieser befindet sich im sogenannten *Java Development Kit (JDK)*.

1.3.2 Entwicklungsumgebungen

Java Programme können mit jedem beliebigen Texteditor geschrieben werden, der reine Textdateien speichern kann (z.B. `TextEdit`, `NotePad`, `Atom` oder ähnliche Pro-

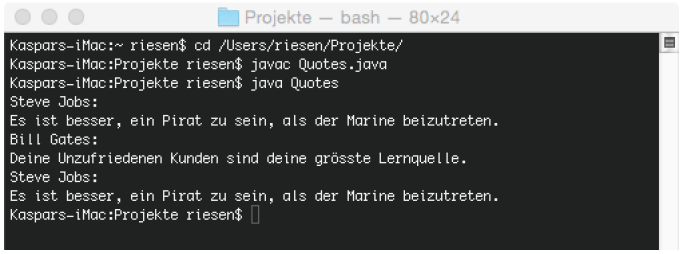

Abbildung 1.3: Java Programme in einem *Terminal* unter MacOS kompilieren und ausführen. →Video K-1.3

gramme). Das Kompilieren und Interpretieren von Java Programmen kann dann in der sogenannten *Konsole* des Computers durchgeführt werden (die Konsole heisst *Eingabeaufforderung* (Windows) bzw. *Terminal* (MacOS und Linux)) .

Nehmen wir an, dass der Quellcode für die Klasse `Quotes` in der Datei `Quotes.java` im Ordner `/Users/riesen/Projekte/` gespeichert ist. Wir öffnen die Konsole und wechseln in den Ordner, in dem sich die Datei mit dem Quellcode befindet.

Damit sich Programme übersetzen und ausführen lassen, müssen wir nun nacheinander die Programme `javac` (zum Kompilieren) und `java` (zum Ausführen) aufrufen. Mit dem Befehl `javac` können wir also die Klasse `Quotes` in der Datei `Quotes.java` kompilieren:

```
javac Quotes.java
```

Der Kompilierer legt – vorausgesetzt, der Quellcode ist fehlerfrei – nun die Datei `Quotes.class` an. Diese Datei enthält den Bytecode unseres Programmes. Beachten Sie, dass dieser Bytecode nun plattformunabhängig ist. Das heisst, das erzeugte Programm kann auf einem beliebigen Computer, der einen Interpreter für Java installiert hat, ausgeführt werden.

Der Java Interpreter liest den Bytecode Anweisung für Anweisung aus, übersetzt diese in Maschinensprache und führt die erzeugten Befehle auf dem Prozessor aus. Der entsprechende Befehl, der hierzu eingegeben werden muss, lautet:

```
java Quotes
```

Als Argument erwartet der Interpreter den Namen der Klasse (ohne Endung `.class`).

In Abb. 1.3 ist der gesamte Prozess, der in der Konsole durchgeführt wird, illustriert (in diesem Beispiel unter MacOS).

Eine *Entwicklungsumgebung* beschreibt die Menge von Werkzeugen, die man zum Erstellen, Testen, Modifizieren und Ausführen von Quellcode benötigt (es gibt sowohl frei erhältliche als auch kommerzielle Entwicklungsumgebungen). Eine Umgebung heisst *Integrierte Entwicklungsumgebung* (engl. *Integrated Development Environment* (*IDE*)), wenn diese mehrere Werkzeuge in einer Software vereint. IDEs bieten oftmals graphische Oberflächen und zusätzliche Fähigkeiten an, die das Programmieren unterstützen können.

Es existieren zahlreiche IDEs, die zur Java Programmentwicklung verwendet werden können, wie z.B. *NetBeans* oder *IntelliJ IDEA*. Wir empfehlen Ihnen für diese Vorlesung, die IDE *Eclipse* zu verwenden[6]. Eclipse ist weit verbreitet, frei erhältlich und unterstützt auch andere Programmiersprachen wie zum Beispiel *C*, *C++*, *PHP* oder *Python*. Eclipse ist zudem plattformunabhängig und läuft unter Windows, Linux und MacOS.

1.3.3 Programmierfehler

Jede Programmiersprache besitzt ihre eigene *Syntax*. Die Syntax einer Programmiersprache gibt vor, *wie* man gültige Programmieranweisungen bilden kann. Wir haben bereits einige Elemente der Java Syntax diskutiert (z.B. die Regeln, dass ein Bezeichner nicht mit einer Ziffer beginnen darf, dass Programmieranweisungen mit einem Semikolon abgeschlossen werden müssen, oder dass geschweifte Klammern den Klassen- und Methodenrumpf einschliessen müssen).

Die *Semantik* einer Programmieranweisung definiert, *was* passieren soll, wenn die Anweisung ausgeführt wird. Die Semantik beschreibt also die Bedeutung von Quellcode. Ein syntaktisch korrektes Programm muss nicht zwingend semantisch korrekt sein – ein Programm wird immer das tun, was wir tatsächlich programmiert haben und nicht das, was wir gemeint haben zu programmieren.

Die Syntax definiert, *wie* ein Programm geschrieben werden darf, die Semantik hingegen definiert, *was* ein geschriebenes Programm tun wird. Dementsprechend kann man zunächst zwei Fehlerarten unterscheiden:

[6] Obwohl sich die Details der einzelnen Umgebungen unterscheiden, ist der grundlegende Kompilierungs- und Ausführungsprozess im Wesentlichen immer derselbe.

- *Kompilierfehler* – Fehler in der Syntax
- *Logische Fehler* – Fehler in der Semantik

Wird ein Java Programm kompiliert, werden alle Regeln der Syntax vom Kompilierer kontrolliert. Jeder Fehler, der vom Kompilierer entdeckt wird, heisst *Kompilier-* oder auch *Syntaxfehler*. Falls ein Java Programm syntaktisch nicht korrekt ist (ein einziger Syntaxfehler reicht hierzu schon aus), erzeugt der Kompilierer eine Fehlermeldung und es wird kein Bytecode erzeugt. Das heisst, ein Programm mit fehlerhafter Syntax kann überhaupt nicht ausgeführt werden (auch nicht teilweise). Umgekehrt erzeugt der Kompilierer bei erfolgreicher Kompilierung den Bytecode und speichert diesen automatisch in einer Datei mit der Endung `.class` ab[7].

Die zweite Fehlerart umfasst *logische* oder *semantische Fehler*. In diesem Fall kompiliert der Quellcode und das Programm kann ausgeführt werden, aber es verhält sich fehlerhaft. Z.B. erzeugt das Programm – unter Verwendung einer falschen Formel – eine fehlerhafte Ausgabe.

Es existiert noch eine dritte Fehlerart:

- *Laufzeitfehler* – Fehler bei der Ausführung

Laufzeitfehler treten erst während der Programmausführung auf, z.B. beim Versuch, durch Null zu dividieren, was zu einer *nicht ordnungsgemässen Terminierung* führt (umgangssprachlich formuliert: Das Programm "stürzt ab").

Eine Programmiererin sollte ihre Programme immer ausführlich testen und die erwarteten Ergebnisse mit den tatsächlichen Ergebnissen vergleichen. Ausserdem liegt es an der Programmiererin, möglichst robuste Programme zu schreiben, welche verhindern, dass das Programm bei einer fehlerhaften oder unerwarteten Manipulation des Benutzers gleich abstürzt.

Wenn logische Fehler oder Laufzeitfehler beobachtet werden, so muss der Ursprung des Problems im Quellcode gefunden werden – dieser Prozess ist unter dem Namen *Debugging* bekannt und kann eine spannende, manchmal aber auch langwierige Aufgabe sein.

[7] IDEs wie z.B. Eclipse kompilieren den Quellcode ständig – also bereits während der Erstellung. Deshalb können Syntaxfehler bereits beim Schreiben des Quellcodes von Eclipse angezeigt werden (z.B. mit einer roten, gewellten Linie unterhalb des fehlerhaften Quellcodes).

1.4 Objektorientiertes Programmieren

Wie eingangs erwähnt, ist der primäre Zweck des Programmierens das Lösen eines Problems. Problemlösen wiederum besteht aus mehreren Aktivitäten:

- Verstehen des Problems
- Entwerfen einer Lösung
- Erwägen von Alternativen und Verfeinern der Lösung
- Implementieren der Lösung
- Testen der Lösung

Diese Aktivitäten werden typischerweise nicht linear von oben nach unten abgearbeitet – sie überschneiden sich, es gibt Rückkopplungen und es werden oft mehrere Iterationen benötigt, bis das Problem tatsächlich gelöst ist.

Der zentrale Schlüssel zur Problemlösung ist die Zerlegung des Problems in handhabbare Teile. In der objektorientierten Programmierung entwirft man hierzu separate, mehr oder weniger unabhängige Programmteile, die jeweils für gewisse Aufgaben verantwortlich sind. Die Gesamtheit dieser Programmteile, die dann zur Laufzeit miteinander interagieren können, entspricht der entwickelten Software. In Java sind diese Programmteile sogenannte *Objekte*, die in *Klassen* kategorisiert und definiert werden.

Objekte sind *die* fundamentalen Elemente in jedem Java Programm. Mit Objekten können *Entitäten* aus der echten Welt leicht und intuitiv mit einer Programmiersprache modelliert werden. Entitäten sind hierbei unterscheidbare, physisch oder gedanklich existierende Konzepte der zu modellierenden Welt (der sogenannten *Problemdomäne*). Die Möglichkeit, die Entitäten der Problemdomäne eins-zu-eins in einem Programm abzubilden, ist eine der wichtigsten Charakteristiken der objektorientierten Programmierung und hat massgeblich zum Erfolg dieses Paradigmas in der Softwareentwicklung beigetragen.

Beispiel 2 *Beispiele für Objekte sind: Menschen, Studierende, Angestellte, Fahrzeuge, Autos, Motorräder, Möbel, Schreibtische, Geräte, Handys, Tablets, Bankkonten, Sparkonten, Rechnungen, Zahlungen, Schulden, Nachrichten, Tweets, Videos, Esswaren, Teigwaren, Eiscremes, etc.*

1.4.1 Variablen und Methoden

Objekte besitzen zwei charakteristische Merkmale:

- Ein Objekt besitzt *Eigenschaften*, die entweder konstant sind oder sich im Laufe der Zeit ändern können.
- Ein Objekt besitzt ein *Verhalten* oder *Funktionalitäten*, d.h. es kann selber Dinge tun und/oder andere Objekte können etwas mit diesem Objekt tun.

Die Eigenschaften und die Funktionalitäten definieren das Objekt letztendlich als eigenständiges Konzept. Die Eigenschaften der Objekte programmiert man in Java mit *Variablen* und die verschiedenen Verhaltensmöglichkeiten von Objekten programmiert man in Java mit *Methoden*.

Die *Variablen* eines Objektes sind die Eigenschaften, die ein Objekt speichern und verwalten soll. Die in einem Objekt gespeicherten Variablen können auch wieder Objekte sein oder sogenannt *primitive Daten* (bestehend aus Zeichen und Ziffern). Ein Objekt *Bankkonto* kann z.B. den aktuellen *Kontostand* als eine Folge von Ziffern und den Kontoinhaber als Objekt *Kunde* speichern. Den Variablen eines Objektes können Werte zugewiesen werden und im Allgemeinen können sich diese Werte während der Programmausführung ändern. Die aktuellen Werte der Variablen eines Objektes nennt man auch den *Zustand des Objektes*.

Die *Methoden* eines Objektes bestimmen das mögliche Verhalten des Objektes. Eine *Methode* ist – wie wir schon wissen – eine Gruppierung von Programmieranweisungen, die unter einem Namen zusammengefasst werden. Wenn eine Methode aufgerufen wird, werden die einzelnen Anweisungen der Methode ausgeführt. Um z.B. das Verhalten "Einzahlung auf ein Bankkonto" zu programmieren, müssen wir im Objekt *Bankkonto* eine Methode definieren, die mehrere Anweisungen enthält, welche die Variable *Kontostand* des Objektes *Bankkonto* korrekt anpasst.

1.4.2 Objekt vs. Klasse

Objekte sind immer durch eine *Klasse* definiert. Eine Klasse ist ein Modell oder eine Art "Bauplan" für Objekte eines bestimmten Typs. Denken Sie an einen Bauplan eines Hauses erstellt durch einen Architekten. Ein Bauplan definiert die wichtigen Eigenschaften und Verhaltensmöglichkeiten des Hauses (Wände, Fenster, Türen, Raumanordnung, Anzahl Stockwerke, Heizung, Wasseranschlüsse, etc.).

Ist der Bauplan einmal erstellt, so können theoretisch mehrere Häuser nach diesem Bauplan gebaut werden. Einerseits werden sich die so gebauten Häuser in verschiedenen Aspekten unterscheiden (z.B. in der Farbe der Fassade, der tatsächlichen Adresse, der Möblierung, der Bewohner, etc.), andererseits sind die Häuser, da sie ja auf dem gleichen Bauplan basieren, in vielerlei Hinsicht identisch. Um ein komplett anderes Haus bauen zu können (z.B. ein Mehrfamilien- statt einem Einfamilienhaus), bräuchten wir einen anderen Bauplan.

Die Unterscheidung zwischen Klasse und Objekt ist sehr wichtig: Eine Klasse ist *kein* Objekt (ein Bauplan ist auch noch kein Haus). Eine Klasse ist "nur" der Bauplan für Objekte. Die Klasse bestimmt die Eigenschaften, die das Objekt beschreiben sollen (also die Variablen) sowie das Verhalten, das das Objekt ausführen können soll (also die Methoden). Ist eine Klasse aber einmal definiert, so können mehrere Objekte aus dieser Klasse erstellt oder *instanziiert* werden – man sagt auch: ein Objekt ist eine *Instanz* einer Klasse. Haben wir z.B. eine Klasse `Bankkonto` definiert, die das allgemeine Konzept eines Bankkontos repräsentiert, so können wir beliebig viele konkrete Objekte (also Bankkonten) aus der Klasse `Bankkonto` instanziieren.

Beispiel 3 *Im Java API existiert eine Klasse* `Circle`*, welche das Konzept eines graphischen Kreises in Java modelliert. Ein Kreis besitzt als Eigenschaften einen Mittelpunkt (definiert durch* (x, y)*-Koordinaten), einen Radius (in Pixel) und eine bestimmte Farbe. Diese Eigenschaften sind in der Klasse* `Circle` *als Variablen definiert. Zudem besitzt ein Kreis auch ein Verhalten – er kann z.B. seine Farbe ändern, sich in* x *und/oder* y *Richtung verschieben und seinen Radius vergrössern oder verkleinern. Dieses Verhalten ist in der Klasse* `Circle` *mit Methoden definiert:*

```
// Wir erzeugen vier Objekte aus der Klasse Circle
// Jedes Circle Objekt hat ein Zentrum (x,y) und einen Radius
Circle circle1 = new Circle(40, 40, 10);
Circle circle2 = new Circle(60, 50, 20);
Circle circle3 = new Circle(170, 70, 30);
Circle circle4 = new Circle(40, 80, 10);

// Jedes Circle Objekt erhält eine eigene Farbe
circle1.setFill(Color.AQUAMARINE);
circle2.setFill(Color.BISQUE);
circle3.setFill(Color.BLACK);
circle4.setFill(Color.CORAL);

// Das Objekt circle2 wird verschoben und vergrössert
circle2.setCenterX(100);
circle2.setRadius(40);
```

In obigem Beispiel werden aus der Klasse `Circle` *vier Objekte instanziiert und un-*

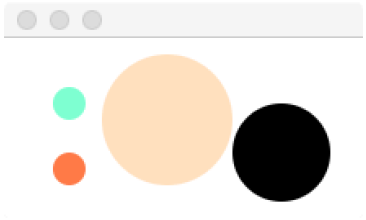

Abbildung 1.4: Das erzeugte Fenster mit vier Objekte der Klasse `Circle`.

ter den Bezeichnern `circle1`*,* `circle2`*,* `circle3` *und* `circle4` *gespeichert (mit unterschiedlichen Zentren und Radien). Die Details, wie die Objekte aus der Klasse instanziiert werden, sind jetzt noch nicht wichtig. Wichtig ist aber, dass Sie sehen, dass es eine Klasse* `Circle` *gibt, aus der wir beliebig viele Objekte instanziieren können.*

Jedes dieser Objekte ist unabhängig von den anderen Objekten. Wir rufen für die vier Objekte jeweils die Methode `setFill` *auf und definieren so für jedes der vier Kreisobjekte eine eigene Farbe. An Objekt* `circle2` *wird noch das Verhalten des Verschiebens (in x-Richtung) und das Verändern des Radius demonstriert (durch zwei Methodenaufrufe). Beachten Sie, dass jetzt nur das Kreisobjekt* `circle2` *verschoben und vergrössert wird.*

Das Programmfenster, das aus diesem Quellcode erzeugt wird, ist in Abb. 1.4 zu sehen. Obschon es nur eine Klasse `Circle` *gibt, sind in diesem Fenster die vier instanziierten und unabhängigen Objekte* `circle1`*,* `circle2`*,* `circle3` *und* `circle4` *vom Typ* `Circle` *zu sehen.*

Zusammenfassend: Eine Klasse entspricht einem Konzept – Objekte sind Realisierungen/Instanzen dieses Konzepts. Aus der gleichen Klasse können mehrere Objekte erzeugt werden.

1.4.3 Kapselung und Vererbung

Zwei der wichtigsten Eigenschaften der Objektorientierung sind die *Kapselung* (engl. *encapsulation*) und die Vererbung (engl. *inheritance*).

Kapselung bedeutet, dass die Eigenschaften eines Objektes (bzw. die Werte der Variablen des Objektes) gegen aussen verborgen bleiben. Das heisst, ein Objekt verbirgt seinen Zustand gegenüber anderen Objekten des Programmes und verwaltet

alle Zugriffe auf die eigenen Variablen selber.

Das bedeutet, dass z.B. Änderungen der Variablen eines Objektes nur über die Methoden dieses Objektes vollzogen werden können. Objekte sollten also so entworfen werden, dass andere Objekte keinen direkten Zugriff auf die Variablen des Objektes haben, sondern immer via Methoden auf die Variablen zugreifen müssen. Dies erlaubt dem Objekt die alleinige Kontrolle über dessen Zustand. Diese alleinige Kontrolle des Objektes über seinen Zustand hat den Vorteil, dass nur zulässige Veränderungen an den Variablen durchgeführt werden können.

Beispiel 4 *Gegeben sei ein Objekt* `rectangle`*, das aus einer Klasse* `Rectangle` *instanziiert wurde. Die Klasse* `Rectangle` *modelliere ein graphisches Rechteck. Kapselung bedeutet, dass der folgende direkte Zugriff, auf die im Objekt gespeicherte Variable* `width`*, verhindert werden sollte:*

```
rectangle.width = -5;
```

Vielmehr sollten wir sicherstellen, dass Änderungen der Variablen immer über eine entsprechende Methode erfolgen. Z.B.:

```
rectangle.setWidth(-5);
```

Dies hat den Vorteil, dass wir als Programmiererin in der Methode `setWidth` *durch "sorgfältiges" Programmieren sicherstellen können, dass der Variablen* `width` *nur gültige Werte zugewiesen werden. In obigem Beispiel könnte man die Methode* `setWidth` *z.B. so programmieren, dass die Variable* `width` *nur dann geändert wird, wenn die neue Breite eine positive Zahl ist.*

Klassen können auf der Basis bestehender Klassen definiert werden, indem *Vererbung* verwendet wird. Vererbung ist eine Möglichkeit, bereits programmierte Klassen wiederzuverwenden. Vererbung ist dann sinnvoll einsetzbar, wenn verschiedene Klassen, die wir erstellen wollen, starke Ähnlichkeiten aufweisen (also viele identische Variablen und Methoden besitzen). Da die Variablen und Methoden einer Klasse bei der Vererbung an die Kinder vererbt werden, können wir somit zuerst eine Basisklasse erstellen, von der dann die zu erstellenden, ähnlichen Klassen erben können.

Aus einer Klasse können also durch Vererbung verschiedene Klassen abgeleitet werden. Aus den abgeleiteten Klassen können dann wiederum weitere Klassen durch

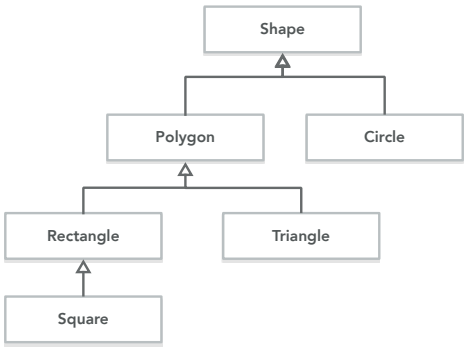

Abbildung 1.5: Eine einfache Klassenhierarchie für verschiedene geometrische Formen.

Vererbung abgeleitet werden. Somit kann eine komplette *Klassenhierarchie* erstellt werden. Gemeinsame (häufige) Charakteristiken werden dann typischerweise in Klassen definiert, die sich hoch oben in der Hierarchie befinden, während spezifischere Variablen und Methoden nur in abgeleiteten Klassen (tiefer unten in der Hierarchie) definiert werden.

Beispiel 5 *In Abb. 1.5 ist eine Klassenhierarchie für verschiedene geometrische Formen gezeigt. Eigenschaften und Verhaltensweisen, welche alle Formen teilen, würde man zuoberst in der Klasse* `Shape` *definieren (z.B. die Variable* `area`*). Die für Kreise spezifische Eigenschaft* `radius` *hingegen würde man dann wohl nur in der Klasse* `Circle` *definieren.*

Kapitel 2

Daten und Ausdrücke

2.1 Zeichenketten

Eine *Zeichenkette* ist eine Folge von beliebigen Zeichen (bestehend aus Buchstaben, Ziffern, Interpunktionszeichen und anderen Spezialzeichen). In Java kann man Zeichenketten definieren, indem Folgen von Zeichen innerhalb doppelter Anführungszeichen gesetzt werden. Wie wir im nächsten Kapitel sehen werden, sind Zeichenketten in Java eigentlich Objekte, welche durch die Klasse `String` definiert sind (für die Zwecke dieses Kapitels ist dies aber noch unwesentlich).

Beispiel 6 *Nachfolgend ein paar Beispiele für Zeichenketten:*

- *`"Wir sind Schweizermeister 2020!"`*
- *`"Meine eMail Adresse: kaspar.riesen@inf.unibe.ch"`*
- *`""`*

Das letzte Beispiel entspricht der leeren Zeichenkette – eine Zeichenkette, die also keine Zeichen enthält.

2.1.1 Die Methoden `print` und `println`

In der Klasse `Quote` im vorherigen Kapitel haben wir die Methode `println` unter anderem wie folgt aufgerufen:

```
System.out.println("Steve Jobs:");
```

Anhand dieser Programmieranweisung kann man schön illustrieren, wie man Methoden und Objekte in Java verwenden kann. Das Objekt `System.out` repräsentiert

ein Ausgabegerät oder eine Ausgabedatei[1]. Standardmässig entspricht das Objekt `System.out` der sogenannten *Konsole* des Computers, welche Ein- und Ausgaben von Text ermöglicht. Objekte modellieren i.d.R. ein Verhalten, das wir nutzen können – dieses Verhalten ist jeweils in Methoden definiert. Das Verhalten, das wir hier nutzen wollen, ist in der Methode `println` programmiert.

Um die Methode `println` nutzen zu können, benötigen wir den Objektnamen (in unserem Beispiel `System.out`) gefolgt vom Namen der Methode (`println`). Beachten Sie den Punkt zwischen dem Bezeichner des Objektes und dem Bezeichner der Methode. Nach dem Bezeichner einer Methode folgt *immer* ein Klammerpaar `()`. Das, was wir einer Methode innerhalb dieser Klammern mitgeben, heisst *Parameter*. Die Methode `println` erwartet einen einzigen Parameter, nämlich die Zeichenkette, die in der Konsole ausgegeben werden soll.

Zusammenfassend: Um eine Methode mit dem Bezeichner `methodenName` eines Objektes mit Bezeichner `objektName` und Parameter `parameterName` aufzurufen, folgt man in Java sehr oft folgender Schreibweise:

```
objektName.methodenName(parameterName);
```

Das heisst, Methoden werden i.d.R. auf Objekten aufgerufen (es gibt Ausnahmen für diese Regel, auf die wir später zu sprechen kommen werden). Wir haben dieses Prinzip im vorherigen Kapitel auch im Beispiel mit der Klasse `Circle` angewendet. Z.B. haben wir die Methode `setRadius` auf dem Objekt `circle2` aufgerufen:

```
circle2.setRadius(40);
```

Diese Methode verlangt eine ganze Zahl als Parameter. Es gibt Methoden, die keine Parameter benötigen – die Klammern sind trotzdem immer nötig. Z.B. Kann man die Methode `println` tatsächlich auch ohne Parameter aufrufen:

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

Diese Anweisung gibt eine Zeile ohne Inhalt aus.

[1] Präziser müsste man eigentlich sagen: Das Objekt mit dem Bezeichner `out`, das in der Klasse `System` gespeichert ist, repräsentiert ein Ausgabegerät oder eine Ausgabedatei.

Das Objekt `System.out` bietet mit der Methode `print` noch eine weitere Funktionalität an. Die Methode `print` ermöglicht Ausgaben *ohne* automatischen Zeilenumbruch. Das heisst, alles was nach einer `print` Anweisung ausgegeben wird, erscheint auf der gleichen Zeile. Ansonsten funktioniert die Methode `print` genau gleich wie `println`.

Beispiel 7 *Zur Unterscheidung von* ***print*** *und* ***println*** *beachten Sie folgendes Programm (→Video K-2.1):*

```
public class Start {

        /**
         * Erzeugt eine Ausgabe auf drei Zeilen
         */
        public static void main(String[] args) {
                System.out.print("Achtung, ");
                System.out.print("Fertig, ");
                System.out.println("Los!");
                System.out.println("Fehlstart!");
                System.out.print("Nochmals...");
        }
}
```

Die Ausgabe, die dieses Programm erzeugt, lautet:

```
Achtung, Fertig, Los!
Fehlstart!
Nochmals...
```

2.1.2 Konkatenation von Zeichenketten

Zeichenketten dürfen beliebig lang sein. Dies kann aber dazu führen, dass wir die Konvention, dass eine Zeile Quellcode nicht mehr als 80 Zeichen umfassen sollte, nicht einhalten können. Zur Erinnerung:

```
                System.out.println("Es ist besser, ein Pirat zu sein, als der Marine beizutreten.");
```

Wird die Zeichenkette aber einfach mit einem Zeilenumbruch getrennt, führt dies zu einem Kompilierfehler. Mit anderen Worten: Zeichenketten dürfen das Spezialzeichen "Zeilenumbruch" nicht enthalten. Folgende Lösung ist also nicht gültig:

```
                System.out.println("Es ist besser, ein Pirat zu sein,
                als der Marine beizutreten.");
```

Wenn wir eine lange Zeichenkette definieren, die sich über mehrere Zeilen erstreckt, können wir die *String Konkatenation* verwenden, um eine Zeichenkette mit einer anderen zu "verkleben". Der zugehörige Operator ist das Pluszeichen (+) und wird zum Beispiel folgendermassen verwendet:

```
System.out.println("Es ist besser, ein Pirat zu sein, " +
"als der Marine beizutreten.");
```

Beachten Sie, dass die ausgegebene Zeichenkette nun *keinen* Zeilenumbruch enthält. Die ausgegebene Zeichenkette befindet sich also auf einer Zeile – wir kommen gleich darauf zu sprechen, wie Sie in einer Zeichenkette Zeilenumbrüche integrieren können.

Der Konkatenationsoperator + kann auch dazu verwendet werden, um Zahlen mit Zeichenketten zu verbinden.

Beispiel 8 *In folgender Zeile wird eine Zeichenkette mit einer Zahl verbunden und das Resultat wird ausgegeben:*

```
System.out.println("Ich bin " + 42 + " Jahre alt.");
```

Beachten Sie, dass im obigen Beispiel die Zahl 42 nicht von Anführungszeichen umschlossen wird (das heisst, 42 wird als Zahl und nicht als Zeichenkette interpretiert). In diesem Fall wird die Zahl automatisch in eine Zeichenkette konvertiert und die beiden Zeichenketten werden anschliessend konkateniert.

Hinweis: Wir hätten natürlich auch einfach schreiben können (da Ziffern auch in Zeichenketten vorkommen dürfen):

```
System.out.println("Ich bin 42 Jahre alt.");
```

Neben der Konkatenation wird das Pluszeichen (+) auch für die arithmetische Addition verwendet. Die tatsächliche Funktion des Symbols + hängt vom Typ der Daten ab, auf denen es angewendet wird. Der Operator + führt eine Konkatenation durch, wenn mindestens einer der beiden Operanden eine Zeichenkette ist. Sind hingegen beide Operanden Zahlen, so werden diese addiert.

Beachten Sie, dass in Java Operationen grundsätzlich von links nach rechts ausgewertet werden – man kann aber Klammern verwenden, um die Reihenfolge der Auswertung zu beeinflussen.

Beispiel 9 *Beachten Sie das folgende Programm (→ Video K-2.2).*

```
public class AdditionConcatenation {

        /**
         * Konkateniert bzw. addiert zwei Zahlen
         */
        public static void main(String[] args) {
                System.out.println("Wir konkatenieren 7 und 3: " + 7 + 3);
                System.out.println(7 + 3 + ": Auswertung erfolgt von links nach rechts");
                System.out.println("Erzwingen einer anderen Reihenfolge: " + (7 + 3));
        }
}
```

Im ersten Aufruf der Methode **`println`** *führen beide* **`+`** *Operationen eine Konkatenation aus (da diese von links nach rechts ausgeführt werden). In der zweiten Anweisung wird zunächst eine Addition und danach eine Konkatenation ausgeführt. In der dritten Anweisung erzwingen wir mit Hilfe von Klammern, dass zuerst zwei Zahlen addiert werden und danach erst wird die Zeichenkette mit dem Resultat der Addition konkateniert. Die Ausgabe des Programmes lautet demnach:*

```
Wir konkatenieren 7 und 3: 73
10: Auswertung erfolgt von links nach rechts
Erzwingen einer anderen Reihenfolge: 10
```

2.1.3 Escape Sequenzen

Beachten Sie, dass Java doppelte Anführungszeichen verwendet, um den Start und das Ende einer Zeichenkette festzulegen. Das heisst, dass die folgende Anweisung einen Kompilierfehler auslösen wird, da der Kompilierer die zweiten Anführungszeichen als Ende der Zeichenkette interpretiert – statt eines `H` erwartet der Kompilierer jetzt eigentlich eine schliessende Klammer und ein Semikolon `);`

```
System.out.println("Er sagte: "Hallo."");
```

Um dieses (und ähnliche) Probleme zu bewältigen, bietet Java eine Reihe von *Escape Sequenzen* an, mit denen Spezialzeichen in Zeichenketten integriert werden können. *Escape Sequenzen* starten immer mit einem Backslash \. Hier vier Beispiele von solchen Sequenzen:

- `\t`: Tabulator
- `\n`: Zeilenumbruch

- \": Anführungszeichen
- \\: Backslash

Korrektur der fehlerhaften Anweisung von oben:

```
System.out.println("Er sagte: \"Hallo.\"");
```

Beispiel 10 *Die Ausgabe des Programmes* **Poem** *wird folgendermassen formatiert:*

```
Vier Maurer sassen einst auf einem Dach.
Da sprach der erste: "Ach!"
      Der zweite: "Wie ist's möglich dann?"
            Der dritte: "Dass das Dach halten kann!!!"
                  Der vierte: "Ist doch kein Träger dran!!!!",
Und mit einem Krach
Brach das Dach.

      \  Kurt Schwitters \
```

```
public class Poem {

        /**
         *  Gibt ein Gedicht formatiert aus
         */
        public static void main(String[] args) {
                System.out.println("Vier Maurer sassen einst auf einem Dach.\n"+
                                "Da sprach der erste: \"Ach!\"\n"+
                                "\tDer zweite: \"Wie ist's möglich dann?\"\n"+
                                "\t\tDer dritte: \"Dass das Dach halten kann!!!\"\n"+
                                "\t\t\tDer vierte: \"Ist doch kein Träger dran!!!!!\"\n"+
                                "Und mit einem Krach\n"+
                                "Brach das Dach. \n\n\t \\ Kurt Schwitters \\ ");
        }
}
```

Beispiel 11 *In folgendem Programm wird in verschiedenen* **println** *Anweisungen der Umgang mit Zeichenketten zusammengefasst (→Video K-2.3).*

```
public class PrimFacts {

        /*
         * Vom Umgang mit Zeichenketten
         */
        public static void main(String[] args) {
                // Zeichenketten können konkateniert werden
                System.out.println("Es gibt unendlich viele Primzahlen. Ein System, "
                                + "welche Zahlen Primzahlen sind, ist nicht bekannt.");

                // Sonderzeichen werden mit Escape Sequenzen integriert
                System.out.println("\"Die Anzahl der Dummheiten übersteigt die der "
```

```
                                + "Primzahlen.\nGibt es nicht unendlich viele "
                                + "Primzahlen?\"\n\tGregor Brand");

                // Zeichenketten dürfen auch Ziffern enthalten
                System.out.println("Primzahlen sind natürliche Zahlen, die genau "
                                + "2 Teiler haben, nämlich 1 und sich selbst.");

                // Zahlen können zu Zeichenketten konkateniert werden
                // (automatische Konvertierung Zahl -> Zeichenkette)
                System.out.println("Daher zählt die " + 1 + " nicht zu den Primzahlen.");
        }
}
```

2.2 Variablen und Konstanten

In Java ist eine *Variable* ein Name für einen Speicherort, der einen bestimmten Wert enthalten kann. Die sogenannte *Deklaration* einer Variablen enthält immer zwei Dinge: Der Datentyp der Variablen und einen Bezeichner für die Variable.

Folgende Codezeile deklariert eine Variable vom Datentyp `int` mit dem Bezeichner `pages` (der Datentyp `int` entspricht einer ganzen Zahl (engl. *integer*)):

```
int pages;
```

Mit einer Deklaration instruieren wir den Kompilierer, dass ein Teil des Hauptspeichers reserviert werden muss, der gross genug ist, um einen Wert des angegebenen Datentyps speichern zu können.

Solange einer deklarierten Variablen noch kein Wert zugewiesen ist, nennen wir die Variable *undefiniert*. Ist eine Variable deklariert, so können wir dieser einen Wert des deklarierten Typs zuweisen:

```
pages = 256;
```

In diesem Fall sprechen wir von einer *Zuweisung*, da diese Operation einer Variablen einen Wert zuweist. Wird eine Zuweisung ausgeführt, so wird die rechte Seite des Zuweisungsoperators (`=`) ausgewertet und das Resultat wird im Speicher an den Platz gelegt, auf den die Variable auf der linken Seite zeigt.

Der Datentyp der Variablen muss bei einer Zuweisung nicht nochmals angegeben werden – dieser wird nur genau einmal angegeben, nämlich bei der Deklaration der

Variablen. Eine Deklaration und eine Zuweisung können aber auch auf der gleichen Zeile erfolgen:

```
int figures = 46;
```

In einer Deklaration können zudem mehrere Variablen des gleichen Typs gleichzeitig deklariert werden. Jede Variable der Deklaration kann dabei mit oder ohne initiale Zuweisung deklariert werden. Folgende Zeile deklariert zwei Variablen vom Typ `int`. Eine Variable bleibt undefiniert und einer Variablen wird der Wert `46` zugewiesen:

```
int figures = 46, tables;
```

Beispiel 12 *Beachten Sie die Klasse* `BookStatistics` *(→ Video K-2.4):*

```
public class BookStatistics {

        /*
         * Demonstriert das Referenzieren von Variablen
         */
        public static void main(String[] args) {
                int pages;
                pages = 256;
                int figures = 46, tables;
                tables = 17;

                System.out.println("Anzahl Seiten: " + pages);
                System.out.println("Anzahl Abbildungen: " + figures
                                + "; Anzahl Tabellen: " + tables);
        }
}
```

Zunächst deklarieren wir eine Variable mit dem Bezeichner `pages`*, die eine ganze Zahl – also einen* `int` *– speichern kann. Danach weisen wir der Variablen* `pages` *den Wert 256 zu und es werden zwei weitere Variablen vom Typ* `int` *deklariert und auch diesen Variablen werden Werte zugewiesen.*

Eine Deklaration reserviert genügend Speicherplatz für den angegebenen Datentyp. Gleichzeitig definiert eine Deklaration den Bezeichner, über den der Wert/Inhalt dieser Variablen *referenziert* werden kann. Das heisst, dieser Bezeichner kann nach der Deklaration in unserem Quellcode verwendet werden.

Im Programm `BookStatistics` werden beim ersten Aufruf der Methode `println` zwei Informationseinheiten verwendet: Eine Zeichenkette (`"Anzahl Seiten: "`) und die Variable `pages`. Wird eine Variable referenziert, so wird der Wert, der unter

dieser Variablen gespeichert ist, verwendet. Daher wird dieses Programm folgende Ausgabe generieren:

```
Anzahl Seiten: 256
Anzahl Abbildungen: 46; Anzahl Tabellen: 17
```

Wir betrachten als nächstes ein Programm, das den Wert einer Variablen zur Laufzeit des Programmes ändert (→Video K-2.5):

```java
public class Polygon {

        /**
         * Demonstriert Zuweisungen zu einer Variablen
         */
        public static void main(String[] args) {

                int sides = 3; // Deklaration und Zuweisung
                System.out.println("Anzahl Seiten eines Trigons: " + sides);
                sides = 12; // Zuweisung
                System.out.println("Anzahl Seiten eines Dodekagons: " + sides);
                sides = 19; // Zuweisung
                System.out.println("Anzahl Seiten eines Nonadekagons: " + sides);

                // lesen (ohne Zuweisung) ändert die Variable nicht!
                System.out.println("Anzahl Seiten eines Ikosagons: " + (sides + 1));
                System.out.println("In der Variablen sides ist gespeichert: " + sides);
        }
}
```

In diesem Programm wird zunächst eine Variable vom Typ `int` mit dem Bezeichner `sides` deklariert und mit dem Wert `3` initialisiert. Danach wird der aktuelle Wert der Variablen `sides` ausgegeben. Die nächste Anweisung in der Methode `main` ändert den gespeicherten Wert der Variablen `sides` mit einer neuen Zuweisung auf den Wert `12`. Danach erfolgt eine weitere Zuweisung mit dem Wert `19`. Eine Variable kann immer nur einen Wert gleichzeitig speichern. Bei einer Zuweisung wird der alte Wert also mit dem neu zugewiesenen Wert überschrieben. Die ersten drei Ausgaben des Programmes `Polygon` lauten deshalb:

```
Anzahl Seiten eines Trigons: 3
Anzahl Seiten eines Dodekagons: 12
Anzahl Seiten eines Nonadekagons: 19
```

Wird eine Variable aber nur verwendet (z.B. innerhalb einer arithmetischen Operation), ohne dass eine Zuweisung stattfindet, so bleibt der Wert der Variablen un-

verändert: Lesen von Daten verändert die Daten niemals – nur mit Zuweisungen können Sie die Werte von Variablen ändern! Betrachten Sie zum Beispiel die Anweisung:

```
System.out.println("Anzahl Seiten eines Ikosagons: " + (sides + 1));
```

Hier wird die Variable `sides` nur gelesen und deshalb bleibt diese unverändert. Die zwei weiteren Ausgaben des Programmes `Polygon` lauten deshalb:

```
Anzahl Seiten eines Ikosagons: 20
In der Variablen sides ist gespeichert: 19
```

Eine Konstante ist ein Bezeichner der ähnlich zu einer Variablen ist. Der Unterschied besteht darin, dass einer Konstante während ihrer gesamten Existenz nur einmal ein Wert zugewiesen werden kann (wie der Begriff andeutet, ist es eben eine *Konstante* und nicht eine *Variable*).

In Java kann mit dem reservierten Wort `final` in der Deklaration festgelegt werden, dass eine Variable eine unveränderliche Konstante speichern soll. Folgende Zeile deklariert bspw. eine Konstante mit dem Bezeichner `MAX_VISITORS` vom Typ `int` und weist der Konstanten den Wert `499` zu (z.B. zur Festlegung der maximalen Anzahl Zuschauer in einem Kino):

```
final int MAX_VISITORS = 499;
```

Die Konvention, dass man Bezeichner für Konstanten mit Grossbuchstaben schreibt und einzelne Wörter mit einem Unterstrich trennt, hilft dem menschlichen Leser Konstanten von Variablen zu unterscheiden.

Konstanten sind aus mindestens drei Gründen hilfreich:

- Wenn wir in unserem Quellcode die maximal zulässige Zuschauerzahl referenzieren wollen, so könnten wir auch einfach die Zahl `499` verwenden. Die Verwendung des Bezeichners `MAX_VISITORS` im Quellcode ist im Gegensatz zur Verwendung der Zahl `499` aber besser verständlich[2].

[2] Werden im Quellcode Zahlen statt Konstanten/Variablen verwendet, so spricht man auch von *Magic Numbers*, weil i.d.R. nicht klar ist, was diese bedeuten, woher sie kommen und ob diese verändert werden dürfen.

- Konstanten vereinfachen die Wartung des Quellcodes. Auch wenn die Konstante `MAX_VISITORS` an mehreren Stellen im Quellcode verwendet wird, muss diese nur einmal mit dem Wert `499` definiert werden. So wird die Gefahr eliminiert, dass bei nachträglichen Anpassungen nicht alle Werte im Quellcode gefunden und angepasst werden.

- Der Kompilierer wird eine Fehlermeldung erzeugen, wenn man versucht, den Wert einer Konstanten mit einer weiteren Zuweisung zu ändern. Konstanten stellen also sicher, dass einmal zugewiesene Werte nicht verändert werden.

2.3 Primitive Datentypen

In Java existieren acht *primitive Datentypen* mit denen Variablen deklariert werden können. Vier dieser Datentypen repräsentieren ganze Zahlen (`byte`, `short`, `int`, `long`), zwei repräsentieren Gleitkommazahlen (`float`, `double`), ein Datentyp steht für einzelne Zeichen (`char`) und einer steht für Wahrheitswerte (`boolean`). Alle acht Bezeichner für diese Datentypen sind reservierte Wörter in Java.

- **Numerische Datentypen:** Die Unterschiede der numerischen Datentypen bezieht sich auf deren Grösse und der Wertebereiche, die diese speichern können:

Datentyp	**Min, Max**
`byte`	$-128, 127$
`short`	$-32'768, 32'767$
`int`	$-2'147'483'648, +2'147'483'647$
`long`	$\approx \pm 9 \times 10^{18}$
`float`	$\approx \pm 3.4 \times 10^{38}$
`double`	$\approx \pm 1.7 \times 10^{308}$

Wir werden in diesem Skript häufig den Datentyp `int` für ganze Zahlen und den Datentyp `double` für Gleitkommazahlen verwenden.

Beispiel 13 *Nachfolgend ein paar Beispiele von Deklarationen (und Zuweisungen) von numerischen Variablen und Konstanten:*

```
int num = -17;
long var1, var2, var3;
final double PI = 3.141592655359;
```

- **Zeichen:** Eine Variable vom Typ `char` kann *genau ein* Zeichen speichern (Gross und Kleinbuchstaben, Ziffern, Interpunktionszeichen und weitere Sonderzeichen – auch unter Verwendung der *Escape* Sequenzen). Ein `char` wird in Java mit einem *einfachen* Anführungszeichen definiert (z.B. `'b'` oder `'J'`).

  Beispiel 14 *Nachfolgend ein paar Beispiele von Deklarationen (und Zuweisungen) von Variablen vom Typ* `char`*:*

  ```
  char topValue = '6';
  char character1 = 'A', character2 = 'a';
  char eol = ';', sep = '\t', space = ' ';
  ```

  Hinweis: Verwechseln Sie den Datentyp `char` nicht mit Zeichenketten. Zeichenketten werden mit doppelten Anführungszeichen begrenzt und dürfen null oder mehr Zeichen beinhalten (`""`, `"Q"`, `"ABC"`, ...). Eine Zeichenkette ist auch *kein* primitiver Datentyp, sondern ist definiert durch eine eigene Klasse – die Klasse `String`, welche im nächsten Kapitel genauer betrachtet wird. Tatsächlich kann man aber Zeichenketten genau gleich wie Variablen eines primitiven Datentyps deklarieren:

  ```
  String name = "Mathias";
  ```

- **Wahrheitswerte:** Eine *Boolesche Variable*, d.h. eine Variable vom Typ `boolean`, kann nur die zwei Werte `true` oder `false` annehmen (`true` und `false` sind beides reservierte Wörter). Eine Variable vom Typ `boolean` wird üblicherweise dazu verwendet, um anzugeben, ob eine bestimmte Bedingung wahr oder falsch ist.

  Beispiel 15 *Nachfolgend ein paar Beispiele von Deklarationen (und Zuweisungen) von Variablen vom Typ* `boolean`*:*

  ```
  boolean in = false;
  boolean tooLarge, tooSmall;
  boolean done = true;
  ```

2.3.1 Datenkonvertierung

Jede Variable in Java ist von einem bestimmten Datentyp. Java ist eine sogenannt *stark typisierte* Sprache. Das bedeutet, dass es Java nicht erlaubt, einer Variablen einen Wert zuzuweisen, der nicht dem deklarierten Typ entspricht. Manchmal ist

es aber hilfreich oder nötig den Datentyp einer bestimmten Variablen anzupassen (in einer bestimmten Situation möchten wir z.B. eine Ganzzahl als Gleitkommazahl behandeln). Das heisst, wir benötigen die Möglichkeit Datentypen von Variablen zu *konvertieren*.

Konvertierungen von einem Datentyp mit einem kleinen Wertebereich zu einem anderen Datentyp mit einem grösseren Wertebereich (z.B. vom Datentyp `int` zum Datentyp `double`) können problemlos durchgeführt werden, da hierbei keine Informationen verloren gehen können.

Beispiel 16 *In folgendem Beispiel weisen wir der Variablen* `num` *vom Typ* `double` *einen* `int` *Wert zu.*

```
int count = 17;
double num = count;
```

Für die zweite Zuweisung wird der Wert der Variablen `count` automatisch in einen `double` konvertiert. Das heisst, die Variable `num` enthält nun den Wert `17.0`. Der Datentyp von `count` ist aber – und das ist wichtig – weiterhin `int` und der gespeicherte Wert in `count` ist nach wie vor die ganze Zahl `17`. Konvertierungen ändern den Typ einer Variablen oder den darin gespeicherten Wert also nicht – sie konvertieren nur einen Wert als Teil einer Berechnung oder Zuweisung.

Immer wenn von einem "grösseren" zu einem "kleineren" Datentyp konvertiert wird, besteht die Gefahr, dass wir Informationen verlieren. Deshalb ist die umgekehrte Zuweisung nicht erlaubt. Würden wir also z.B. versuchen, der Variablen `count` einen Wert vom Typ `double` zuzuweisen, erhalten wir einen Kompilierfehler. Das heisst, folgende Zuweisung funktioniert nicht:

```
double num = 12.34;
int count = num;
```

Mit einer sogenannten *Cast-Konvertierung* können wir aber auch solche Zuweisungen erzwingen. Ein *Cast* ist ein Java Operator, der durch einen Typennamen in Klammern definiert ist. Der *Cast* Operator wird vor die Variable, deren Datentyp konvertiert werden soll, gestellt. Soll Beispielsweise `num` vom Typ `double` zu einer ganzen Zahl konvertiert werden, kann dies mit folgender Anweisung geschehen:

```
int count = (int) num;
```

Diese *Cast* Operation entfernt den gebrochenen Teil vom Wert, der in `num` gespeichert ist. Ist z.B. der Wert `12.34` in `num` gespeichert, so enthält nun `count` den Wert `12`. Der Typ und der Wert der Variablen `num` wird dabei aber nicht verändert (ist also weiterhin `double` bzw. `12.34`).

2.4 Eingaben vom Benutzer Entgegennehmen – *Teaser* zur Klasse `Scanner`

Die bis hierhin geschriebenen Programme lassen keine Interaktion mit dem Benutzer zu. Viele Programme werden aber erst dann "spannend", wenn der Benutzer Eingaben machen kann und das Programm dann seine Anweisungen mit dieser Eingabe durchführt.

Die Klasse `Scanner` aus dem Java API bietet Methoden an, um Eingaben vom Benutzer über die Tastatur in ein Programm einzulesen. Um die Funktionalität der Klasse `Scanner` nutzen zu können (d.h. deren Methoden aufzurufen), muss zuerst mit dem *`new`-Operator* und dem sogenannten *Konstruktor* ein entsprechendes Objekt instanziiert werden[3].

Folgende Programmieranweisung deklariert eine Objektvariable mit Bezeichner `scan` und weist dieser ein neu instanziiertes Objekt vom Typ `Scanner` zu:

```
Scanner scan = new Scanner(System.in);
```

Der Konstruktor der Klasse `Scanner` erwartet als Parameter die Quelle der Eingabe. Wir übergeben dem Konstruktor in obigem Beispiel das Objekt `System.in`, welches der Tastatur des Computers entspricht.

Ist ein Objekt vom Typ `Scanner` instanziiert, so können die Methoden mit dem Punkt-Operator aufgerufen werden. Wir verwenden vorerst nur die folgenden zwei Methoden der Klasse `Scanner`:

- `String nextLine()`: Gibt die komplette nächste Zeile als `String` zurück.

[3] Wir werden die Klasse `Scanner` in diesem Kapitel in einigen Beispielen verwenden – das Konzept der Instanziierung von Objekten aus Klassen mit Hilfe des Konstruktors wird im nächsten Kapitel ausführlich behandelt.

- `int nextInt()`: Gibt das nächste Element der Eingabe als Wert vom Typ `int` zurück. Analog dazu existiert z.B. die Methode `nextDouble()` u.a.

Beispiel 17 *Folgendes Programm verwendet ein* `Scanner` *Objekt (→ Video K-2.6).*

```
import java.util.Scanner;

public class Echo {

        /**
         * Gibt ein Echo der Benutzereingabe aus
         */
        public static void main(String[] args) {
                Scanner scan = new Scanner(System.in);

                System.out.print("Ihr Text: ");
                String input = scan.nextLine();
                System.out.println("Mein Echo:\n\"" + input + "\"");
        }
}
```

Eine mögliche Ein- und Ausgabe des Programmes sieht folgendermassen aus:

```
Ihr Text: Hallo Java!
Mein Echo:
"Hallo Java!"
```

Die Klasse `Scanner` muss gesondert importiert werden, wenn wir diese verwenden wollen. Beachten Sie hierzu die `import` Deklaration auf Zeile 1:

```
import java.util.Scanner;
```

Diese Anweisung teilt dem Programm mit, dass wir die Klasse `Scanner` in unserem Quellcode verwenden wollen[4].

2.5 Arithmetische Ausdrücke

Ein *Ausdruck* (engl. *Expression*) ist eine Kombination von einem oder mehreren Operatoren und Operanden. Die Operanden in einem Ausdruck können unter anderem Werte eines bestimmten Datentyps (z.B. `427`, `3.14`, `false`, `"Haus"`, `'c'`, etc.) oder zuvor deklarierte Variablen/Konstanten sein.

[4]Solche `import` Anweisungen werden ebenfalls nächste Woche ausführlich besprochen.

Im Folgenden werden wir uns auf arithmetische Ausdrücke fokussieren, die numerische Operanden verwenden und numerische Resultate berechnen (im nächsten Abschnitt behandeln wir dann Boolesche Ausdrücke, die mit Variablen vom Typ `boolean` operieren). In Java sind die vier arithmetischen Grundoperationen (`+`, `-`, `*`, `/`) wie gewohnt definiert.

Beispiel 18 *Zwei Beispiele von arithmetischen Ausdrücken in Java (eine Addition und eine Multiplikation):*

```
double num1 = 5.1, num2 = 3.2;
double sum = num1 + num2;
double product = num1 * num2;
```

Die Grundoperationen sind nicht nur für Gleitkommazahlen (`double`), sondern auch für ganze Zahlen (`int`) definiert. Bei der Division von ganzen Zahlen muss man allerdings vorsichtig sein.

Wenn *beide* Operanden einer Division ganze Zahlen sind, so wird das Resultat der Operation auch eine ganze Zahl (der gebrochene Teil des Resultates wird dann einfach von Java weggelassen.) Das bedeutet z.B., dass in Java `14/3` zu `4` oder `8/12` zu `0` ausgewertet wird. Umgekehrt reicht es, wenn einer der beiden Operanden eine Gleitkommazahl ist, um eine Gleitkommadivision zu erzwingen (d.h. in Java werden `10.0/4`, `10/4.0` oder `10.0/4.0` alle zu `2.5` ausgewertet).

Beispiel 19 *Folgendes Code-Fragment weist der Variablen* `result` *den Wert* `6.6667` *zu. Ohne Cast* `(double) total` *würde die Variable* `result` *den Wert* `6.0` *speichern:*

```
int total = 40;
double result = (double) total / 6;
```

Java besitzt neben den vier Grundoperatoren noch weitere arithmetische Operatoren, z.B. den Modulo-Operator `%`. Dieser Operator gibt den Rest zurück, der entsteht, wenn man den ersten Operanden durch den zweiten Operanden dividiert. Zum Beispiel ergibt `18 % 4 = 2`, da `18/4 = 4` mit *Rest* `2`.

Der Modulo-Operator kann auch auf Gleitkommazahlen angewendet werden. Dies kann insbesondere hilfreich sein, wenn man den Nachkommabereich einer Zahl extrahieren möchte (`12.34 % 1 = 0.34`).

Operatoren können auch kombiniert werden. Z.B. kann man in Java folgende Anweisung programmieren:

```
double result = 20 + 10 / 2;
```

In Java werden (wie aus der Mathematik bekannt) Multiplikationen und Divisionen vor Additionen und Subtraktionen durchgeführt (der Modulo-Operator besitzt die gleiche Priorität wie die Multiplikation und Division). Operatoren mit der gleichen Priorität werden von links nach rechts abgearbeitet – mit Hilfe von Klammern kann die Reihenfolge der Operationen aber beeinflusst werden.

Beispiel 20 *Die folgende Anweisung*

```
double result = 20 + 10 / 2;
```

weist der Variablen `result` *den Wert* `25` *zu, während*

```
double result = (20 + 10) / 2;
```

der Variablen `result` *den Wert* `15` *zuweist. Beachten Sie, dass Klammern auch geschachtelt werden können. Die Evaluation eines Ausdrucks beginnt dann bei der innersten Klammer und arbeitet sich schrittweise nach aussen vor. Die folgende Anweisung*

```
double result = 3 * ((15 - 3) / 2);
```

weist also der Variablen `result` *den Wert* `18` *zu.*

Hinweis: Der Zuweisungsoperator (=) hat die kleinere Priorität als *alle* arithmetischen Operationen. Die gesamte rechte Seite einer Zuweisung wird also immer zuerst berechnet und danach erst der Variablen auf der linken Seite des Zuweisungsoperators zugewiesen (siehe Abb. 2.1).

Beispiel 21 *Betrachten Sie die Klasse* `Grading` *(→ Video K-2.7):*

```
import java.util.Scanner;

public class Grading {

        /**
```

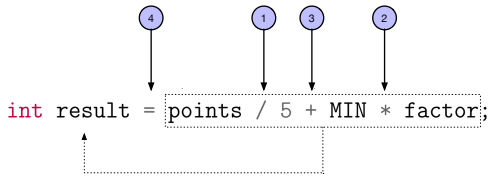

Abbildung 2.1: Reihenfolge der Auswertung: Der Zuweisungsoperator = hat die niedrigste Priorität.

```
         * Berechnet einen Prozentwert und eine Schulnote
         * auf Basis der erreichten Punkte
         */
        public static void main(String[] args) {
                final int MAX_POINTS = 20;

                Scanner scan = new Scanner(System.in);
                System.out.print("Erreichte Punkte eingeben: ");
                int points = scan.nextInt(); // erreichte Punkte einlesen

                double percentage = (double) points / MAX_POINTS * 100;
                double grade = (double) points / MAX_POINTS * 5 + 1;

                System.out.println("Sie haben " + points + " von " +
                                MAX_POINTS + " Punkten erreicht");
                System.out.println("Dies entspricht " + percentage + " Prozent");
                System.out.println("Schulnote: " + grade);
        }
}
```

Das Programm liest zunächst die erreichten Punkte vom Benutzer ein. Danach werden die ganzzahlige Konstante `MAX_POINTS` *und die ganzzahlige Variable* `points` *in zwei arithmetischen Ausdrücken miteinander verrechnet und die Resultate werden schliesslich den Variablen* `percentage` *und* `grade` *zugewiesen. Eine mögliche Ein- und Ausgabe des Programmes lautet:*

```
Erreichte Punkte eingeben: 17
Sie haben 17 von 20 Punkten erreicht
Dies entspricht 85.0 Prozent
Schulnote: 5.25
```

Die rechte und linke Seite einer Zuweisung dürfen auch die gleiche Variable beinhalten. Das heisst, dass zum Beispiel folgende Anweisung syntaktisch korrekt ist:

```
count = count + 1;
```

Zuerst wird `1` zum ursprünglichen Wert von `count` addiert und danach wird das

berechnete Resultat `count` zugewiesen (das heisst, der alte Wert von `count` wird überschrieben). Beachten Sie insbesondere den Unterschied zur Anweisung

```
count + 1;
```

welche keine Wirkung auf die Variable `count` hat.

Zwei weitere sehr praktische arithmetische Operatoren sind das *Inkrement* (`++`) und das *Dekrement* (`--`). Beide Operatoren verwenden nur einen Operanden und addieren bzw. subtrahieren zu diesem `1`. Die folgende Anweisung führt bspw. dazu, dass der Wert von `count` um `1` erhöht wird:

```
count++;
```

Das Resultat wird wiederum in der Variablen `count` gespeichert. Das heisst, das Inkrement ist funktional äquivalent zu

```
count = count + 1;
```

Oftmals müssen auf Variablen Operationen ausgeführt werden und das Resultat dieser Operation soll danach wieder in der gleichen Variablen gespeichert werden. Zum Beispiel möchten wir zur Variablen `num` den Wert der Variablen `count` addieren. Beachten Sie, dass die Anweisung

```
num + count;
```

die Variable `num` nicht verändert. Um den Wert der Variablen `num` entsprechend zu ändern, schreiben wir folgende Anweisung:

```
num = num + count;
```

Der Einfachheit halber existieren in Java *Zuweisungsoperatoren*, welche diesen Vorgang erleichtern. Zum Beispiel ist die Anweisung

```
num += count;
```

äquivalent zu der obigen Zuweisung.

Es existieren unterschiedliche Zuweisungsoperatoren in Java, z.B.:

- `x += y` entspricht `x = x + y`
- `x -= y` entspricht `x = x - y`
- `x *= y` entspricht `x = x * y`
- `x /= y` entspricht `x = x / y`

Die rechte Seite einer Zuweisungsoperation kann auch ein komplexer Ausdruck sein. Dieser Ausdruck wird zuerst komplett ausgewertet und erst dann mit der linken Seite kombiniert (und in der Variablen gespeichert). Dies gilt für alle Zuweisungsoperatoren. Das bedeutet, dass z.B.

```
total += (points - 10) / MAX_POINTS;
```

äquivalent ist zu

```
total = total + ((points - 10) / MAX_POINTS);
```

2.6 Boolesche Ausdrücke und die `if`-Anweisung

Java Programme starten immer mit der ersten Anweisung der `main` Methode und führen danach – solange nichts anderes definiert ist – alle weiteren Anweisungen dieser Hauptmethode von oben nach unten aus, bis das Ende der Methode `main` erreicht wird. Innerhalb einer Methode können wir aber mit Hilfe von speziellen Programmieranweisungen – z.B. mit *Bedingungen* – die Reihenfolge der Ausführungen beeinflussen. In diesem Abschnitt schauen wir hierzu eine erste Möglichkeit an.

Eine *`if`-Anweisung* besteht aus dem reservierten Wort `if` gefolgt von einem *Booleschen Ausdruck* in Klammern, gefolgt von einer beliebigen Programmieranweisung. Ein Boolescher Ausdruck ist entweder wahr oder falsch (besitzt also den Wert `true` oder `false`). Wenn er wahr ist, so wird die Programmieranweisung ausgeführt und wenn nicht, so wird die Programmieranweisung übersprungen und es wird mit der nächsten Anweisung unterhalb der `if`-Anweisung fortgefahren (siehe Abb. 2.2).

Beispiel 22 *Betrachten Sie folgendes Beispiel einer `if`-Anweisung:*

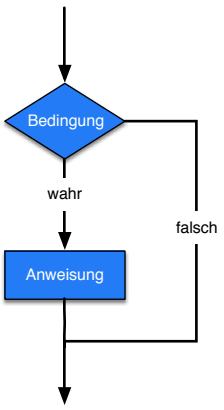

Abbildung 2.2: Die Logik einer `if`-Anweisung.

```
if (count > 10)
        System.out.print("10 überschritten!");
```

Der Boolesche Ausdruck in dieser ***`if`****-Anweisung ist der geklammerte Ausdruck* ***`count`*** *> 10. Dieser Ausdruck ist entweder wahr oder falsch. Wenn* ***`count`*** *grösser ist als 10, so wird die* ***`print`*** *Anweisung ausgeführt. Andernfalls wird die* ***`print`*** *Anweisung einfach übersprungen.*

Hinweis: Die Anweisung unter der `if` Klausel ist eingerückt. Dies impliziert, dass diese Programmieranweisung zur gesamten `if`-Anweisung gehört (und nur dann ausgeführt wird, wenn die Bedingung wahr ist). Für den menschlichen Leser von Quellcode sind solche Einrückungen wichtig, der Kompilierer hingegen ignoriert jegliche Formatierungen des Quellcodes.

Beispiel 23 *Das folgende Beispielprogramm fällt eine Entscheidung aufgrund des Wertes der Variablen* ***`hours`*** *(→ Video K-2.8).*

Wenn der Wert dieser Variablen grösser ist, als die im Quellcode definierte Konstante ***`MAX_HOURS`****, wird* ***`Arbeit stoppen!`*** *ausgegeben. Andernfalls wird diese* ***`println`*** *Anweisung übersprungen. In jedem Fall wird am Schluss* ***`Arbeitszeit überprüft.`*** *ausgegeben.*

```
import java.util.Scanner;

public class LimitCheck {

        /**
```

```
         * Überprüft, ob die geleistete Arbeitszeit
         * kleiner als die maximal erlaubte Zeit ist
         */
        public static void main(String[] args) {
                // maximal erlaubte Arbeitszeit
                final int MAX_HOURS = 500;

                Scanner scan = new Scanner(System.in);
                System.out.print("Arbeitszeit in Stunden eingeben: ");
                int hours = scan.nextInt(); // tatsächlich geleistete Arbeitszeit

                if (hours > MAX_HOURS)
                        System.out.println("Arbeit stoppen!");
                System.out.println("Arbeitszeit überprüft.");
        }
}
```

2.6.1 Boolesche Ausdrücke

Das obige Beispiel einer `if`-Anweisung basiert auf einem Booleschen Ausdruck, der zwei numerische Werte miteinander vergleicht (im Beispiel haben wir hierzu den Operator `>` verwendet). In Java existieren weitere sogenannte *relationale Operatoren*, die den Vergleich zweier numerischer Werte und die Definition eines Booleschen Ausdrucks ermöglichen:

- `==`: Sind zwei Werte gleich?
- `!=`: Sind zwei Werte ungleich?
- `<`: Ist der erste Wert kleiner als der zweite Wert?
- `>`: Ist der erste Wert grösser als der zweite Wert?
- `<=`: Ist der erste Wert kleiner-gleich dem zweiten Wert?
- `>=`: Ist der erste Wert grösser-gleich dem zweiten Wert?

Beispiel 24 *Die folgende `if`-Anweisung gibt den Satz `Differenz zwischen Total und Kosten.` aus, wenn die beiden Variablen ungleiche Werte speichern.*

```
if (total != cost)
        System.out.println("Differenz zwischen Total und Kosten.");
```

Das logische AND, das logische OR und das logische NOT sind in Java ebenfalls vorhanden:

- Der Operator `!` wird verwendet, um ein logisches NOT auszudrücken. Der Ausdruck `!a` ist wahr, wenn `a` falsch ist und umgekehrt.

- Der Operator `&&` wird verwendet, um ein logisches AND auszudrücken. Der Ausdruck `a && b` ist wahr, wenn `a` *und* `b` beide wahr sind und sonst falsch.
- Der Operator `||` wird verwendet, um ein logisches OR auszudrücken. Der Ausdruck `a || b` ist wahr, wenn `a` *oder* `b` *oder* beide wahr sind und sonst falsch.

Ein logischer Operator kann mit einer *Wahrheitstabelle* beschrieben werden, der alle möglichen `true`-`false`-Kombinationen für die involvierten Ausdrücke auflistet. Für das logische NOT gibt es nur zwei mögliche Situationen:

a	!a
false	true
true	false

Beachten Sie, dass das logische NOT den gespeicherten Wert in `a` nicht verändert (wollen wir `a` ändern, so braucht es eine Zuweisung: `a = !a;`).

Für das logische AND (`&&`) und das logische OR (`||`) gibt es vier Situationen:

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Das logische NOT hat die höchste Priorität der drei logischen Operatoren, gefolgt vom logischen AND und vom logischen OR.

Beispiel 25 *Betrachten Sie das folgende Code-Fragment:*

```
boolean a = false, b = true, c = true;
boolean e = a || b && !c;
```

Zuerst wird `!c` *zu* `false` *ausgewertet, danach wird* `b && false` *zu* `false` *ausgewertet und letztlich wird* `a || false` *zu* `false` *ausgewertet.*

2.6.2 Die `if`-`else` Anweisung

Manchmal möchten wir in einem Programm etwas ausführen, wenn eine Bedingung wahr ist und etwas anderes, wenn die Bedingung falsch ist. Wir können für solche Fälle ein `else` zu der `if`-Anweisung hinzufügen.

Betrachten Sie hierzu folgendes Beispiel:

```
if (count <= MAX)
        count++;
else
        count = 0;
```

Wenn die Boolesche Bedingung `(count <= MAX)` wahr ist, so wird die erste Anweisung durchgeführt (die Variable `count` wird um `1` erhöht), andernfalls wird die zweite Anweisung durchgeführt (`count` wird der Wert `0` zugewiesen). Beachten Sie, dass zur Laufzeit genau eine der beiden Anweisungen ausgeführt wird.

Beispiel 26 *Das folgende Beispielprogramm fällt eine Entscheidung aufgrund der Variablen* `age` *(→ Video K-2.9).*

```
import java.util.Scanner;

public class AgeComment {

        /**
         * Gibt einen Kommentar auf Basis des Alters aus
         */
        public static void main(String[] args) {
                final int MINOR = 18;

                Scanner scan = new Scanner(System.in);
                System.out.println("Ihr Alter: ");
                int age = scan.nextInt();

                if (age < MINOR)
                        System.out.println("Jugend ist kein Fehler.");
                else
                        System.out.println("Alter ist kein Verdienst.");

                System.out.println("Alter ist eine Zahl.");
        }
}
```

Wenn das eingegebene Alter kleiner ist, als die im Quellcode definierte Konstante `MINOR`*, wird* `Jugend ist kein Fehler.` *ausgegeben. Andernfalls wird diese* `println` *Anweisung übersprungen und dafür* `Alter ist kein Verdienst.` *ausgegeben. In beiden Fällen wird danach der Satz* `Alter ist eine Zahl.` *ausgegeben.*

Mögliche Ein- und Ausgabe:

```
Ihr Alter:
42
Alter ist kein Verdienst.
Alter ist eine Zahl.
```

2.6.3 Blockanweisungen

Manchmal möchte man mehr als nur eine Anweisung ausführen, wenn die Bedingung in einer `if`-Anweisung wahr ist. In Java kann man einzelne Programmieranweisungen in einer `if-else` Anweisung mit *Blockanweisungen* ersetzen. Eine Blockanweisung ist eine Sammlung von Anweisungen umschlossen mit geschweiften Klammern (genau wie wir die Rümpfe von Klassen und Methoden umschliessen). Dies kann dann beispielsweise folgendermassen aussehen:

```
if (count > MAX) {
      System.out.println("Fehler");
      errorCount++;
}
```

Wenn der Ausdruck `(count > MAX)` wahr ist, werden in obiger `if`-Anweisung nun *beide* Anweisungen durchgeführt: Eine Ausgabe von `Fehler` und ein Inkrement der Variablen `errorCount`.

Beachten Sie, dass in Java der `if`-Teil, der `else`-Teil oder beide Teile aus Block-Anweisungen bestehen können. In der folgenden `if-else` Anweisung bestehen beispielsweise beide Teile aus Blockanweisungen:

```
if (count > MAX) {
      System.out.println("Fehler");
      errorCount++;
} else {
      System.out.println("Zählerstand: " + count);
      count += 2;
}
```

Kapitel 3

Klassen des Java API Verwenden (Teil 1)

Die Programmiersprache Java wird durch sogenannte *Klassenbibliotheken* (engl. *Class Libraries*) unterstützt, die wir bei der Programmierung nutzen können. Eine Klassenbibliothek besteht aus bereits erstelltem Quellcode, der eine enorme Menge an Funktionalität bieten kann und so die eigene Programmierung unterstützt.

Der Kompilierer von Java (etwas präziser: das JDK) beinhaltet solche Klassenbibliotheken – die sogenannte *Standard Class Library* auch *Java API* genannt (API steht für *Application Programming Interface*). Es gibt aber auch Klassenbibliotheken, die von Drittfirmen separat erworben werden können. In diesem Kapitel diskutieren wir den Gebrauch der Standardbibliothek von Java (verschiedene Klassen, die wir bereits verwendet haben (z.B. `System`, `Scanner` oder `String`), sind Teil der Standardbibliothek von Java). Insbesondere geht es dabei um den Prozess, Objekte aus bestehenden Klassen des Java APIs zu instanziieren und dann die Funktionalitäten der erzeugten Objekte zu verwenden.

3.1 Objekte aus Klassen Instanziieren

In Java repräsentiert eine Variable entweder einen primitiven Datentypen (z.B. `int`, `double`, `char` oder `boolean`) oder ein Objekt[1]. Genau wie bei Variablen für primitive Datentypen müssen auch Variablen, die ein Objekt speichern sollen, zunächst deklariert werden. Die Deklarationen von Objektvariablen haben die gleiche Struktur wie Deklarationen von Variablen für primitive Datentypen.

[1] Die Sprache Java ist also nicht bis zur letzten Konsequenz objektorientiert. Primitive Datentypen wie zum Beispiel ganze Zahlen oder Wahrheitswerte werden in Java nicht als Objekte verwaltet.

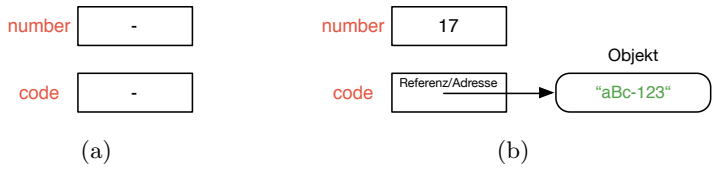

Abbildung 3.1: (a) Zwei deklarierte aber noch nicht initialisierte Variablen. (b) Den Variablen wurden Werte zugewiesen.

Die Klasse, die als Bauplan für das Objekt dient, kann dabei als Datentyp des Objektes interpretiert werden.

Beispiel 27 *Betrachten Sie die folgenden Deklarationen zweier Variablen mit den Bezeichnern* `number` *und* `code`*:*

```
int number;
String code;
```

Die erste Deklaration definiert eine Variable, die eine ganze Zahl speichern soll (der deklarierte Datentyp ist `int`*). Die zweite Deklaration definiert eine Variable, die ein Objekt vom Datentyp* `String` *speichern soll (der deklarierte Datentyp ist* `String`*).*

Im Gegensatz zu Variablen für primitive Datentypen, die den Wert der Variablen "direkt" speichern, enthält eine Objektvariable niemals das Objekt selber sondern eine "Adresse" (auch *Referenz* genannt), die auf das Objekt zeigt. Wir sagen: Die Variable referenziert das Objekt. *Objektvariablen* werden deshalb auch *Referenzvariablen* genannt (in diesem Skript werden beide Begriffe synonym verwendet).

Deklarierte Variablen besitzen zu Beginn keinen Wert. Wir sagen, die Variablen sind *nicht initialisiert* oder *undefiniert* (siehe Abb. 3.1 (a)). Es ist wichtig, dass Variablen immer initialisiert werden, bevor diese verwendet werden. Für Objektvariablen bedeutet dies, dass diese ein Objekt referenzieren müssen, bevor sie verwendet werden. In den meisten Situationen wird der Kompilierer eine Fehlermeldung auslösen, wenn versucht wird, auf eine Variable zuzugreifen, die noch undefiniert ist.

Obschon wir also bspw. mit `String code;` eine Objektvariable deklariert haben, existiert zu diesem Zeitpunkt noch *kein* Objekt vom Typ `String`, das von der Variable mit dem Bezeichner `code` referenziert wird. Daher kann die Funktionalität des

Objektes `code` auch noch nicht verwendet werden.

In Klassen sind die Eigenschaften und die Funktionalitäten der Objekte programmiert – und zwar mit Variablen und Methoden. Der Vorgang, ein neues Objekt aus einer Klasse zu generieren, heisst *Instanziierung* (man sagt denn auch, ein Objekt ist eine *Instanz* einer bestimmten Klasse).

Um ein Objekt zu instanziieren, wird der sogenannte *`new`-Operator* verwendet, der ein Objekt aus dem entsprechenden Bauplan – also der Klasse – generiert. Der `new`-Operator instanziiert Objekte, indem der sogenannte *Konstruktor* der entsprechenden Klasse aufgerufen wird. Ein Konstruktor ist eine spezielle Methode, die immer den gleichen Bezeichner besitzt, wie die Klasse selbst.

Zum Beispiel können wir mit

```
new String("...");
```

den Konstruktor der Klasse `String` aufrufen, der uns dann ein neues Objekt aus der Klasse `String` einrichtet. In diesem Beispiel kann dem Konstruktor `String` ein Parameter mitgegeben werden, nämlich eine Zeichenkette `"..."`, welche das neu instanziierte Objekt speichern soll.

Der Konstruktor hat noch eine zweite Aufgabe: Er erzeugt nicht nur das Objekt, er gibt auch die Adresse auf das instanziierte Objekt zurück, die dann einer Variablen – zum Beispiel der Variablen `code` – zugewiesen werden kann.

Beispiel 28 *Die folgenden zwei Programmieranweisungen weisen den zuvor deklarierten Variablen* ***`number`*** *und* ***`code`*** *Werte zu:*

```
number = 17;
code = new String("aBc-123");
```

Da eine Objektvariable nur die Adresse des Objektes enthält, kann eine solche Variable auch als *Zeiger* (engl. *Pointer*) aufgefasst werden, der auf den Speicherort des entsprechenden Objektes im Hauptspeicher zeigt (siehe Abb. 3.1 (b)).

Analog zur Deklaration und Initialisierung von Variablen mit primitivem Datentyp – wie zum Beispiel `int i = 0;` – kann die Deklaration einer Objektvariablen und

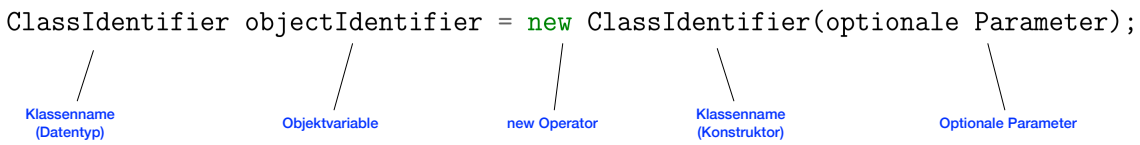

Abbildung 3.2: Deklaration und Instanziierung eines Objektes.

deren Instanziierung auch in einer Anweisung vollzogen werden:

```
String title = new String("Programmieren mit Java");
```

Bis auf ein paar Ausnahmen folgt das Deklarieren und Instanziieren von Objekten aus Klassen immer dem in Abb. 3.2 gezeigten Muster.

Beispiel 29 *In den folgenden vier Anweisungen deklarieren und instanziieren wir vier Objektvariablen* `hello`, `random`, `reader` *und* `p1`. *Diese Variablen sind vom Typ* `String`, `Random`, `Scanner` *und* `Point` *(alles Klassen des Java APIs). Beachten Sie, dass bei einigen Konstruktoren Parameter nötig sind und bei anderen nicht.*

```
String hello = new String("Hello");
Random random = new Random();
Scanner reader = new Scanner(System.in);
Point p1 = new Point(2, 5);
```

Obschon eine Zeichenkette kein primitiver Datentyp ist, kann man diese trotzdem instanziieren, wie es bei primitiven Datentypen üblich ist – nämlich mit direkter Wertangabe (also ohne `new`-Operator und ohne Konstruktor):

```
String city = "Bern";
```

Diese Initialisierung der Variablen `city` ist eine Abkürzung, die äquivalent ist zu:

```
String city = new String("Bern");
```

Diese Abkürzung ist nur bei der Klasse `String` (und bei sogenannten *Wrapper-Klassen*, die wir später behandeln) möglich, da Zeichenketten sehr häufig vorkommen und ein relativ einfaches Konzept darstellen.

Objektvariablen können in Java immer auch mit `null` initialisiert werden. Das Wort `null` ist ein reserviertes Wort in Java und definiert das "leere" Objekt. Dies kann

hilfreich sein, wenn der Kompilierer verlangt, dass ein Objekt initialisiert sein muss, Sie aber noch nicht wissen, mit welchen konkreten Inhalten das Objekt instanziiert werden soll (z.B. weil die entsprechenden Angaben erst zur Laufzeit des Programmes eingegeben werden).

Beispiel 30 *Beispielsweise können wir ein Objekt vom Typ* `String` *oder ein Objekt vom Typ* `Point` *(oder ein Objekt jedes anderen Typs) mit* `null` *initialisieren:*

```
String name = null;
Point point = null;
```

Die Variablen `name` *und* `point` *vom Typ* `String` *und* `Point` *sind nun deklariert und auch initialisiert (je mit dem leeren Objekt* `null`*).*

3.1.1 Methoden von Instanziierten Objekten Verwenden

Sobald ein Objekt instanziiert worden ist, können wir den *Punkt-Operator* verwenden, um auf die Funktionalitäten des Objektes zuzugreifen. Die verfügbaren Funktionalitäten eines Objektes sind in der entsprechenden Klasse des Objektes als Methoden programmiert. Der Punkt-Operator wird direkt hinter die Objektvariable angehängt und danach folgt der Name der Methode, die von diesem Objekt ausgeführt werden soll. Ein Aufruf einer Methode kann als Aufforderung an ein Objekt verstanden werden, einen bestimmten Dienst auszuführen.

Beispiel 31 *Die Klasse* `String` *besitzt zum Beispiel eine Methode* `length`*. Um die Methode* `length` *aufzurufen, können wir den Punkt-Operator auf der Variablen* `code` *(vom Typ* `String`*) ausführen:*

```
code.length();
```

Die Methode `length()` erwartet keine Parameter – die Klammern sind aber dennoch nötig, um dem Kompilierer anzugeben, dass es sich bei dem Bezeichner `length` um eine Methode handelt.

Einige Methoden produzieren eine Rückgabe (engl. *return value*), sobald alle Anweisungen der Methode ausgeführt worden sind. Dies ist nicht zwingend – es existieren Methoden (bzw. man kann Methoden programmieren), die keine Rückgabe erzeugen.

Die Methode `length` der Klasse `String` evaluiert die Länge der gespeicherten Zeichenkette (zählt also die Anzahl gespeicherter Zeichen) und gibt diesen Wert zurück.

Referenziert die Variable `code` aktuell die Zeichenkette `"aBc-123"`, gibt die Anweisung `code.length();` den Wert `7` zurück.

Oft wollen wir die Rückgabe einer Methode im weiteren Verlauf unseres Programmes verwenden. Zu diesem Zweck können wir die Rückgabe speichern, indem wir diese z.B. einer Variablen zuweisen:

```
int codeLength = code.length();
```

In diesem Beispiel wird der zurückgegebene Wert der Methode `length` der Variablen `codeLength` vom Typ `int` zugewiesen.

3.2 Das Java API

Die Klasse `String`, die wir in den bisherigen Beispielen verwendet haben, ist *kein* Bestandteil der Programmiersprache Java. Sie ist Teil der *Java Standard Class Library*, die in jeder Java Entwicklungsumgebung gefunden werden kann. Die verfügbaren Klassen dieser Bibliothek wurden durch Mitarbeiterinnen und Mitarbeiter von *Sun Microsystems* bzw. *Oracle* programmiert (also von den selben Menschen, die Java entwickelt haben bzw. die Sprache Java weiterentwickeln).

Die Klassen der Standardbibliothek stellen Lösungen für verschiedene Aufgaben zur Verfügung, die beim Programmieren häufig erledigt werden müssen. Insbesondere kann durch die Verwendung dieser Klassen Zeit gespart werden, da nicht für jedes Teilproblem *eigener* Quellcode erstellt werden muss. Tatsächlich wird man oft "abhängig" von dieser Bibliothek und nimmt diese in Java geschriebenen Programme fälschlicherweise als Teil der Programmiersprache wahr.

Die Java Standardbibliothek besteht aus verschiedenen Gruppen von Klassen, die miteinander verwandt sind – den sogenannten *Packages*. Zum Beispiel befinden sich im *Package* `java.sql` verschiedene Klassen, welche den Zugriff auf Daten, die in einer Datenbank gespeichert sind, ermöglichen. Die *Packages* nennt man häufig auch Java APIs (*Application Programming Interfaces*) und die gesamte *Java Standard Class Library* wird deshalb oftmals einfach *Java API* genannt.

In der folgenden Tabelle sind einige *Packages* kurz beschrieben:

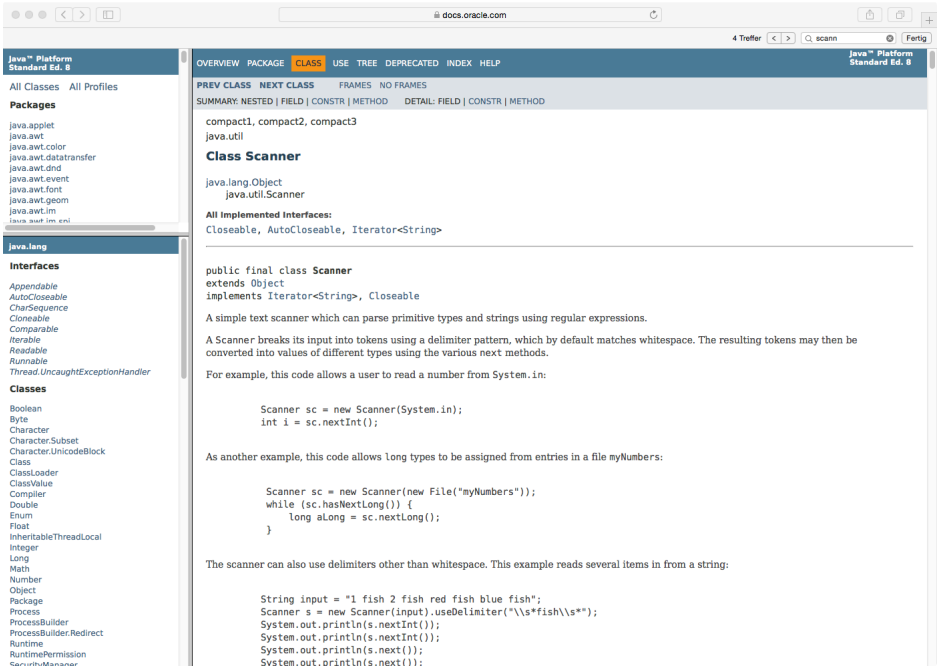

Abbildung 3.3: Beispielseite der Java API Dokumentation.

Package	**Ermöglicht...**
`java.lang`	... generelle Unterstützung bei der Programmierung.
`java.beans`	... die Definition von Komponenten, die kombiniert werden können.
`java.io`	... vielfältige Input-Output Funktionen (z.B. über das Dateisystem).
`java.math`	... das Durchführen mathematischer Operationen mit hoher Präzision.
`java.net`	... die Kommunikation über Netzwerke.
`java.rmi`	... das Erstellen von Programmen, die auf verteilten Systemen laufen sollen.
`java.security`	... das Erzwingen von Sicherheitsrestriktionen.
`java.sql`	... die Interaktion mit Datenbanken.
`java.text`	... vielfältige Textformatierungen.
`java.util`	... die Anwendung allgemeiner Hilfsmittel.
`java.xml`	... die Verarbeitung von XML Dokumenten.

Die *online Java API Dokumentation*[2] ist von grossem Wert, da diese ausführliche Beschreibungen für jede Klasse des Java APIs enthält. Zum Beispiel sind alle Methoden der jeweiligen Klassen aufgelistet und beschrieben (oftmals auch mit Beispielen). In Abb. 3.3 ist ein Beispiel gezeigt (für die Klasse `Scanner`, die wir gleich genauer betrachten werden).

Die Klassen des *Packages* `java.lang` sind automatisch verfügbar, sobald man ein eigenes Java Programm schreibt. Wenn Sie hingegen eine Klasse aus einem anderen

[2]https://docs.oracle.com/en/java/javase/14/docs/api/

Package verwenden wollen, müssten Sie eigentlich den vollen Namen inkl. Angabe des *Packages* verwenden:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

In der Regel werden wir die entsprechende Klasse oder das gesamte zugehörige *Package* aber in unseren Quellcode importieren, so dass der Klassenname ausreicht (`Scanner` statt `java.util.Scanner`). Nach dem reservierten Wort `import` geben wir hierzu das entsprechende *Package* und dann die Klasse, die in unserem Programm nutzbar sein soll, an. Um z.B. die Klasse `Scanner` aus dem Package `java.util` verwenden zu können, muss folgende `import` Anweisung zu unserem Quellcode hinzugefügt werden:

```
import java.util.Scanner;
```

Importanweisungen haben in Java immer vor dem Klassenkopf zu erfolgen.

Werden mehr als zwei Klassen aus einem bestimmten *Package* benötigt, so verwendet man oft einen *Asterisk* (`*`) um anzugeben, dass *alle* Klassen des entsprechenden *Packages* importiert werden sollen:

```
import java.util.*;
```

Beachten Sie, dass das Java API Hunderte von Klassen für unterschiedlichste Zwecke bereitstellt. Wir verwenden in den restlichen Abschnitten dieses Kapitels nur ein paar wenige Klassen des Java APIs, um uns exemplarisch mit wichtigen Konzepten der Programmierung und der Objektorientierung vertraut zu machen.

3.3 Die Klasse `String`

Wir haben die Klasse `String` bereits beispielhaft verwendet. Nun wollen wir die Klasse noch etwas genauer betrachten. Die Klasse `String` gehört zum *Package* `java.lang` – muss also nicht selber importiert werden. In folgender Liste sind ein paar nützliche Methoden der Klasse `String` zusammengefasst[3].

[3] In diesem Skript werden wir immer nur einige der verfügbaren Methoden betrachten – wenn Sie weitere Informationen benötigen, sollten Sie die entsprechende Beschreibung in der online API Dokumentation studieren.

Die Listen mit den Methoden von Klassen sind in diesem Skript immer folgendermassen aufgebaut:

- `Datentyp Methodenbezeichner(Datentyp Parameterbezeichner)`: Beschreibung der Methode

Hierbei entspricht der erste Datentyp dem Datentyp der Rückgabe und der zweite Datentyp (falls vorhanden) dem Datentyp des oder der Parameter. Die einzige Ausnahme bildet immer die Beschreibung des Konstruktors, der keine Rückgabe erzeugt und für den deshalb auch kein entsprechender Datentyp angegeben werden kann.

- `String(String p)`: Der Konstruktor der Klasse instanziiert zusammen mit dem `new`-Operator ein neues Zeichenketten Objekt mit den Zeichen, die in `p` übergeben werden.
- `char charAt(int index)`: Gibt das Zeichen an Position `index` einer Zeichenkette zurück.
- `String concat(String p)`: Gibt ein neues `String` Objekt zurück, das aus dieser Zeichenkette konkateniert mit `p` besteht.
- `boolean equals(String p)`: Gibt `true` zurück, wenn diese Zeichenkette die gleichen Zeichen in der gleichen Reihenfolge wie `p` speichert (andernfalls wird `false` zurückgegeben).
- `boolean equalsIgnoreCase(String p)`: Funktioniert gleich wie `equals`, aber ohne die Gross- und Kleinschreibung zu beachten.
- `int length()`: Gibt die Anzahl Zeichen dieser Zeichenkette zurück.
- `String replace(char oldChar, char newChar)`: Gibt eine Kopie dieser Zeichenkette zurück, in der alle Zeichen `oldChar` mit `newChar` ersetzt werden.
- `String substring(int offset, int endIndex)`: Gibt eine Kopie dieser Zeichenkette zurück, in der alle Zeichen vor `offset` und ab `endIndex` abgeschnitten werden.
- `String toLowerCase()`: Gibt eine Kopie dieser Zeichenkette zurück, in der alle Grossbuchstaben mit dem zugehörigen Kleinbuchstaben ersetzt werden.
- `String toUpperCase()`: Gibt eine Kopie dieser Zeichenkette zurück, in der alle Kleinbuchstaben mit dem zugehörigen Grossbuchstaben ersetzt werden.

Einige der Methoden der Klasse `String` verwenden einen *Index*, um ein bestimmtes Zeichen zu referenzieren. Ein Zeichen in einem `String` Objekt kann über dessen Index spezifiziert werden. Der Index des ersten Zeichens in einer Zeichenkette ist `0`, das nächste Zeichen hat Index `1`, usw. Im `String` Objekt `"Pferd"` hat das Zeichen `P` also den Index `0` und der Index des letzten Buchstabens `d` ist `4`.

Gibt man in Java mit zwei Indizes Grenzen an, so gilt die Faustregel, dass die untere Grenze inklusive ist und die obere Grenze nicht mehr zur Auswahl gehört. Referenziert also zum Beispiel die Variable `s` die Zeichenkette `"Hallo"`, so gibt beispielsweise `s.substring(1, 3);` die Zeichenkette `"al"` zurück.

Wurde ein `String` Objekt instanziiert, so kann weder dessen Grösse noch können einzelne Zeichen verändert werden. Objekte, deren Zustand sich nicht verändern lässt, heissen *unveränderlich* (engl. *immutable*). Da `String` Objekte unveränderlich sind, können Veränderungsmethoden (z.B. die Methode `replace`) nur ein neues `String` Objekt zurückgeben, welches das Resultat von Veränderungen an der ursprünglichen Zeichenkette sind – die ursprüngliche Zeichenkette bleibt dabei aber unverändert!

Beispiel 32 *In folgendem Programm werden einige Methoden der Klasse* ***`String`*** *demonstriert (→ Video K-3.1):*

```
public class StringOperations {

        /**
         * Demonstriert einige Methoden der Klasse String
         */
       public static void main(String[] args) {

               String original = "Impossible is temporary";
               System.out.println("Original: " + original);

               String string1 = original.concat(", impossible is nothing.");
               String string2 = original.replace('e', 'X');
               String string3 = original.substring(3, 7);
               String string4 = original.toUpperCase();

               System.out.println("Veränderung 1: " + string1);
               System.out.println("Veränderung 2: " + string2);
               System.out.println("Veränderung 3: " + string3);
               System.out.println("Veränderung 4: " + string4);
               System.out.println("Länge der Veränderung 3: " + string3.length());

               if (original.equals("Impossible is temporary"))
                       System.out.println("Original ist unverändert: " + original);
       }
}
```

Die Ausgabe dieses Programmes lautet:

```
Original: Impossible is temporary
Veränderung 1: Impossible is temporary, impossible is nothing.
Veränderung 2: ImpossiblX is tXmporary
Veränderung 3: ossi
Veränderung 4: IMPOSSIBLE IS TEMPORARY
Länge der Veränderung 3: 4
Original ist unverändert: Impossible is temporary
```

Beachten Sie insbesondere, dass das ursprüngliche `String` *Objekt* `original` *nicht verändert worden ist.*

3.4 Die Klasse `Scanner`

Im letzten Kapitel haben wir die Klasse `Scanner` bereits verwendet, um Eingaben vom Benutzer via Tastatur entgegenzunehmen[4]. Die Eingabe kann aber auch aus anderen Quellen stammen (z.B. aus einer Datei). Die Klasse `Scanner` kann zudem dazu verwendet werden, um ein `String` Objekt (also eine Zeichenkette) in einzelne Teile zu zerlegen (z.B. um einen eingegebenen Text in einzelne Wörter zu zerlegen).

Die folgende Liste fasst einige wichtige Methoden der Klasse `Scanner` zusammen:

- `Scanner(InputStream source)`: Der Konstruktor der Klasse um ein neues `Scanner` Objekt zu instanziieren und Daten von einer bestimmten Quelle einzulesen (z.B. `System.in`). Analog dazu existieren Konstruktoren, die eine Datei (`File source`) oder eine Zeichenkette (`String source`) als Parameter erwarten.

- `String next()`: Gibt das nächste Element der Eingabe als `String` Objekt zurück. Hierbei wird "Leerraum" (z.B. ein Leerschlag oder ein Tabulator) als Trennzeichen verwendet.

- `String nextLine()`: Gibt die komplette nächste Zeile der Eingabe (bis zum ersten Zeilenumbruch) als `String` Objekt zurück

- `int nextInt()`: Gibt das nächste Element der Eingabe als Wert vom Typ `int` zurück. Analog dazu existieren z.B. die Methoden `nextDouble()` oder

[4] Die Klasse `Scanner` ist Teil des *Packages* `java.util` und muss importiert werden.

`nextBoolean()`. Diese Methoden erzeugen einen Fehler, wenn das zu lesende Element nicht mit dem angegebenen Datentyp übereinstimmt.

- `Scanner useDelimiter(String pattern)`: Erlaubt die Definition eines eigenen Trennzeichens `pattern`, mit dem die Eingabe in einzelne Teile zerlegt werden soll.
- `boolean hasNext()`: Gibt `true` zurück, wenn das `Scanner` Objekt ein weiteres Element lesen kann (andernfalls `false`).

Ist ein Objekt vom Typ `Scanner` instanziiert, können die Methoden wie gewohnt mit dem Punkt-Operator aufgerufen werden. Folgende Anweisung führt zum Beispiel die Methode `nextLine` auf dem instanziierten Objekt `scan` aus:

```
Scanner scan = new Scanner(System.in);
String line = scan.nextLine();
```

Beispiel 33 *Folgendes Programm verwendet ein Objekt vom Typ* `Scanner` *(→ Video K-3.2).*

```
import java.util.Scanner;

public class ParkingFee {

        /**
         * Berechnet die Parkgebühr auf Basis der Parkzeit
         */
        public static void main(String[] args) {

                final double FEE_PER_HOUR = 2.5; // Parkgebühr für die ersten 5 Stunden
                final int MAX_HOURS = 5;
                final double FACTOR = 1.5; // 1.5-fache Parkgebühr ab 5 Stunden Parkzeit

                Scanner scan = new Scanner(System.in);

                double fee = 0.0;
                System.out.print("Parkzeit in ganzen Stunden eingeben: ");
                int parkingTime = scan.nextInt();

                if (parkingTime <= MAX_HOURS)
                        fee = parkingTime * FEE_PER_HOUR;
                else {
                        int additionalHours = parkingTime - MAX_HOURS;
                        fee = MAX_HOURS * FEE_PER_HOUR;
                        fee += additionalHours * FEE_PER_HOUR * FACTOR;
                }
                System.out.println("Zu bezahlen: " + fee + " CHF");
        }
}
```

In diesem Beispiel soll der Benutzer die Parkzeit in Stunden eingeben und dann wird die entsprechende Parkgebühr berechnet. Ab der sechsten Stunde wird die `1.5`-fache Parkgebühr erhoben (hierzu wird ein `if-else` eingesetzt). Eine mögliche Ein- und Ausgabe sieht folgendermassen aus:

```
Parkzeit in ganzen Stunden eingeben: 6
Zu bezahlen: 16.25 CHF
```

Hinweis: Das Programm `ParkingFee` stürzt ab, wenn der Benutzer nicht eine ganze Zahl eingibt (wenn er also zum Beispiel `4.5` oder `zwei` eingibt). Das Abfangen solcher Probleme behandeln wir später.

Ein `Scanner` Objekt verwendet den Leerraum in der Eingabe (also Leerzeichen, Tabulatoren oder Zeilenumbrüche), um die Eingabe in Einzelteile (sogenannte *Token*) zu zerlegen. Beim Aufruf der Methode `next` wird der nächste *Token* eingelesen und von der Methode zurückgegeben. Das heisst, ist die Eingabe z.B. eine Reihe von Wörtern separiert durch Leerschläge, gibt jeder Aufruf von `next` das nächste Wort dieser Eingabe zurück.

Die Zeichen, welche definieren, wie eine Eingabe in einzelne *Token* zerlegt werden soll, nennt man *Trennzeichen* (engl. *Delimiter*). Mit der Methode `useDelimiter` kann die Standardzerlegung geändert werden. Möchten wir z.B. ein Komma und einen Leerschlag als Trennzeichen definieren, führen wir auf einem instanziierten Objekt `scan` folgenden Methodenaufruf aus:

```
scan.useDelimiter(", ");
```

Beispiel 34 *In folgendem Beispiel werden zwei `Scanner` Objekte instanziiert – eines zum Einlesen via Tastatur und eines zum Zerlegen der Eingabe in einzelne Token. Das heisst, das erste Objekt wird mit `System.in` und das zweite Objekt mit einem `String` als Parameter instanziiert (→Video K-3.3):*

```
import java.util.Scanner;

public class StringTokenizer {

        /**
         * Zerlegt Eingaben getrennt durch Kommas
         */
        public static void main(String[] args) {
```

```
            Scanner scan = new Scanner(System.in);
            System.out.print("Geben Sie drei Namen getrennt durch Kommas ein: ");
            String names = scan.nextLine();

            Scanner lineScanner = new Scanner(names);
            lineScanner.useDelimiter(", ");

            String name1 = lineScanner.next();
            String name2 = lineScanner.next();
            String name3 = lineScanner.next();

            System.out.println("Name 1: " + name1);
            System.out.println("Name 2: " + name2);
            System.out.println("Name 3: " + name3);
        }
}
```

Eine mögliche Ein- und Ausgabe sieht folgendermassen aus:

```
Geben Sie drei Namen getrennt durch Kommas ein: Maxime, Emilie, Noah
Name 1: Maxime
Name 2: Emilie
Name 3: Noah
```

3.5 Die Klasse `Random`

In Programmen müssen relativ häufig zufällige Entscheidungen getroffen werden (z.B. Werfen eines Würfels in einem Spiel oder zufällige Initialisierung bei einer Simulation). Die Klasse `Random` repräsentiert einen Zufallszahl-Generator, ist ebenfalls Teil des *Packages* `java.util` und muss deshalb importiert werden.

Die folgende Liste fasst vier wichtige Methoden der Klasse `Random` zusammen:

- `Random()`: Der Konstruktor der Klasse (ohne Parameter) – instanziiert ein neues Objekt vom Typ `Random` zur Erzeugung von Zufallszahlen.
- `float nextFloat()`: Gibt eine zufällige Zahl zwischen `0.0` (inklusive) und `1.0` (exklusive) zurück.
- `int nextInt()`: Gibt eine zufällige ganze Zahl zwischen `-2147483648` und `2147483647` zurück.
- `int nextInt(int max)`: Gibt eine zufällige ganze Zahl zwischen `0` und `max-1` zurück.

Diese drei Methoden reichen tatsächlich aus, um Zufallszahlen aus vielfältigen Intervallen zu generieren. Um zum Beispiel eine zufällige ganze Zahl von 1 bis 6 zu erhalten, können wir zur Rückgabe von `nextInt(6);` noch 1 addieren. Um eine Gleitkommazahl in einem grösseren Bereich als `[0,1[` zu erhalten, kann mit einer Multiplikation das Intervall vergrössert werden (z.B. erzeugt der Ausdruck `100 * nextFloat();` eine zufällige Gleitkommazahl im Intervall `[0,100[`).

Beispiel 35 *In folgendem Programm werden mit Hilfe der Methoden der Klasse* ***Random*** *zufällige Zahlen in unterschiedlichen Intervallen erzeugt.*

```
import java.util.Random;

public class RandomNumbers {

    /**
    * Demonstriert den Gebrauch der Klasse Random
    */
    public static void main(String[] args) {
        Random random = new Random();
        int num1;
        double num2;

        num1 = random.nextInt();
        System.out.println("Eine zufällige ganze Zahl: " + num1);
        num1 = random.nextInt(2);
        System.out.println("0 oder 1?: " + num1);
        num1 = random.nextInt(10);
        System.out.println("Zufallszahl zwischen 0 und 9: " + num1);
        num1 = random.nextInt(10) + 1;
        System.out.println("Zufallszahl zwischen 1 und 10: " + num1);
        num1 = random.nextInt(21) - 10;
        System.out.println("Zufallszahl zwischen -10 und +10: " + num1);
        num2 = random.nextFloat(); // 0.0 bis 0.99999999
        System.out.println("Zufallszahl aus [0.0, 1.0[: " + num2);
        num2 = random.nextFloat() * 5;  // 0.0 bis 4.99999999
        System.out.println("Zufallszahl aus [0.0, 5.0[: " + num2);
    }
}
```

Eine mögliche Ausgabe des Programmes lautet:

```
Eine zufällige ganze Zahl: 424791840
0 oder 1?: 0
Zufallszahl zwischen 0 und 9: 0
Zufallszahl zwischen 1 und 10: 6
Zufallszahl zwischen -10 und +10: -3
Zufallszahl aus [0.0, 1.0[: 0.777391791343689
Zufallszahl aus [0.0, 5.0[: 2.8571033477783203
```

Beispiel 36 *Die Klasse* ***GuessingGame*** *demonstriert den Einsatz der beiden Klassen* ***Scanner*** *und* ***Random*** *(→ Video K-3.4):*

```
import java.util.Random;
import java.util.Scanner;

public class GuessingGame {

        /**
         * Spielt 1, 2 oder 3 mit dem Benutzer
         */
        public static void main(String[] args) {
                final int MAX = 3;
                int answer, guess;

                Scanner scan = new Scanner(System.in);
                Random random = new Random();

                answer = random.nextInt(MAX) + 1;

                System.out.print("Gesucht ist eine Zahl zwischen 1 und " + MAX +
                                ". Raten Sie! ");
                guess = scan.nextInt();

                if (guess == answer)
                        System.out.println("Glück gehabt. Das ist richtig...");
                else {
                        System.out.println("Leider daneben.");
                        System.out.println("Die gesuchte Zahl war " + answer + ".");
                }
        }
}
```

Die Idee des Programmes ist, dass der Computer eine Zufallszahl zwischen 1 und 3 generiert und der Variablen ***answer*** *zuweist. Danach wird die geratene Zahl des Benutzers eingelesen und der Variablen* ***guess*** *zugewiesen. Wenn die getippte Zahl des Benutzers der zuvor generierten Zufallszahl entspricht, wird ein entsprechender Text ausgegeben. Wenn nicht, werden zwei Ausgaben generiert (gesammelt in einer Blockanweisung).*

3.6 Die Klasse `DecimalFormat`

Es ist oftmals nötig, Text zu formatieren, so dass dieser korrekt oder "schön" angezeigt wird. Das Java API enthält einige Klassen, welche verschiedene Formatierungsmöglichkeiten anbieten. Wir betrachten im Folgenden die Klasse `DecimalFormat`, welche das Formatieren von Zahlen nach einem benutzerdefinierten Muster ermöglicht (diese Klasse ist Bestandteil des *Packages* `java.text` und muss importiert werden).

Die drei wichtigsten Methoden der Klasse `DecimalFormat` sind:

- `DecimalFormat(String p)`: Der Konstruktor der Klasse nimmt als Parameter eine Zeichenkette `p` entgegen, die das Muster repräsentiert, nach dem die Ausgaben formatiert werden sollen.
- `String format(double num)`: Formatiert die Zahl `num` gemäss dem aktuell gültigen Muster.
- `void applyPattern(String p)`: Definiert das Formatierungsmuster neu[5].

Die Zeichenkette `p`, die dem Konstruktor `DecimalFormat` oder der Methode `applyPattern` als Muster übergeben wird, kann sehr ausgeklügelt sein und verschiedene Sonderzeichen beinhalten. Für einfache Formatierungen sind die Null `"0"`, die Interpunktionszeichen `"."` und `","` sowie das Zeichen `"#"` wichtig.

- Die Zeichen `"0"` und `"#"` dienen als Platzhalter einzelner Ziffern an bestimmten Positionen in einer Dezimalzahl. Grundsätzlich bedeutet eine `"0"`, dass die Ziffer an dieser Position immer angezeigt wird, während ein `"#"` bedeutet, dass an dieser Position führende und nachfolgende Nullen nicht angezeigt werden.
- Der Punkt `"."` definiert die Grenze zum Nachkommabereich und das Komma `","` das Tausender-Trennzeichen.

Beispiel 37 *Zum Beispiel definiert die Zeichenkette `"#,#00.0#"` folgendes Formatierungsmuster:*

- *Verwendung des Tausender-Trennzeichens*
- *mindestens 2 ganzzahlige Ziffern (erzwungen mit `"00."`)*
- *mindestens 1 Nachkommastelle (erzwungen mit `".0"`)*
- *maximal 2 Nachkommastellen*

Mit diesem Muster werden also bspw. mit der Methode `format` folgende Formatierungen vorgenommen:

- *`1234.564` → `"1'234.56"`; `14.032` → `"14.03"`; `1.803` → `"01.8"`; `17` → `"17.0"`*

[5]Hinweis: `void` ist ein spezieller Rückgabetyp, den wir im nächsten Kapitel behandeln.

Beispiel 38 *Das Programm* `Circle` *berechnet die Fläche und den Umfang eines Kreises mit einem Radius, der vom Benutzer des Programmes eingegeben wird. Die Resultate der Berechnung werden formatiert ausgegeben (→ Video K-3.5).*

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class Circle {

        /**
         * Demonstriert den Nutzen der Klasse DecimalFormat
         */
        public static void main(String[] args) {

                final double PI = 3.14159265359;
                // Anzeigen von min. 1 und max. 2 Nachkommastellen
                DecimalFormat formater = new DecimalFormat("0.0#");

                Scanner scan = new Scanner(System.in);
                System.out.print("Radius Eingeben: ");
                double radius = scan.nextDouble();

                double area = radius * radius * PI;
                double circumference = 2 * radius * PI;
                System.out.println("Kreisfläche: " + formater.format(area));
                System.out.println("Kreisumfang: " +  formater.format(circumference));
        }
}
```

Eine mögliche Ein- und Ausgabe des Programmes lautet:

```
Radius: 2.3
Kreisfläche: 16.62
Kreisumfang: 14.45
```

Die Formatierungsmuster der Klasse `DecimalFormat` dürfen auch weitere Zeichen enthalten. Zum Beispiel:

```
DecimalFormat formater = new DecimalFormat("CHF 000.00");
```

Das Objekt `formater` formatiert dann eine Zahl wie bspw. `4.5` zu `CHF 004.50`.

Zudem erlaubt die Klasse `DecimalFormat` auch die Formatierung als Prozentwert:

```
DecimalFormat formater = new DecimalFormat("0.000%");
```

Dieses Objekt formatiert eine Zahl wie bspw. `0.5562` zu `55.620%`.

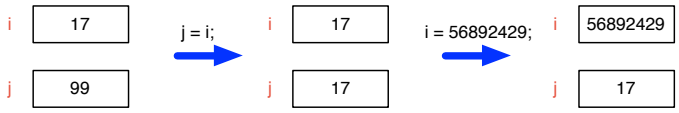

Abbildung 3.4: Kopieren und Verändern von Variablen im Fall primitiver Datentypen.

3.7 Kopieren von Objekten – Aliase

Wie zu Beginn dieses Kapitels erläutert, speichern Objektvariablen Adressen (oder Referenzen) und nicht das Objekt selber. Das schafft eine interessante Situation: Auf ein Objekt kann über mehrere Referenzvariablen zugegriffen werden.

Betrachten wir zunächst den Effekt einer Variablenzuweisung und des Kopierens im Fall von primitiven Datentypen.

Beispiel 39 *Wir deklarieren und initialisieren zwei* `int` *Variablen* `i` *und* `j` *mit den Werten* `17` *und* `99`*:*

```
int i = 17;
int j = 99;

j = i;
```

Danach weisen wir eine Kopie des Wertes, der in `i` *gespeichert ist, der Variablen* `j` *zu. Der ursprüngliche Wert* `99` *der Variablen* `j` *wird mit dem Wert* `17` *überschrieben. Die beiden Variablen* `i` *und* `j` *beinhalten nun den Wert* `17`*, befinden sich aber weiterhin an zwei separaten und unabhängigen Stellen im Speicher (siehe Abb. 3.4). Dies wird insbesondere deutlich, wenn wir nun eine der beiden Variablen durch eine Zuweisung verändern. Zum Beispiel mit*

```
i = 56892429;
```

Nach dieser Zuweisung speichert die Variable `j` *weiterhin den Wert* `17`*.*

Im folgenden Beispiel machen wir im Wesentlichen das Selbe wie oben – aber dieses mal mit Objektvariablen. Zur Illustration verwenden wir hierzu Objektvariablen der Klasse `Point` aus dem *Package* `java.awt` (man könnte hierzu aber (fast) jede andere Klasse aus dem Java API verwenden).

Objekte der Klasse `Point` repräsentieren Punkte, die eine Position in einem (x, y)-Koordinatenraum darstellen. Der Konstruktor der Klasse `Point` erwartet zwei ganze

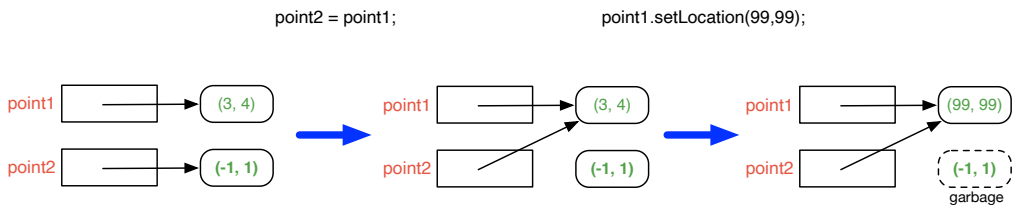

Abbildung 3.5: Kopieren und Verändern von Variablen im Fall von Objektvariablen.

Zahlen als Parameter, welche die x- und die y-Koordinate des Punktes repräsentieren. Mit folgender Programmieranweisung instanziieren wir z.B. einen Punkt an Position `(3, 4)`:

```
Point point1 = new Point(3, 4);
```

Mit der Methode `setLocation`, welche ebenfalls zwei ganze Zahlen als Parameter entgegennimmt, können wir die Position eines instanziierten Objektes vom Typ `Point` nachträglich anpassen. Zum Beispiel können wir den eben gerade instanziierten Punkt `point1` an die Position `(99, 99)` verschieben:

```
point1.setLocation(99, 99);
```

Beispiel 40 *Wir deklarieren und initialisieren zunächst zwei Objektvariablen* ***point1*** *und* ***point2*** *vom Typ* ***Point*** *und weisen dann eine Kopie des Wertes, der in* ***point1*** *gespeichert ist, der Variablen* ***point2*** *zu:*

```
Point point1 = new Point(3, 4);
Point point2 = new Point(-1, 1);

point2 = point1;
```

Der ursprüngliche Wert der Variablen ***point2*** *wird überschrieben mit dem Wert, der in der Objektvariablen* ***point1*** *gespeichert ist. Objektvariablen beinhalten aber nur eine Adresse des Objektes (und nicht das Objekt selber). Das heisst, in diesem Fall wird lediglich die Adresse auf das Objekt, aber nicht das Objekt selber kopiert (siehe auch Abb. 3.5).*

Ursprünglich haben die beiden Variablen zwei unterschiedliche Objekte referenziert – nach der Zuweisung ***point2 = point1;*** *beinhalten aber beide Variablen die gleiche Adresse und zeigen demnach auf das gleiche Objekt.*

Die beiden Variablen `point1` und `point2` sind jetzt *Aliase* voneinander, da diese auf dasselbe Objekt zeigen. Aliase können nützlich sein, sollten aber sorgfältig verwaltet werden. Der wichtigste Effekt von Aliase ist: Wenn wir eines der beiden Objekte ändern, so ändern wir die andere Referenz genauso (weil sie auf ein und dasselbe Objekt zeigen).

Dies wird insbesondere deutlich, wenn wir nun eine der beiden Variablen verändern. Zum Beispiel mit

```
point1.setLocation(99, 99);
```

Nach Ausführung dieser Programmieranweisung sind die Koordinaten der Objektvariablen `point2` ebenso auf `(99, 99)` geändert.

Auf das Objekt vom Typ `Point` an Position `(-1, 1)` zeigt nun keine Objektvariable mehr. Wenn die letzte Referenz auf ein Objekt gelöscht wird (also wenn keine Variable mehr auf das Objekt zeigt), wird das entsprechende Objekt automatisch als Müll (engl. *garbage*) markiert und somit zum Kandidaten für die sogenannte *garbage collection*. Von Zeit zu Zeit führt Java zur Laufzeit eines Programmes automatisch eine Methode aus, die alle mit *garbage* markierten Objekte sammelt und deren Speicher für das System wieder freigibt[6].

Bleibt noch die Frage, wie wir eine unabhängig Kopie eines Objektes erstellen können, *ohne* Aliase zu erzeugen. Eine Möglichkeit ist, dass alle relevanten Variablen des zu kopierenden Objektes einzeln kopiert werden.

Statt also das ganze Objekt zu kopieren (mit `point2 = point1`), könnten wir in diesem Beispiel zunächst die (x, y)-Koordinaten aus `point1` auslesen und diese in zwei Variablen `x` und `y` vom Typ `int` zwischenspeichern (hierzu bietet die Klasse `Point` die Methoden `getX()` und `getY()`). Mit diesen Angaben und der Methode `setLocation` kann nun das Objekt `point2` geändert werden:

```
double x = point1.getX();
double y = point1.getY();
point2.setLocation(x, y);
```

Beide Punkte `point1` und `point2` besitzen jetzt zwar auch die gleichen Koordinaten

[6] In anderen Sprachen ist die Programmiererin selber für die *garbage-collection* verantwortlich.

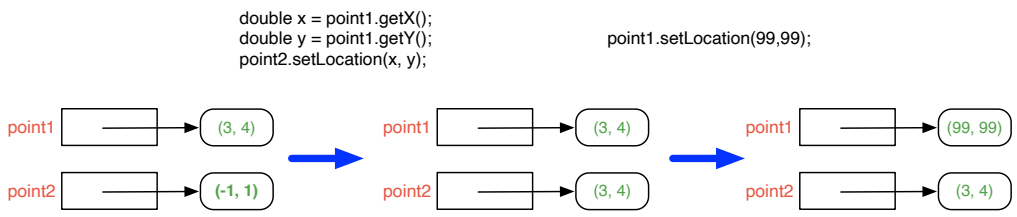

Abbildung 3.6: Unabhängige Kopien von Objekten zu erstellen ist möglich, aber unter Umständen mit etwas grösserem Aufwand verbunden.

(sind also quasi Kopien voneinander), referenzieren aber unabhängige Objekte und sind somit *keine* Aliase. Das heisst, wenn wir nun einen der beiden Punkte verändern, verändern wir wirklich nur *dieses* Objekt – das Vorgehen und die Effekte sind in Abb. 3.6 illustriert.

Kapitel 4

Eigene Klassen Programmieren (Teil 1)

4.1 Einführung

Im vorherigen Kapitel haben wir exemplarisch einige Klassen verwendet, die uns das Java API zur Verfügung stellt (z.B. die Klassen `String`, `Scanner`, oder `Random`). Wir haben aus den Klassen Objekte instanziiert und die instanziierten Objekte bzw. deren Funktionalitäten verwendet (um z.B. Zeichenketten zu manipulieren, um Eingaben entgegenzunehmen oder um Zufallszahlen zu generieren). Obschon das Java API viele nützliche Klassen zur Verfügung stellt, ist der Kern der objektorientierten Programmierung das Definieren und Programmieren *eigener* Klassen, welche die spezifischen Bedürfnisse an ein zu erstellendes Programm erfüllen.

Wie Sie bereits wissen, sind Klassen eine Art Bauplan, aus denen Objekte generiert werden können. Die Klasse repräsentiert das allgemeine Konzept der Objekte und umgekehrt ist jedes Objekt eine konkrete Realisierung des allgemeinen Konzeptes. Z.B. repräsentiert die Klasse `String` das Konzept einer Zeichenkette und jedes Objekt vom Typ `String` repräsentiert eine spezifische Instanz dieses Konzeptes (`"Hallo"` oder `"Welt"` oder `"42"` sind verschiedene Objekte der Klasse `String`).

Wir betrachten ein anderes Beispiel. Nehmen wir an, wir wollen in unserem Programm das Konzept eines Mitarbeitenden modellieren. Hierzu definieren wir in Java eine Klasse `Employee`, welche "nur" das Modell eines Mitarbeitenden repräsentiert. Jedes instanziierte Objekt vom Typ `Employee` repräsentiert dann einen tatsächlich existierenden Mitarbeitenden (z.B. die Programmiererin *Anna Muster* mit Personalnummer *1234* oder den Projektleiter *Hans Meier* mit Personalnummer *2435*).

In einem Programm zur Verwaltung von Mitarbeitenden programmieren wir also *eine* Klasse `Employee`, aus der dann zur Laufzeit des Programmes *mehrere* Objekte vom Typ `Employee` instanziiert werden können.

Jedes Objekt besitzt einen *Zustand*, der durch die Werte der zum Objekt gehörenden Eigenschaften definiert ist. Mögliche Eigenschaften eines Mitarbeitenden wären z.B. Name, Funktion, Gehalt, etc. In Java werden die Eigenschaften als *Variablen*, die innerhalb der Klassen deklariert sind, programmiert. Indem wir die gewünschten Variablen in der Klasse `Employee` programmieren, geben wir vor, dass jedes Objekt vom Typ `Employee` eigene Werte für diese Eigenschaften speichern kann.

Ein Objekt besitzt neben Eigenschaften oftmals auch ein Verhalten oder Funktionalitäten, welche das Objekt ausführen kann. Mögliche Funktionalitäten eines Objektes vom Typ `Employee` könnten sein:

- Ändern der Adresse, der Funktion oder des Gehalts
- Ausgeben des Namens
- Berechnen der Sozialabgaben (basierend auf dem Gehalt)

In Java wird die Funktionalität von Objekten mit *Methoden* innerhalb der entsprechenden Klasse programmiert. In der Klasse `Employee` programmieren wir also Methoden, welche dann auf (oder durch) ein gegebenes Objekt vom Typ `Employee` ausgeführt werden.

Beispiel 41 *Nehmen wir an, dass wir in der Klasse* `Employee` *die Variable* `salary` *und die Methode* `changeSalary` *programmiert haben, mit der man das Gehalt des Mitarbeitenden (also die Variable* `salary`*) auf einen neuen Wert setzen kann. Ist ein Objekt* `emp1` *aus* `Employee` *instanziiert, so können wir – wie gewohnt – mit dem Punkt-Operator die Methode aufrufen:*

```
emp1.changeSalary(150000);
```

4.2 Aufbau einer Java Klasse

In all unseren bisherigen Beispielprogrammen haben wir jeweils eine einzige Klasse geschrieben, die eine einzige Methode enthalten hat (nämlich die Methode `main`). Diese Klassen repräsentieren kleine, komplette Programme. Klassen, die die Methode `main` enthalten, sind aber nur der Anfangspunkt eines Programms. In diesen

Programmen haben wir Objekte aus Klassen des Java APIs instanziiert und anschliessend die Methoden genutzt, die uns diese Objekte anbieten.

Die bereits geschriebenen Klassen des Java APIs waren immer Teil unserer Programme – aber wir haben uns nie explizit um diese gekümmert, wir haben sie nur benutzt (in der Hoffnung, dass diese die Funktionalität, welche sie versprechen, auch tatsächlich korrekt ausführen).

Betrachten Sie die Klasse `RollTheDice`, die ähnlich aufgebaut ist wie unsere bisherigen Beispiele:

```
import java.util.Scanner;

public class RollTheDice {

        /**
         * Erlaubt einen Würfel zu werfen oder
         * diesen auf eine bestimmte Seite zu drehen
         */
        public static void main(String[] args) {
                Scanner scan = new Scanner(System.in);
                int decision;

                // instanziieren eines Würfel Objektes
                Dice dice = new Dice();

                System.out.println("Was wollen Sie mit dem Würfel machen? " +
                                "1=Werfen; 2=Drehen");
                decision = scan.nextInt();

                if (decision == 1) {
                        System.out.println("Würfel wird geworfen...");
                        dice.roll();
                } else {
                        System.out.print("Punktezahl für den Würfel eingeben: ");
                        int points = scan.nextInt();
                        dice.setPoints(points);
                }
                System.out.println("Der Würfel zeigt nun: " + dice.getPoints());
        }
}
```

Diese Klasse beinhaltet auch nur die Methode `main`. Zunächst wird ein Objekt vom Typ `Scanner` instanziiert, um Eingaben vom Benutzer entgegennehmen zu können. Danach wird ein Objekt `dice` vom Typ `Dice` mit dem `new`-Operator und dem zugehörigen Konstruktor `Dice` instanziiert – das Objekt `dice` repräsentiere einen herkömmlichen Spielwürfel mit sechs Seiten.

Der Benutzer wird über seine Möglichkeiten informiert und seine Entscheidung wird entgegengenommen (und der Variablen `decision` zugewiesen). Entscheidet sich der Benutzer für die Option 1, so wird der Würfel `dice` geworfen – hierzu existiert offenbar eine Methode `roll`, die auf Objekten vom Typ `Dice` aufgerufen werden kann. Entscheidet sich der Benutzer hingegen für Option 2, so kann dieser eine beliebige Punktezahl eingeben, auf die der Würfel gedreht werden soll – hierzu existiert eine Methode `setPoints`, die eine ganze Zahl als Parameter entgegennimmt. In beiden Fällen wird zum Schluss des Programmes ausgegeben, was der Würfel `dice` nun aktuell anzeigt – dies geschieht mit der Methode `getPoints`.

Eine mögliche Ein- und Ausgabe wäre demnach:

```
Was wollen Sie mit dem Würfel machen? 1=Werfen; 2=Drehen
1
Würfel wird geworfen...
Der Würfel zeigt nun: 5
```

Alternativ ist folgende Ein- und Ausgabe möglich:

```
Was wollen Sie mit dem Würfel machen? 1=Werfen; 2=Drehen
2
Gewünschte Punktezahl für den Würfel eingeben: 4
Der Würfel zeigt nun: 4
```

Der Unterschied zwischen diesem Beispiel und den Beispielen aus den vorangegangenen Kapiteln ist, dass die Klasse `Dice` *nicht* Teil des Java APIs ist (so wie zum Beispiel die Klassen `String`, `Random`, `Scanner`, etc.). Wir müssen die Klasse `Dice` selber programmieren (ansonsten kann die Klasse `RollTheDice` nicht kompiliert werden).

Jede Klasse kann Deklarationen von Variablen und Methoden enthalten (siehe Abb. 4.1). Die Klassen, die wir bisher selber geschrieben haben, folgen auch diesem Muster. Der Unterschied ist, dass wir bisher keine Variablen ausserhalb von Methoden deklariert haben und dass wir immer nur eine einzige Methode (nämlich die Methode `main`) programmiert haben (solche Klassen definieren den Startpunkt des Programmes und man nennt diese auch *Treiberklassen*).

Kernstück der objektorientierten Programmierung ist es, Klassen zu programmieren, die Objekte und deren Eigenschaften und Funktionalitäten modellieren – und

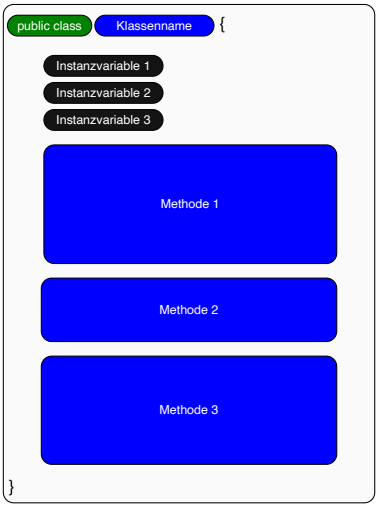

Abbildung 4.1: Klassen enthalten Deklarationen von Variablen und Methoden.

zwar mit Variablen und Methoden. Ein Objekt vom Typ `Dice` zum Beispiel soll die maximal zulässige Punktezahl sowie die aktuell gezeigte Punktezahl des Würfels in Variablen speichern. Objekte vom Typ `Dice` sollen zudem Methoden zur Verfügung stellen, die den Würfel werfen, den Würfel manuell auf eine beliebige Seite drehen und die aktuelle Punktezahl des Würfels auslesen.

Die folgende Klasse `Dice` erfüllt diese Anforderungen (→Video K-4.1): Die Klasse enthält drei Deklarationen von Variablen mit den Bezeichnern `MAX`, `points` und `randomGenerator` (`MAX` ist als Konstante deklariert). Daneben beinhaltet die Klasse fünf Methoden mit den Bezeichnern `Dice`, `roll`, `setPoints`, `getPoints` und `printMessage`. Oberhalb jeder Variablen und jeder Methode wird in einem Kommentar jeweils der Zweck der Variable oder der Methode knapp erläutert. Diese Kommentare helfen Programmiererinnen, den Quellcode schneller zu verstehen. Zudem kann man den Quellcode mit Kommentaren formatieren und strukturieren und so eine bessere Lesbarkeit erreichen.

Anhand der Kommentare wissen wir, dass die Konstante `MAX` die maximal mögliche Punktezahl des Würfels definiert, die Variable `points` die aktuelle Punktezahl des Würfels speichert und die Variable `randomGenerator` verwendet werden kann, um Zufallszahlen zu generieren. Zudem sehen wir anhand der Kommentare, dass die Methode `Dice` ein Objekt vom Typ `Dice` instanziieren kann und die Methode `roll` die Punktezahl des Würfels mit einer Zufallszahl neu definiert. Mit der Methode `setPoints` kann man die Punktezahl des Würfels manuell definieren und die aktuelle

Punktezahl des Würfels kann man mit der Methode `getPoints` auslesen. Schliesslich ist die Methode `printMessage` dazu da, eine Systemmeldung zu generieren.

```
import java.util.Random;

public class Dice {

        // maximale Punktezahl des Würfels
        public final int MAX = 6;
        // aktuelle Punktezahl des Würfels
        private int points;
        // Objekt zur Erzeugung von Zufallszahlen
        private Random randomGenerator;

        /**
         * instanziiert einen neuen Würfel und initialisiert die Variablen
         */
        public Dice(){
                this.randomGenerator = new Random();
                this.points = 1;
        }

        /**
         * "wirft" den Würfel (setzt die Punktezahl auf eine zufällige, gültige Zahl)
         */
        public int roll(){
                this.points = this.randomGenerator.nextInt(MAX) + 1;
                return this.points;
        }

        /**
         * setzt die aktuelle Punktezahl des Würfels auf den übergebenen Wert
         */
        public void setPoints(int points){
                if (points >= 1 && points <= MAX)
                        this.points = points;
                else
                        this.printMessage("Unzulässige Eingabe. "
                                        + "Würfel wird nicht verändert.");
        }

        /**
         * gibt die aktuelle Punktezahl des Würfels zurück
         */
        public int getPoints(){
                return this.points;
        }

        /**
         * gibt eine Systemmeldung aus
         */
        private void printMessage(String message){
                System.out.println(message);
        }
}
```

In den Beispielen, die wir in diesem Buch erarbeiten, werden wir jede Klasse in einer eigenen Datei speichern[1]. Das bedeutet in diesem Beispiel, dass wir die Klasse `RollTheDice` in der Datei `RollTheDice.java` und die Klasse `Dice` in einer separaten Datei `Dice.java` speichern.

In den folgenden Abschnitten wollen wir nun wichtige Konzepte der objektorientierten Programmierung anhand der Klasse `Dice` und deren Variablen und Methoden einführen und besprechen.

4.3 Instanzvariablen und Sichtbarkeitsmodifikatoren

Bis jetzt haben wir Variablen immer innerhalb von Methoden deklariert. Wird eine Variable innerhalb einer Methode deklariert, so kann auch nur innerhalb *dieser* Methode auf sie zugegriffen werden. Solche Variablen nennt man *lokale Variablen*.

In der Klasse `Dice` sind sowohl die Konstante `MAX` als auch die Variablen `points` und `randomGenerator` innerhalb der Klasse, aber nicht innerhalb einer Methode deklariert. Solche Variablen nennt man *Instanzvariablen*. Instanzvariablen können von *jeder* Methode der entsprechenden Klasse verwendet werden. Wo also eine Variable deklariert wird, definiert den Bereich, in dem diese tatsächlich benutzt werden kann (man sagt diesem Bereich auch *der Scope einer Variablen*).

Variablen wie z.B. `points` nennt man Instanzvariablen, weil für jede Instanz der Klasse eigener Speicherplatz für die Variable reserviert wird. Mit anderen Worten, jedes instanziierte Objekt vom Typ `Dice` besitzt eine eigene, unabhängige Variable `points`. So kann jedes Objekt seinen eigenen Zustand besitzen.

Beispiel 42 *Die Unabhängigkeit von Instanzvariablen von Objekten des gleichen Typs wird in folgendem Programm illustriert. In der Methode* `main` *werden drei Objekte vom Typ* `Dice` *instanziiert – die ersten beiden Würfel werden mit der Methode* `roll` *geworfen und der dritte Würfel wird auf die Seite mit der Punktezahl* `6` *gedreht.*

```
public class DiceTester {

        /**
         * Illustriert die Unabhängigkeit von Objekten
         */
```

[1] Java erlaubt es grundsätzlich mehrere Klassen in einer Datei zu speichern. Wenn eine Datei mehrere Klassen enthält, darf nur eine dieser Klassen mit dem reservierten Wort `public` deklariert werden. Ferner muss der Bezeichner der `public` Klasse dem Namen der Datei entsprechen.

```
        public static void main(String[] args) {
                Dice firstDice = new Dice();
                Dice secondDice = new Dice();
                Dice thirdDice = new Dice();

                firstDice.roll();
                secondDice.roll();
                thirdDice.setPoints(6);

                System.out.println("Würfel 1: " + firstDice.getPoints());
                System.out.println("Würfel 2: " + secondDice.getPoints());
                System.out.println("Würfel 3: " + thirdDice.getPoints());
        }
}
```

Eine mögliche Ausgabe des Programms lautet:

```
Würfel 1: 2
Würfel 2: 5
Würfel 3: 6
```

Ein Würfel besitzt also aktuell z.B. die Punktezahl `2`*, während ein anderer Würfel aktuell* `5` *oder* `6` *in der Variablen* `points` *speichert.*

Instanzvariablen sollten nur durch das eigene Objekt gelesen oder verändert werden können. Das heisst zum Beispiel, dass nur die Methoden der Klasse `Dice` den aktuellen Punktewert – also die Variable `points` – lesen oder ändern können sollten. Wir sollten es für Quellcode ausserhalb einer Klasse schwierig (oder sogar unmöglich) machen, in ein Objekt "hineinzugreifen" und dort Werte von Variablen direkt zu ändern oder auszulesen. Diese Strategie nennt man *Kapselung*.

Es gibt also zwei Sichten auf ein Objekt:

- *interne Sicht*: Die Details der Variablen und Methoden der Klasse, die es definiert
- *externe Sicht*: Die Dienste, die ein Objekt bereitstellt und wie das Objekt mit dem Rest des Systems interagiert

Aus der externen Sicht betrachtet, ist ein Objekt eine gekapselte Einheit, die eine Reihe von Diensten bereitstellt.

Ein *Modifikator* (engl. *Modifier*) ist ein reserviertes Wort, das bestimmte Eigenschaften einer Methode oder einer Variablen festlegt. Wir haben z.B. den Modifikator

`final` zur Definition von Konstanten kennengelernt. Sogenannte *Sichtbarkeitsmodifikatoren* (engl. *Visibility Modifier*) kontrollieren die Kapselung der Instanzvariablen und die Zugriffsmöglichkeiten auf die Methoden einer Klasse (zur Definition der externen Sicht auf ein Objekt). In Java werden hierzu die Sichtbarkeiten `public` und `private` verwendet[2].

Hat eine Variable oder eine Methode die Sichtbarkeit `public`, so kann diese auch von *ausserhalb* der Klasse referenziert werden (die Variable oder die Methode ist also über Klassengrenzen hinweg sichtbar). Besitzt eine Variable oder eine Methode hingegen die Sichtbarkeit `private`, so kann diese nur *innerhalb* der Klasse referenziert werden – ausserhalb der Klasse bleiben diese Variablen oder Methoden unsichtbar (können also nicht verwendet werden).

Eine mit `public` modifizierte Variable verletzt das Prinzip der Kapselung – Quellcode ausserhalb der Klasse besitzt direkten Zugriff auf diese Variable und der Wert der Variablen kann somit unkontrolliert ausgelesen, gelöscht oder verändert werden. Instanzvariablen (also Variablen die innerhalb einer Klasse, aber nicht innerhalb einer Methode deklariert sind) sollten Sie deshalb i.d.R mit dem Sichtbarkeitsmodifikator `private` deklarieren.

Kapselung verlangt, dass Daten von aussen nicht direkt veränderbar oder einsehbar sein sollen – Konstanten können aber per Definition nicht verändert werden (wegen dem Modifikator `final` in der Deklaration). Zudem müssen Konstanten oftmals von verschiedenen Klassen verwendet werden. Es ist deshalb im Allgemeinen akzeptabel, wenn man Konstanten mit dem Sichtbarkeitsmodifikator `public` deklariert.

Beispiel 43 *In der Klasse* `Dice` *haben wir die Konstante* `MAX` *mit Sichtbarkeit* `public` *deklariert – die Instanzvariablen* `points` *und* `randomGenerator` *hingegen sind durch den Sichtbarkeitsmodifikator* `private` *gegen aussen gekapselt:*

```
// maximale Punktezahl des Würfels
public final int MAX = 6;
// aktuelle Punktezahl des Würfels
private int points;
// Objekt zur Erzeugung von Zufallszahlen
private Random randomGenerator;
```

Da sich die drei Deklarationen nicht innerhalb einer Methode befinden, sind dies

[2] Es existiert noch ein dritter Sichtbarkeitsmodifikator, `protected`, der relevant ist für das Konzept der Vererbung und den wir in späteren Kapiteln einführen werden.

	Variablen	Konstanten	Methoden
public	Verletzt das Prinzip der Kapselung	Konstante ist gegen Aussen sichtbar	Bieten Funktionalität gegen Aussen
private	Kapselung der Variablen	Kapselung der Konstanten	Unterstützen die Methoden dieser Klasse
protected	Teilweise Kapselung bleibt erhalten (Details folgen später)		

Abbildung 4.2: Die Effekte der Sichtbarkeitsmodifikatoren `public`, `private` und `protected` auf Variablen (bzw. Konstanten) und Methoden zusammengefasst.

nicht lokale Variablen, sondern Instanzvariablen und somit haben alle Methoden der Klasse `Dice` Zugriff auf diese.

Der Sichtbarkeitsmodifikator, den wir auf Methoden anwenden, hängt vom Zweck der Methode ab. Methoden, die Funktionalitäten für andere Objekte anbieten, müssen mit dem Modifikator `public` deklariert werden, so dass diese von Aussen aufgerufen werden können. Diese Methoden nennt man auch *Service Methoden*.

Eine mit Sichtbarkeit `private` deklarierte Methode kann hingegen nicht von ausserhalb der Klasse aufgerufen werden. Der einzige Zweck einer so deklarierten Methode ist die Unterstützung anderer Methoden in der gleichen Klasse. Man nennt diese Methoden auch *Support Methoden*.

Beispiel 44 *In der Klasse `Dice` sind die Methoden `Dice`, `roll`, `setPoints` und `getPoints` mit Sichtbarkeit `public` deklariert – dies sind die Service-Methoden, die auch ausserhalb der Klasse `Dice` aufrufbar sind. Tatsächlich haben wir diese ja auch von ausserhalb, nämlich aus der Klasse `RollTheDice`, aufgerufen – z.B. mit `dice.roll()`.*

Die Methode `printMessage` hingegen, unterstützt die Methode `setPoints` und soll gegen aussen nicht sichtbar sein – dies ist also eine Support Methode. Es ist somit zum Beispiel nicht möglich, die Anweisung `dice.printMessage("Nachricht");` in der Klasse `RollTheDice` auszuführen.

In Abb. 4.2 sind die Effekte der Sichtbarkeiten `public`, `private` und `protected` auf Variablen (bzw. Konstanten) und Methoden zusammengefasst.

4.4 Methoden Aufrufen und dem Kontrollfluss Folgen

Eine Methode ist eine Gruppierung von Programmieranweisungen, die unter einem Bezeichner zusammengefasst werden. Eine Methode definiert innerhalb des Methodenrumpfes alle Programmieranweisungen, die ausgeführt werden sollen, wenn die Methode über den entsprechenden Methodennamen aufgerufen wird. In Java sind Methoden immer Teil einer bestimmten Klasse – Sie können also Methoden nicht ausserhalb von Klassen programmieren.

Wird eine Methode aufgerufen, so wird der sogenannte *Kontrollfluss* des Programms an diese Methode abgegeben. Jetzt werden die Programmieranweisungen dieser Methode ausgeführt. Sind alle Anweisungen, die in der Methode zusammenfasst sind, ausgeführt, kehrt die Kontrolle zum Punkt des Programmes zurück, wo die Methode aufgerufen wurde.

Die Methodenaufrufe befinden sich in den meisten Fällen innerhalb anderer Methoden. Methoden können also zwei Rollen einnehmen: *aufrufend* oder *aufgerufen*. Rufen wir z.B. in der Methode `main` die Methode `setPoints` auf, so ist `main` in diesem Fall die *aufrufende Methode* und `setPoints` die *aufgerufene Methode*.

Wenn sich die aufrufende und die aufgerufene Methode in der gleichen Klasse befinden, so ist für den Aufruf nur der Bezeichner der Methode nötig. Wenn die aufgerufene Methode aber Teil einer anderen Klasse ist als die aufrufende Methode, so ist neben dem Bezeichner der Methode eine Objektvariable und der Punkt-Operator zum Aufruf nötig. Tatsächlich haben wir solche Methodenaufrufe über Klassengrenzen hinweg schon mehrfach angewendet: Zum Beispiel mit den Methodenaufrufen `scan.next()` oder `string.toUpperCase()`, etc.

Befinden sich die aufrufende und die aufgerufene Methode in der gleichen Klasse, so wird häufig mit der `this`-Referenz gearbeitet. Das heisst, statt dem Methodennamen – z.B. `printMessage()` – wird `this.printMessage()` geschrieben, um deutlich zu machen, dass sich die aufgerufene Methode auch in *diesem* Objekt befindet. Wir kommen später in diesem Kapitel nochmals auf das reservierte Wort `this` zurück.

Beispiel 45 *In Abb. 4.3 ist ein möglicher Verlauf des Kontrollflusses für das Programm* `RollTheDice` *abstrahiert. Bei jedem Aufruf einer Methode springt der Kontrollfluss aus der aufrufenden Methode zur aufgerufenen Methode.*

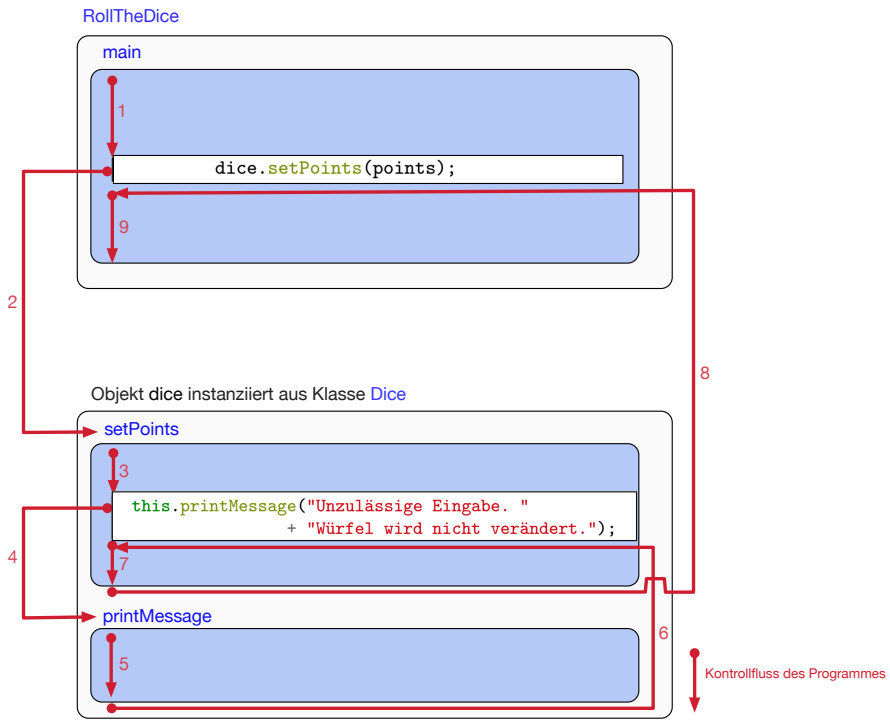

Abbildung 4.3: Der Kontrollfluss eines Programmes folgt den Aufrufen der Methoden.

Aus der Methode `main` *in der Klasse* `RollTheDice` *wird die Methode* `setPoints` *auf dem instanziierten Objekt* `dice` *aufgerufen. Der Kontrollfluss springt jetzt also aus* `main` *raus in die Methode* `setPoints`*. Die Methode* `setPoints` *wiederum ruft die Methode* `printMessage` *auf, die sich in der gleichen Klasse befindet – hierzu reicht der Bezeichner der Methode aus (ergänzt mit* `this`*). Immer wenn alle Anweisungen einer aufgerufenen Methode ausgeführt sind, kehrt der Kontrollfluss zur aufrufenden Methode zurück.*

4.5 Der Methodenkopf

Eine Methode ist definiert durch einen *Methodenkopf* und einen *Methodenrumpf* (auch *Methodenblock* genannt). Der Methodenkopf umfasst:

1. Einen Sichtbarkeitsmodifikator (z.B. `public` oder `private`)
2. Den Datentyp der Rückgabe (z.B. `int`, `double`, `String`, `Dice`, etc.) oder das reservierte Wort `void`, das angibt, dass diese Methode *keine* Rückgabe erzeugt.
3. Einen Bezeichner der Methode (z.B. `roll` oder `printMessage`). Zur Erinnerung: Methodennamen beginnen nach gängiger Konvention mit einem Klein-

buchstaben und jedes interne Wort beginnt wieder mit einem Grossbuchstaben.

4. Nach dem Bezeichner der Methode folgen optional eine Reihe von Parametern in Klammern `()`. Jeder Parameter wird beschrieben durch einen Datentyp und einen Bezeichner. Mehrere Parameter werden mit einem Komma separiert.

Nach dem Methodenkopf folgt der Methodenrumpf, in dem mehrere Programmieranweisungen innerhalb geschweifter Klammern gruppiert werden können.

Beispiel 46 *Drei Beispiele von Methodenköpfen:*

```
public void doThis() {
        // Methodenrumpf
}

private void make(String name) {
        // Methodenrumpf
}

private int help(int num1, int num2) {
        // Methodenrumpf
}
```

1. *Die erste Methode ist auch ausserhalb der Klasse sichtbar (`public`), während die anderen beiden Methoden nur innerhalb dieser Klasse verwendet werden können (`private`).*

2. *Die ersten beiden Methoden geben nichts zurück (der Rückgabetyp ist `void`), während die letzte Methode eine ganze Zahl zurückgeben wird (der Rückgabetyp ist `int`).*

3. *Die Bezeichner der drei Methoden sind `doThis`, `make` und `help`.*

4. *Die erste Methode nimmt keine Parameter entgegen, die zweite Methode nimmt einen Parameter vom Typ `String` und die dritte Methode zwei Parameter vom Typ `int` entgegen.*

4.6 Konstruktoren

Wenn wir eine eigene Klasse programmieren, dann definieren wir üblicherweise auch einen *Konstruktor*. Konstruktoren sind spezielle Methoden, die den gleichen Bezeichner wie die Klasse haben (der Konstruktor der Klasse `Scanner` hat bspw. den Bezeichner `Scanner` und der Konstruktor von `Dice` muss zwingend `Dice` heissen).

Konstruktoren unterscheiden sich in zwei Punkten von "normalen" Methoden:

1. Ein Konstruktor instanziiert das Objekt aus der Klasse und gibt danach die Adresse (die Referenz) auf das erzeugte Objekt zurück. Neben der Rückgabe dieser Adresse, kann der Konstruktor keine Werte zurückzugeben und besitzt demnach keinen Rückgabetyp (nicht zu verwechseln mit `void`). Der Methodenkopf des Konstruktors besteht also nur aus dem Sichtbarkeitsmodifikator (üblicherweise `public`), dem Bezeichner der Methode (z.B. `Dice`) und ggf. einer Reihe von Parametern in Klammern.

2. Der Konstruktor wird immer zusammen mit dem `new`-Operator aufgerufen, um ein neues Objekt aus der entsprechenden Klasse zu instanziieren.

Die vom Konstruktor zurückgegebene Adresse auf das eben instanziierte Objekt kann man z.B. einer deklarierten Objektvariablen zuweisen:

```
Dice dice = new Dice();
```

Der Konstruktor wird üblicherweise so programmiert, dass dieser die Instanzvariablen initialisiert. Zum Beispiel wird der Konstruktor der Klasse `Dice` dazu verwendet, die Instanzvariable `points` mit `1` zu initialisieren und der Instanzvariablen `randomGenerator` ein neu instanziiertes Objekt vom Typ `Random` zuzuweisen:

```
        // aktuelle Punktezahl des Würfels
        private int points;
        // Objekt zur Erzeugung von Zufallszahlen
        private Random randomGenerator;

        /**
         * instanziiert einen neuen Würfel und initialisiert die Variablen
         */
        public Dice() {
                this.randomGenerator = new Random();
                this.points = 1;
        }
```

Beachten Sie die Verwendung des reservierten Wortes `this` bei beiden Zuweisungen. Das reservierte Wort `this` erlaubt einem Objekt sich selber zu referenzieren. Eine `this` Referenz bezieht sich also immer auf *dieses* Objekt (in *diesem* Objekt soll die Instanzvariable `points` mit `1` initialisiert werden und in *diesem* Objekt soll die Instanzvariable `randomGenerator` instanziiert werden).

Wir müssen nicht zwingend für jede Klasse, die wir selber programmieren, einen Konstruktor definieren. Jede Klasse besitzt nämlich automatisch einen *Standardkonstruktor* (ohne Parameter). Der Standardkonstruktor instanziiert Objekte aus

der entsprechenden Klasse ohne Instanzvariablen zu initialisieren. Sobald Sie einen eigenen Konstruktor in einer Klasse programmiert haben, kann man den Standardkonstruktor für diese Klasse aber nicht mehr aufrufen.

4.7 Parameter für Methoden

Ein *Parameter* ist ein Wert, der einer Methode mitgegeben wird, wenn diese aufgerufen wird. Die Parameterliste im Methodenkopf definiert dabei die Typen der übergebenen Werte und deren Bezeichner. Diese Bezeichner können wir dann innerhalb der Methode verwenden, wenn wir die Werte der entsprechenden Parameter benötigen. Die Bezeichner der Parameter im Methodenkopf nennt man *formale Parameter*. Die Werte, die beim Aufruf an die Methode übergeben werden, nennt man *tatsächliche Parameter* oder auch *Argumente*.

Formale Parameter sind Bezeichner, die als Variablen innerhalb einer Methode dienen und dessen initiale Werte den Argumenten beim Aufruf der Methode entsprechen. Das heisst, wenn eine Methode aufgerufen wird, wird jedes Argument dem entsprechenden formalen Parameter zugewiesen.

Beispiel 47 *Betrachten Sie z.B. die Methode* `setPoints` *in der Klasse* `Dice`*:*

```
/**
 * setzt die aktuelle Punktezahl des Würfels auf den übergebenen Wert
 */
public void setPoints(int points){
        if (points >= 1 && points <= MAX)
                this.points = points;
        else
                this.printMessage("Unzulässige Eingabe. "
                                + "Würfel wird nicht verändert.");
}
```

Der formale Parameter ist `points` *vom Datentyp* `int`*. Wird die Methode* `setPoints` *aufgerufen, so muss ein Wert oder eine Variable vom Typ* `int` *übergeben werden: Z.B.* `dice.setPoints(4);` *oder* `dice.setPoints(p);`*, wobei* `p` *eine lokale Variable vom Typ* `int` *ist.*

Der übergebene Wert 4 oder die übergebene Variable `p` *entsprechen den tatsächlichen Parametern (den Argumenten). Diese Werte werden beim Aufruf der Methode dem formalen Parameter* `points` *zugewiesen. Bei einem Methodenaufruf*

```
dice.setPoints(4);
```

geschieht also eigentlich folgende Deklaration und Zuweisung auf der ersten Zeile der Methode:

```
int points = 4;
```

Beim Aufruf einer Methode müssen die tatsächlichen Parameter (die Argumente) und die formalen Parameter im Methodenkopf übereinstimmen. Dies muss so sein, weil das erste Argument dem ersten formalen Parameter, das zweite Argument dem zweiten formalen Parameter, etc. zugewiesen wird.

Beispiel 48 *Beim Aufruf der Methode* `printInfo` *werden die tatsächlichen Parameter* `"Kaspar"` *und* `41` *an die Methode übergeben.*

```
int age = 41;
String name = "Kaspar";
helper.printInfo(name, age);
```

Die tatsächlichen Parameter werden nun den formalen Parametern `name` *und* `a` *zugewiesen.*

```
public void printInfo(String name, int a){
        System.out.println("Name: " + name + ", Alter: " + a);
}
```

In der Methode `printInfo` *geschieht also als Erstes Folgendes:*

```
int a = 41;
String name = "Kaspar";
```

Die Ausgabe der Methode `printInfo` *würde demnach lauten:*
`Name: Kaspar, Alter: 41`

Beachten Sie, dass die tatsächlichen und die formalen Parameter gleiche Bezeichner haben dürfen (z.B. `name` *und* `name`*) – dies ist aber nicht zwingend (z.B.* `age` *und* `a`*).*

Die formalen Parameter in einem Methodenkopf entsprechen lokalen Variablen. Diese Variablen existieren also nur innerhalb der entsprechenden Methode. Es ist möglich, lokale Variablen (und somit auch formale Parameter) mit dem gleichen Bezeichner zu deklarieren, wie er bereits für eine Instanzvariable verwendet wird.

Zum Beispiel haben wir in der Klasse `Dice` eine Instanzvariable `points` deklariert:

```
        // aktuelle Punktezahl des Würfels
        private int points;
```

In der Methode `setPoints` gibt es eine lokale Variable `points` (in der Form eines formalen Parameters).

```
        /**
         * setzt die aktuelle Punktezahl des Würfels auf den übergebenen Wert
         */
        public void setPoints(int points){
                if (points >= 1 && points <= MAX)
                        this.points = points;
                else
                        this.printMessage("Unzulässige Eingabe. "
                                        + "Würfel wird nicht verändert.");
        }
```

Wird innerhalb der Methode `setPoints` der Bezeichner `points` verwendet, wird damit *immer* die lokale Variable angesprochen und nicht die Instanzvariable. Mit dem reservierten Wort `this` kann man aber erzwingen, dass die Instanzvariable referenziert wird. Das bedeutet also, dass folgende Zuweisung

```
this.points = points;
```

der Instanzvariablen `points` *dieses* Objektes den Wert der lokalen Variablen `points` zuweist.

4.8 Die `return` Anweisung

Der Datentyp der Rückgabe, der im Methodenkopf angegeben wird, kann ein primitiver Datentyp sein (z.B. `int`, `char`, etc.), eine Klasse (z.B. `String`, `Dice`, etc.) oder aber das reservierte Wort `void`[3].

Wird `void` als Rückgabetyp definiert, dann erzeugt die Methode keine Rückgabe. Zum Beispiel erzeugen die Methoden `main` oder `setPoints` in unserem Beispiel keine Rückgaben – der Rückgabetyp dieser Methoden ist deshalb `void`.

[3]Nicht vergessen: Konstruktoren besitzen als einzige Methoden keinen Rückgabetyp (auch nicht `void`).

Eine Methode, die einen anderen Rückgabetyp als `void` besitzt, muss mindestens eine `return`-Anweisung im Methodenrumpf beinhalten. Eine `return`-Anweisung besteht aus dem reservierten Wort `return` gefolgt von einem Wert oder einem Ausdruck, der zurückgegeben werden soll. Beachten Sie, dass der Datentyp der Rückgabe dem im Methodenkopf deklarierten Rückgabetyp entsprechen muss (wurde zum Beispiel `int` als Rückgabetyp definiert, so kann die Methode nicht den Wert `3.16` zurückgeben).

Der Datentyp der Rückgabe von `getPoints` in der Klasse `Dice` ist zum Beispiel `int`:

```
/**
 * gibt die aktuelle Punktezahl des Würfels zurück
 */
public int getPoints(){
        return this.points;
}
```

Die Methode `getPoints` gibt in unserem Beispiel `this.points` zurück. Dies entspricht der aktuell gespeicherten Punktezahl in *diesem* Würfel. Der Rückgabetyp der Methode `roll` ist auch `int` und macht fast dasselbe wie `getPoints` – bevor aber der aktuelle Wert des Würfels zurückgegeben wird, wird der Wert der Instanzvariablen `points` noch auf eine neue Zufallszahl zwischen `1` und `6` gesetzt:

```
/**
 * "wirft" den Würfel (setzt die Punktezahl auf eine zufällige, gültige Zahl)
 */
public int roll(){
        this.points = this.randomGenerator.nextInt(MAX) + 1;
        return this.points;
}
```

Wird eine `return`-Anweisung in einer Methode ausgeführt, so wird der definierte Ausdruck an die aufrufende Methode zurückgegeben. Die Rückgabe einer Methode kann dann von der aufrufenden Methode entgegengenommen und verwendet werden. Z.B. kann man die Rückgabe ausgeben:

```
System.out.println("Der Würfel zeigt: " + dice.getPoints());
```

Oder man weist die Rückgabe einer Variablen zu:

```
int currentPoints = dice.getPoints();
```

Die Rückgabe einer Methode kann aber auch ignoriert werden. Das heisst, die Rückgabe muss nicht zwingend von der aufrufenden Methode verarbeitet werden. Zum Beispiel haben wir in der Klasse `RollTheDice` die Methode `roll` aufgerufen (mit `dice.roll()`) und dabei die Rückgabe von `roll` nicht verwendet.

Neben der Rückgabe eines Wertes an die aufrufende Methode hat die `return`-Anweisung noch einen anderen Effekt: Der Kontrollfluss wird *sofort* an die aufrufende Methode zurückgegeben.

Eine Methode ohne Rückgabe (der Rückgabetyp der Methode ist also `void`) kehrt zur aufrufenden Methode zurück, sobald das Ende der Methode erreicht wird. Allerdings können Sie in Methoden mit Rückgabe `void` eine `return`-Anweisung *ohne* Rückgabe verwenden:

```
return;
```

Wird dieses "leere" `return` der Methode erreicht, kehrt der Kontrollfluss sofort zur aufrufenden Methode zurück. Die Anweisungen, die unterhalb einer `return`-Anweisung programmiert sind, werden also nicht mehr ausgeführt.

4.9 *Getter* und *Setter*

Wir haben gesehen, dass die Instanzvariablen in Objekten gekapselt sein sollen. Hierzu werden diese mit dem Sichtbarkeitsmodifikator `private` deklariert und sind somit ausserhalb der Objekte nicht direkt verwendbar. Die Interaktion zwischen diesen gekapselten Daten und anderen Objekten soll/kann nur über die von Ihnen programmierte Menge von Service-Methoden erfolgen – also über die Methoden mit Sichtbarkeit `public`. Diese Menge an sichtbaren Methoden definieren die *Schnittstelle* (engl. *Interface*) zwischen dem Objekt und dem Rest des Quellcodes.

Die Prinzipien der Kapselung und der Schnittstellen sind in Abb. 4.4 illustriert. Objekte sind nicht berechtigt, direkt auf die Variablen eines anderen Objektes zuzugreifen (weil diese mit der Sichtbarkeit `private` deklariert sind). Stattdessen müssen diese die `public` Methoden des Objektes aufrufen, die wiederum mit den im Objekt gekapselten Variablen interagieren.

Methoden wie z.B. `getPoints` nennt man *Getter*, da diese das Auslesen einer als

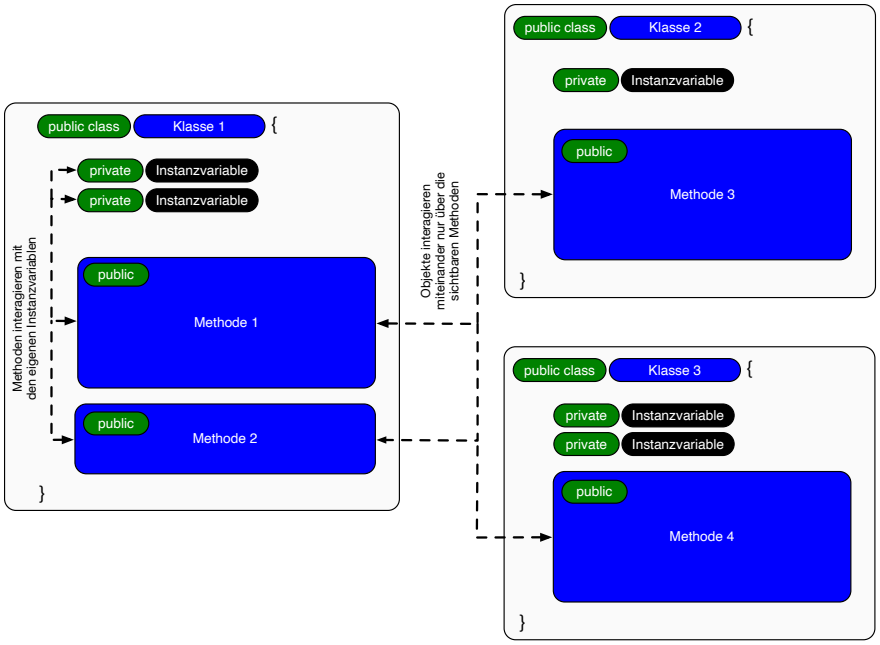

Abbildung 4.4: Objekte interagieren mit anderen Objekten über die `public` Methoden.

`private` deklarierten Variablen ermöglichen. Die Methode `setPoints` ermöglicht das Ändern der als `private` deklarierten Variablen `points`. Diese Methode überschreibt den aktuellen Wert der Instanzvariablen `points` mit dem Wert, der dem formalen Parameter `points` zugewiesen wird. Solche Methoden nennt man auch *Setter*. Es ist gängige Konvention, dass die *Getter* und *Setter* für eine Instanzvariable mit Bezeichner `value` die Bezeichner `getValue` bzw. `setValue` erhalten[4].

Der Nutzen des Umwegs beim Zugriff auf die Instanzvariablen über *Getter* und *Setter* ist *Kontrolle*. Beachten Sie zum Beispiel, dass mit einer `if`-Anweisung in der Methode `setPoints` sichergestellt wird, dass die Variable `points` nur dann geändert wird, wenn `points` im gültigen Bereich liegt (zwischen `1` und `6`).

```
/**
 * setzt die aktuelle Punktezahl des Würfels auf den übergebenen Wert
 */
public void setPoints(int points){
        if (points >= 1 && points <= MAX)
                this.points = points;
        else
                this.printMessage("Unzulässige Eingabe. "
                                + "Würfel wird nicht verändert.");
}
```

[4]Einzige Ausnahme zu dieser Regel bilden die *Getter* für Instanzvariablen vom Typ `boolean` – z.B. benennt man den *Getter* für eine Variable `full` vom Typ `boolean` typischerweise mit `isFull`.

Würden wir den direkten Zugriff auf Instanzvariablen erlauben, könnte man Objekte unzulässig manipulieren oder unzulässigerweise Variablenwerte auslesen. Z.B. könnten wir so in der Methode `main` in der Klasse `RollTheDice` folgende Anweisung programmieren:

```
dice.points = 17;
```

Mit dem Umweg über *Getter* und *Setter* haben wir als Programmiererinnen immer die volle Kontrolle darüber, was mit den Instanzvariablen gemacht werden darf und was nicht.

Beispiel 49 *Betrachten Sie folgende Ein- und Ausgabe für unser Beispielprogramm:*

```
Was wollen Sie mit dem Würfel machen? 1=Werfen; 2=Drehen
2
Gewünschte Punktezahl für den Würfel eingeben: 17
Unzulässige Eingabe. Würfel wird nicht verändert.
Der Würfel zeigt nun: 1
```

Die unzulässige Eingabe des Benutzers wird also durch die sorgfältige Programmierung des Setters abgefangen.

Betrachten Sie die Methode `roll` in der Klasse `Dice`:

```
/**
 * "wirft" den Würfel (setzt die Punktezahl auf eine zufällige, gültige Zahl)
 */
public int roll(){
        this.points = this.randomGenerator.nextInt(MAX) + 1;
        return this.points;
}
```

Diese Methode ändert den Wert der Instanzvariablen `points` zufällig und liefert dann den aktualisierten Wert der Variablen zurück (arbeitet also ähnlich wie ein *Getter*). Beachten Sie wiederum, dass der Quellcode der Methode `roll` dafür besorgt ist, dass der Wert der Variablen `points` zwar zufällig gewählt wird, aber in jedem Fall zwischen `1` und `6` liegt. Das heisst, die Methode `roll` ist so konstruiert, dass nur zulässige Änderungen an der Variablen `points` vorgenommen werden können.

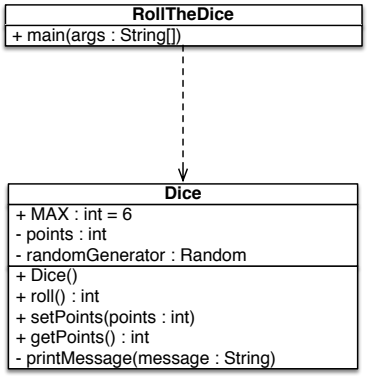

Abbildung 4.5: Ein UML Klassendiagramm für das Programm `RollTheDice`.

4.10 UML Klassendiagramme

Wir werden in dieser Vorlesung zwischendurch *UML Diagramme* verwenden, um die Struktur unserer Programme zu visualisieren[5]. UML steht für *Unified Modeling Language* und ist die populärste Notationssprache für objektorientiertes Programmdesign. Es existieren unterschiedlichste UML Typen – jeder Typ kann für unterschiedliche Aspekte bei der Visualisierung von objektorientierten Programmen genutzt werden. Wir fokussieren uns auf UML *Klassendiagramme*, um die Inhalte von Klassen und deren Beziehungen zu visualisieren.

In einem UML Klassendiagramm wird jede Klasse in einem Rechteck mit drei Bereichen repräsentiert. Der erste Bereich enthält den Klassennamen, der zweite Bereich enthält die Variablen und der dritte Bereich die Methoden der Klasse. In Abb. 4.5 ist das Klassendiagramm für unser `RollTheDice` Programm gezeigt.

Der Pfeil, der die beiden Klassen `RollTheDice` und `Dice` miteinander verbindet, zeigt an, dass eine Beziehung zwischen diesen beiden Klassen existiert. Eine gestrichelte Linie mit offenem Pfeilkopf bedeutet, dass die eine Klasse *abhängig* ist von der anderen Klasse (in diesem Fall *verwendet* die Klasse `RollTheDice` Methoden der Klasse `Dice`). Andere Beziehungstypen werden in UML mit anderen Typen von Pfeilen visualisiert (diese Pfeile werden im weiteren Verlaufe des Semesters eingeführt).

UML wurde nicht speziell für Java entwickelt (UML ist eine sprachunabhängige Notationsmöglichkeit). Die Syntax in UML Diagrammen ist deshalb nicht identisch mit der Syntax von Java.

[5]Beachten Sie das zusammenfassende *UML Mini Cheat Sheet* auf ILIAS.

- In UML Klassendiagrammen werden die Sichtbarkeiten der Variablen und Methoden mit einem Plus (+) für `public` oder einem Minus (−) für `private` visualisiert.
- Nach dem Bezeichner einer Variablen folgt nach einem Doppelpunkt der zugehörige Datentyp. Optional kann man nach einem Gleichheitszeichen noch einen initialen Wert der Variablen definieren.
- Die Parameter einer Methode werden in Klammern nach dem Bezeichner der Methode ebenfalls nach dem Muster *(Bezeichner : Datentyp)* aufgelistet.
- Nach den Parametern in Klammern wird nach einem Doppelpunkt der Rückgabetyp der Methode angegeben. Den Typ `void` gibt es in UML nicht. Das heisst, bei Methoden ohne Rückgabe wird in UML der Rückgabetyp einfach weggelassen.

4.11 Ein Weiteres Beispiel einer Eigenen Klasse

Wir schauen im Folgenden ein weiteres Beispiel einer eigenen Klasse an - die Klasse `PlayerCard` (→Video K-4.2). Diese Klasse modelliert das Konzept einer Spielerkarte, die man sammeln und tauschen kann (z.B. Karten von Fussball- oder Baseballspielern).

Betrachten Sie zuerst die Deklarationen der zwei Instanzvariablen `playerName` vom Typ `String` und `cardNumber` vom Typ `int`. Für beide Instanzvariablen sind *Getter* und *Setter* programmiert. Beachten Sie insbesondere, wie die Methode `setCardNumber` programmiert ist: Die Variable `cardNumber` wird nur dann geändert, wenn die neue Kartennummer grösser ist als `0`. Ansonsten sind diese Methoden recht ähnlich zu den *Gettern* und *Settern* aus dem vorigen Beispiel der Klasse `Dice`.

Natürlich könnten wir jetzt noch weitere Informationen über Spieler in den Karten modellieren (zum Beispiel die Nationalität, die Position, auf der der Spieler spielt, der Name des Stammclubs, der Jahrgang des Spielers, etc.). Wir werden dies in den folgenden Kapiteln zum Teil nachholen. Wichtig ist für den Moment, dass Sie das allgemeine Prinzip verstanden haben – um Vollständigkeit kümmern wir uns nicht.

Im Gegensatz zum Konstruktor der Klasse `Dice`, besitzt der Konstruktor `PlayerCard` zwei Parameter (`playerName` und `cardNumber`). Das heisst, wenn ein Objekt vom Typ `PlayerCard` instanziiert werden soll, müssen diese zwei Informationen als Argumente an den Konstruktor übergeben werden.

```java
public class PlayerCard {

        // Instanzvariablen für PlayerCard Objekte
        private String playerName;
        private int cardNumber;

        /**
         * Konstruktor für PlayerCard Objekte
         */
        public PlayerCard(String playerName, int cardNumber) {
                this.playerName = playerName;
                this.cardNumber = cardNumber;
        }

        /**
         * Konstruktor für PlayerCard Objekte ohne Kartennummer
         */
        public PlayerCard(String playerName) {
                this.playerName = playerName;
                this.cardNumber = 0;
        }

        /**
         * Getter und Setter für den Spielernamen
         */
        public String getPlayerName() {
                return this.playerName;
        }

        public void setPlayerName(String playerName) {
                this.playerName = playerName;
        }

        /**
         * Getter und Setter für die Kartennummer
         */
        public int getCardNumber() {
                return this.cardNumber;
        }

        public void setCardNumber(int cardNumber) {
                if (cardNumber > 0)
                        this.cardNumber = cardNumber;
                else
                        System.out.println(cardNumber + " ist ungültig!");
        }

        /**
         * Zeichenkettenrepräsentation für PlayerCard Objekte
         */
        public String toString() {
                return this.playerName + ", " + this.cardNumber;
        }
}
```

Ein neues Objekt vom Typ `PlayerCard` kann also folgendermassen deklariert und instanziiert werden:

```
PlayerCard pc1 = new PlayerCard("John", 27);
```

Die Instanzvariablen `playerName` und `cardNumber` des Objektes `pc1` werden dann im Konstruktor gemäss den übergebenen Parametern initialisiert.

Die Klasse `PlayerCard` verfügt über *zwei* Konstruktoren. Über den ersten Konstruktor werden beide Instanzvariablen gemäss den Parametern `playerName` und `cardNumber` initialisiert. Es ist aber durchaus vorstellbar, dass wir eine Spielerkarte ohne Kartennummer instanziieren wollen. Hierzu programmieren wir einen weiteren Konstruktor, so dass Instanziierungen auch nur mit dem Namen des Spielers möglich sind und die Kartennummer in diesem Fall standardmässig auf `0` gesetzt wird.

Die zwei Konstruktoren in der Klasse `PlayerCard` unterscheiden sich in der Anzahl formaler Parameter. Welcher der beiden Konstruktoren durch den `new`-Operator ausgeführt wird, hängt davon ab, wie viele Argumente dem Konstruktor übergeben werden. Mit `new PlayerCard("Mike", 17);` wird zum Beispiel der erste und mit `new PlayerCard("Penny");` der zweite Konstruktor aufgerufen.

Grundsätzlich sind Sie frei, so viele Konstruktoren zu definieren, wie Sie möchten – diese müssen sich aber in den formalen Parametern unterscheiden, damit bei jedem Aufruf des Konstruktors eindeutig ist, welcher der vorhandenen Konstruktoren angesteuert werden soll[6].

Schliesslich befindet sich noch eine Methode mit Bezeichner `toString` in der Klasse `PlayerCard`. In dieser Methode `toString` definieren wir eine Zeichenkette, die das entsprechende Objekt repräsentieren soll, und geben diese zurück (der Rückgabetyp der Methode `toString` ist also `String`). In unserem Beispiel geben wir beide Werte der Instanzvariablen `playerName` und `cardNumber` getrennt durch ein Komma als Zeichenkette zurück.

Die Methode `toString` ist eine spezielle Methode. Diese wird nämlich in gewissen Situationen automatisch aufgerufen. Zum Beispiel immer dann, wenn man mit einer `println` oder `print` Anweisung eine Objektvariable ausgibt. Das heisst, dass in

[6] Dieses Prinzip heisst *Überladen von Methoden* und wird später noch detailliert besprochen.

folgender Anweisung

```
System.out.println(pc1);
```

die Methode `toString` der Klasse `PlayerCard` aufgerufen wird, so dass Java weiss, welche Zeichenkette ausgegeben werden soll. Es ist meistens eine gute Idee, für eigene Klassen die Methode `toString` zu definieren (z.B. zu Testzwecken).

Eigener Quellcode sollte immer ausführlich getestet und die erwarteten Ergebnisse mit den tatsächlichen Ergebnissen verglichen werden. In folgendem Testprogramm wird die Klasse `PlayerCard` mit all ihren Funktionalitäten getestet (→Video K-4.2). Hierzu werden drei Objekte vom Typ `PlayerCard` instanziiert, ausgegeben und modifiziert.

```
public class PlayerCardTest {

        /**
         * Führt ein paar Tests in der Klasse PlayerCard aus
         */
        public static void main(String[] args) {

                // instanziieren von drei PlayerCard Objekten (mit beiden Konstruktoren)
                PlayerCard pc1 = new PlayerCard("John", 27);
                PlayerCard pc2 = new PlayerCard("Mike", 17);
                PlayerCard pc3 = new PlayerCard("Penny");

                // testen der Methode toString
                System.out.println("Alle Spielerkarten:");
                System.out.println(pc1);
                System.out.println(pc2);
                System.out.println(pc3);

                System.out.println();

                // testen der getter und setter
                System.out.println("Kartennummer von " + pc1.getPlayerName() +
                                ": " + pc1.getCardNumber());
                pc1.setPlayerName(pc1.getPlayerName() + " Foo");
                pc1.setCardNumber(55);
                System.out.println("Kartennummer von " + pc1.getPlayerName() +
                                 ": " + pc1.getCardNumber());

                System.out.println();

                // testen des setters für ungültige Eingaben
                pc2.setCardNumber(-99);
                System.out.println("Kartennummer bleibt 17? " + pc2.getCardNumber());
        }
}
```

Die Ausgabe dieses Programmes lautet:

```
Alle Spielerkarten:
John, 27
Mike, 17
Penny, 0

Kartennummer von John: 27
Kartennummer von John Foo: 55

-99 ist ungültig!
Kartennummer bleibt 17? 17
```

Kapitel 5

Klassen des Java API Verwenden (Teil 2)

Wir wenden uns nochmals dem Java API zu. Die in diesem Kapitel besprochenen Klassen sind so ausgewählt, dass man an ihnen einige erweiterte Konzepte der Programmierung illustrieren kann (nämlich *statische Methoden*, *Wrapper*, *Sammlungen von Objekten*, *Schleifen* und *generische Typen*).

5.1 Die Klasse `Math`

Die Klasse `Math` aus dem Java API bietet eine Vielzahl mathematischer Funktionen sowie einige nützliche Konstanten an. Diese Klasse gehört dem *Package* `java.lang` an (muss also nicht via `import` Anweisung importiert werden). Einige wichtige Methoden der Klasse `Math` sind:

- `static double abs(double num)`: Gibt den Absolutwert von `num` zurück.
- `static double cos(double angle)` Gibt den Cosinus des Winkels `angle` zurück (analog dazu existieren die Methoden `sin` und `tan`).
- `static long round(double num)`: Rundet `num` auf die am Nächsten liegende ganze Zahl auf oder ab.
- `static double ceil(double num)`: Gibt die kleinste ganze Zahl zurück, die grösser oder gleich `num` ist (aufrunden).
- `static double floor(double num)`: Gibt die grösste ganze Zahl zurück, die kleiner oder gleich `num` ist (abrunden).
- `static double pow(double x, double y)`: Gibt x^y zurück.

- `static double sqrt(double x)`: Gibt $\sqrt{x}$ zurück.
- `static double min(double x, double y)`: Gibt das Minimum von x und y zurück (analog dazu existiert die Methode `max`).
- `static double random()`: Gibt eine Zufallszahl zwischen `0.0` (inklusive) und `1.0` (exklusive) zurück.

Das Besondere an der Klasse `Math` ist, dass alle Methoden *statisch* deklariert sind (statische Methoden werden auch *Klassenmethoden* genannt). Dies sieht man am Schlüsselwort `static`, welches nach dem Sichtbarkeitsmodifikator im Methodenkopf zu finden ist. Der Methodenkopf der Methode `pow` sieht z.B. folgendermassen aus:

```
public static double pow(double x, double y)
```

Statische Methoden können aufgerufen werden, ohne dass vorher ein Objekt aus der entsprechenden Klasse instanziiert werden muss. Solche Methodenaufrufe programmiert man, indem der Klassenname (statt der Objektname) vor den Punktoperator und den Methodenbezeichner gestellt wird.

Beispiel 50 *Um zum Beispiel in einem Programm einen Wert* ***num*** *auf die nächste ganze Zahl abzurunden (und diesen Wert einer Variablen* ***rounded*** *zuzuweisen), können wir folgende Programmieranweisung definieren*[1]*:*

```
int rounded = (int) Math.floor(num);
```

In diesem Fall verwenden wir also den Klassennamen ***Math*** *vor dem Punktoperator statt dem Bezeichner eines instanziierten Objektes.*

Die Werte, welche die Methoden der Klasse `Math` zurückgeben, können direkt in arithmetischen Ausdrücken weiterverwendet werden.

Beispiel 51 *Die folgende Programmieranweisung*

```
Math.sqrt(1 - Math.pow(Math.sin(alpha), 2));
```

entspricht zum Beispiel der Formel $\sqrt{1-(\sin\alpha)^2}$

[1] Beachten Sie den *Cast* Operator, der die Rückgabe der Methode `floor` – ein `double` – in einen `int` konvertiert.

Beachten Sie, dass man den Methoden der Klasse `Math` eine ganze Zahl als Parameter mitgeben kann (obschon diese eigentlich einen Wert vom Typ `double` erwarten). Java konvertiert in diesem Fall die ganze Zahl automatisch in einen `double`. Folgende Anweisungen sind also beispielsweise erlaubt: `Math.pow(2, 3);` oder `Math.pow(2, 0.5);`

Neben mathematischen Funktionen speichert die Klasse `Math` auch wichtige mathematische Konstanten. Diese sind ebenfalls statisch deklariert und können auch via Klassennamen referenziert werden. Zum Beispiel kann π folgendermassen verwendet werden, um die Fläche eines Kreises mit Radius `r` zu berechnen:

```
double area = Math.pow(r, 2) * Math.PI;
```

Beispiel 52 *Folgender Quellcode zeigt ein interaktives Programm, das die Lösungen einer quadratischen Gleichung der Form $ax^2 + bx + c$ findet (→ Video K-5.1).*

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class QuadraticFormula {

        /**
         * Berechnet die Lösungen einer quadratischen Gleichung
         * (mit statischen Methoden der Klasse Math)
         */
        public static void main(String[] args) {
                // ax^2 + bx + c
                int a, b, c;
                double discriminant, x1, x2;

                Scanner scan = new Scanner(System.in);
                System.out.print("Koeffizient von x^2: ");
                a = scan.nextInt();
                System.out.print("Koeffizient von x: ");
                b = scan.nextInt();
                System.out.print("Konstante c: ");
                c = scan.nextInt();

                // b^2 - 4ac
                discriminant = Math.pow(b, 2) - (4 * a * c);

                x1 = ((-1*b) + Math.sqrt(discriminant)) / (2 * a);
                x2 = ((-1*b) - Math.sqrt(discriminant)) / (2 * a);

                DecimalFormat fmt = new DecimalFormat("0.###");

                System.out.println("Lösungen:");
                System.out.println("x_1: " + fmt.format(x1));
                System.out.println("x_2: " + fmt.format(x2));
        }
}
```

Eine mögliche Ein- und Ausgabe des Programmes wäre bspw.:

```
Koeffizient von x^2: 5
Koeffizient von x: 3
Konstante c: -4
Lösungen:
x_1: 0.643
x_2: -1.243
```

5.2 Wrapper Klassen

Wie bereits diskutiert, speichert Java Daten in zwei verschiedenen Variablentypen: mit Hilfe von primitiven Variablen (wie z.B. `int`, `double`, `char` oder `boolean`) oder mit Objektvariablen. Die Tatsache, dass Java zwei Kategorien von Variablen verwendet, kann in gewissen Situationen problematisch sein (zum Beispiel dienen einige Objekte als Behälter für andere Objekte und primitive Werte könnten nicht in solchen Behältern gespeichert werden).

Wir können eine primitive Variable mit einem Objekt umhüllen (engl. *to wrap*). Sogenannte *Wrapper-Klassen* repräsentieren oder umhüllen jeweils einen bestimmten primitiven Datentypen. Beispielsweise könnten Objekte der Klasse `Integer` ganze Zahlen (also `int` Werte) repräsentieren:

```java
public class Integer {

        // der Wert, der umhüllt wird
        private final int value;

        /**
         * Umhüllt den übergebenen Wert in einem Objekt
         */
        public Integer(int value) {
                this.value = value;
        }

        /**
         * Gibt den umhüllten Wert zurück
         */
        public int intValue() {
                return this.value;
        }
}
```

Etwas genauer: Ein Objekt vom Typ `Integer` speichert einen Wert vom Typ `int` als Instanzvariable `value`. Der Konstruktor dieser Wrapper-Klasse akzeptiert den primitiven Datenwert, den das Objekt umhüllen soll, als Parameter. Zum Beispiel:

```
Integer number = new Integer(40);
```

Nach dieser Deklaration und Instanziierung repräsentiert das Objekt `number` nun die ganze Zahl 40 als Objekt vom Typ `Integer`. Die so umhüllte Zahl kann nun überall dort in einem Programm verwendet werden, wo ein Objekt statt ein primitiver Datentyp nötig ist.

Tatsächlich müssen wir keine eigenen Wrapper-Klassen schreiben, da im Java API für jeden primitiven Datentypen bereits eine zugehörige Wrapper-Klasse vorhanden ist (alle definiert im *Package* `java.lang`):

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

Die Hauptaufgabe der Wrapper-Klassen ist es, einen primitiven Datenwert als Variable in einem Objekt zu speichern. Hierzu wird der zu umhüllende Wert dem entsprechenden Konstruktor mitgegeben[2].

Beispiel 53 *Folgende Anweisungen umhüllen den Wahrheitswert `true` bzw. die Zahl `4.567` mit einem Objekt vom Typ `Boolean` und vom Typ `Double`:*

```
new Boolean(true);
new Double(4.567);
```

Das Hin- und Herwechseln zwischen primitivem Datentyp und dem entsprechenden Wrapper Objekt ist in Java sehr einfach. *Autoboxing* bezeichnet die automatische Konvertierung eines primitiven Datentypen in ein entsprechendes Wrapper Objekt (ohne Aufruf des Konstruktors).

[2] Die Konstruktoren der Wrapper-Klassen gelten allerdings seit Version 9 von Java als veraltet (engl. *deprecated*) und sollten deshalb nicht mehr verwendet werden.

Beispiel 54 *Mit folgender Programmieranweisung wird der* `int` *Wert* `69` *in ein* `Integer` *Objekt konvertiert:*

```
Integer obj = 69; // automatische Instanziierung
```

Die umgekehrte Operation (genannt *Unboxing*) geschieht auch automatisch.

Beispiel 55 *Mit folgenden Anweisungen wird ein umhüllter Wert aus dem Wrapper Objekt* `obj` *extrahiert:*

```
Integer obj = 69;
int num = obj; // automatische Extraktion
```

Hinweis: Autoboxing und Unboxing ist natürlich nur zwischen primitiven Datentypen und den zugehörigen Wrapper-Klassen möglich.

Neben dem Umhüllen von primitiven Datentypen bieten die Wrapper-Klassen auch zahlreiche Methoden zur Verwaltung und Veränderung der gespeicherten primitiven Daten an. Viele dieser Methoden sind statisch deklariert und können somit ohne instanziiertes Objekt verwendet werden.

Beispielsweise enthält die Klasse `Integer` eine statische Methode mit Bezeichner `parseInt`, die eine ganze Zahl, die in einem Objekt vom Typ `String` gespeichert ist, in den korrespondierenden `int` Wert konvertiert.

Beispiel 56 *In folgenden Anweisungen wird die statische Methode* `parseInt` *dazu verwendet, das Objekt* `str` *vom Typ* `String` *in einen* `int` *Wert mit Bezeichner* `num` *zu konvertieren:*

```
String str = "987";
int num = Integer.parseInt(str);
```

Die Variable `num` *speichert nun die ganze Zahl* `987`[3].

Die anderen Wrapper-Klassen bieten ähnliche Methoden an (z.B. bieten die Klassen `Double` oder `Boolean` die Methoden `parseDouble` oder `parseBoolean` an).

Die umgekehrte Operation – also das Konvertieren vom primitiven Datentypen in eine Zeichenkette – wird auch von allen Wrapper-Klassen angeboten – mit der statischen Methode `toString`.

[3] Befindet sich in `str` keine ganze Zahl, löst `parseInt` einen Fehler zur Laufzeit aus.

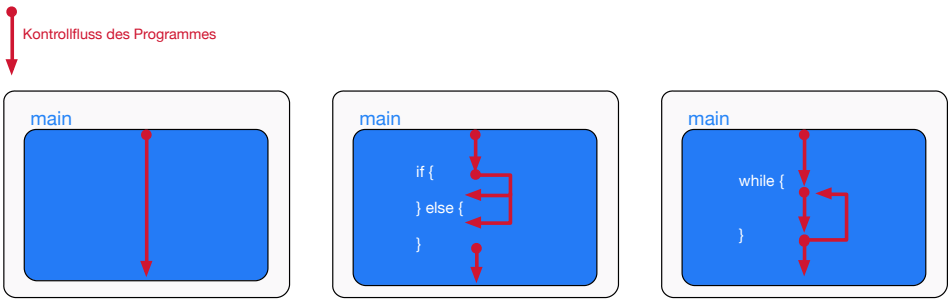

Abbildung 5.1: Der Kontrollfluss eines Programmes kann mit `if`- und `while`-Anweisungen beeinflusst werden.

Beispiel 57 *Folgende Anweisungen konvertieren eine ganze Zahl* `i` *in eine Zeichenkette:*

```
int i = 987;
String str = Integer.toString(i);
```

Einige der Wrapper-Klassen enthalten zudem statische Konstanten, die oftmals hilfreich sein können beim Programmieren. Beispielsweise enthalten die Klassen `Integer` und `Double` je zwei statische Konstanten `MIN_VALUE` und `MAX_VALUE`, welche den kleinst- und grösstmöglichen `int` bzw. `double` Wert speichern.

5.3 Die `while`-Anweisung

Wird ein Programm ausgeführt, so werden im Prinzip alle Programmieranweisungen der Methode `main` genau einmal ausgeführt (siehe Abb. 5.1). Wir haben schon gesehen, dass man mit einer `if`-Anweisung gewisse Teile des Quellcodes überspringen kann, bzw. aus zwei (oder mehr) Optionen eine ausführen lassen kann.

In diesem Abschnitt wollen wir nun erstmals *Schleifen* programmieren (mit `while`-Anweisungen). Schleifen erlauben uns, gewisse Programmieranweisungen mehrfach ausführen zu lassen (ohne diese mehrfach zu programmieren).

Eine `while`-Anweisung evaluiert eine Boolesche Bedingung genau gleich wie eine `if`-Anweisung. Wenn die Bedingung wahr ist, so wird die zugehörige Programmieranweisung ausgeführt. Anders als bei einer `if`-Anweisung wird der Kontrollfluss danach aber nicht zur nächsten Anweisung unterhalb der `while`-Anweisung springen, sondern wird die Boolesche Bedingung nochmals evaluieren. Falls diese immer noch wahr ist, wird die zugehörige Programmieranweisung nochmals ausgeführt. Diese

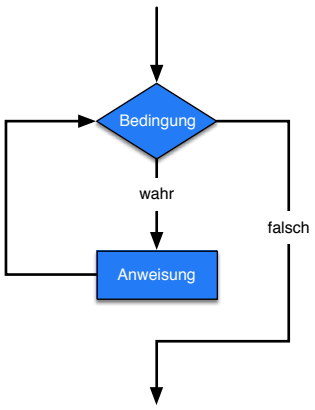

Abbildung 5.2: Die Logik von `while`-Schleifen.

Wiederholung wird solange fortgesetzt, bis die Bedingung irgendwann falsch wird. Erst dann wird mit der nächsten Anweisung unterhalb der `while`-Anweisung fortgefahren (siehe Abb. 5.2).

Beispiel 58 *Die folgende Schleife gibt zum Beispiel die Werte von `1` bis `3` aus:*

```
int count = 1;
while (count <= 3) {
        System.out.println(count);
        count++;
}
```

Beachten Sie, dass im obigen Beispiel zwei Anweisungen in einem Block innerhalb geschweifter Klammern zusammengefasst werden. Der ganze Block wird bei jeder Iteration durch die `while`-Schleife wiederholt.

Beispiel 59 *Das Programm `CashDesk` liest eine Reihe von Beträgen vom Benutzer ein, summiert diese und gibt die Summe der Eingaben aus (→ Video K-5.2). Wir verwenden die Variable `priceSum`, um die Benutzereingaben aufzusummieren und die Variable `count`, um die Anzahl Benutzereingaben zu zählen. In der Variablen `price` werden wir jeweils die aktuelle Benutzereingabe speichern.*

Da wir zu Beginn nicht wissen, wie viele Eingaben der Benutzer machen möchte, braucht der Benutzer eine Möglichkeit, das Programm zu beenden. Im Programm `CashDesk` definieren wir hierzu die Zahl `-1` als sogenannten Wächterwert (engl. sentinel value). Ein Wächterwert sollte immer ausserhalb der normalen Eingabemöglichkeiten liegen. Sobald der Benutzer diesen Wächterwert eingibt, soll die Eingabe beendet werden. Umgekehrt soll der Benutzer weitere Eingaben machen können, solange

der zuletzt eingegebene Wert **price** *ungleich* -1 *ist. Diese Idee programmieren wir mit Hilfe einer* **while**-*Schleife.*

Innerhalb der **while**-*Schleife wird zunächst die letzte Eingabe* **price** *zu* **priceSum** *addiert und die Variable* **count** *wird um eins erhöht. Danach wird eine neue Eingabe vom Benutzer entgegengenommen und wieder der Variablen* **price** *zugewiesen.*

Beachten Sie, dass der Benutzer direkt nach dem Start des Programmes -1 *eingeben könnte. Diese Situation fangen wir mit einer* **if**-*Anweisung ab. Bei mindestens einer gültigen Eingabe wird die Summe und die Anzahl der Eingaben ausgegeben.*

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class CashDesk {

        /**
         * Berechnet die Summe von eingegebenen Preisen
         */
        public static void main(String[] args) {

                double priceSum = 0;
                int count = 0;

                Scanner scan = new Scanner(System.in);
                System.out.print("Preis von Position " + count + " eingeben "
                                + "(-1 zum Beenden): ");
                double price = scan.nextDouble();

                while (price != -1) {
                        priceSum += price;
                        count++;
                        System.out.print("Preis von Position " + count + " eingeben "
                                        + "(-1 zum Beenden): ");
                        price = scan.nextDouble();
                }

                if (count == 0)
                        System.out.println("Keine Eingaben getätigt.");
                else {
                        DecimalFormat fmt = new DecimalFormat("CHF 0.00");
                        System.out.println("Anzahl Eingaben: " + count);
                        System.out.println("Summe der Eingaben: " + fmt.format(priceSum));
                }
        }
}
```

Eine mögliche Ein- und Ausgabe des Programmes lautet zum Beispiel:

```
Preis von Position 0 eingeben (-1 zum Beenden): 3.10
Preis von Position 1 eingeben (-1 zum Beenden): 5.90
```

```
Preis von Position 2 eingeben (-1 zum Beenden): 2.10
Preis von Position 3 eingeben (-1 zum Beenden): -1
Anzahl Eingaben: 3
Summe der Eingaben: CHF 11.10
```

Schleifen können auch dazu verwendet werden, um Programme robuster zu machen. Ein robustes Programm fängt potenzielle Probleme so elegant wie möglich ab. Ungültige Eingaben oder Divisionen durch Null sind typische Probleme, die man mit `if`- oder `while`-Anweisungen abfangen kann.

Beispiel 60 *Das Programm* `SavePercentage` *demonstriert, wie* `while`*-Schleifen dazu verwendet werden können, den Benutzer zu zwingen, eine gültige Eingabe zu machen. Der Kontrollfluss bleibt solange in den* `while`*-Schleifen stecken (jedes Mal mit der Aufforderung eine korrekte Eingabe zu machen), bis der Benutzer einen gültigen Wert eingibt (→ Video K-5.3).*

```java
import java.text.DecimalFormat;
import java.util.Scanner;

public class SavePercentage {

        /**
         * Berechnet die Fangquote eines Torhüters
         */
        public static void main(String[] args) {
                int saves, shots;

                Scanner scan = new Scanner(System.in);

                System.out.print("Anzahl Schüsse auf's Tor eingeben: ");
                shots = scan.nextInt();
                while (shots < 0) {
                        System.out.println("Anzahl muss >= 0 sein!");
                        System.out.print("Anzahl Schüsse auf's Tor eingeben: ");
                        shots = scan.nextInt();
                }

                System.out.print("Anzahl gehaltener Schüsse eingeben: ");
                saves = scan.nextInt();
                while (saves < 0 || saves > shots) {
                        System.out.println("Ungültiger Wert");
                        System.out.print("Anzahl gehaltener Schüsse eingeben: ");
                        saves = scan.nextInt();
                }

                double percentage = (double) saves / shots;
                DecimalFormat fmt = new DecimalFormat("#.##%");
                System.out.println("Fangquote in Prozent: " + fmt.format(percentage));
        }
}
```

Eine mögliche Ein- und Ausgabe des Programmes sieht folgendermassen aus:

```
Anzahl Schüsse auf's Tor eingeben: -6
Anzahl muss >= 0 sein!
Anzahl Schüsse auf's Tor eingeben: 28
Anzahl gehaltener Schüsse eingeben: 31
Ungültiger Wert!
Anzahl gehaltener Schüsse eingeben: 25
Fangquote in Prozent: 89.29%
```

Beachten Sie den Cast zur Berechnung von `percentage` *– weshalb ist dieser hier zwingend nötig?*

5.3.1 Endlosschleifen

Es liegt in der Verantwortung der Programmiererin, dass die Boolesche Bedingung einer Schleife irgendwann falsch wird. Wenn diese nie falsch wird, so werden die Anweisungen der `while`-Schleife für immer ausgeführt (bzw. bis das Programm abgebrochen wird). Eine solche Situation nennt man *Endlosschleife* und ist häufig – aber nicht immer – ein Programmierfehler.

Beispiel 61 *Hier ein Beispiel einer Endlosschleife:*

```
int count = 1;
while (count <= 1) {
        System.out.println(count);
        count--;
}
```

Wenn Sie diese oder ähnliche Schleifen programmieren und diese danach ausführen lassen, sollten Sie wissen, wie man Endlosschleifen bzw. laufende Programme terminieren kann[4].

5.3.2 Verschachtelte Schleifen

Jeder Schleifenblock kann eine weitere Schleife enthalten. Ein solches Konstrukt nennt man auch *verschachtelte Schleife*. Bei jedem Durchlauf durch die äussere Schleife wird die innere Schleife *komplett* abgearbeitet.

[4]Suchen Sie im WWW z.B. nach `terminate java in eclipse`.

Wie oft wird z.B. in der folgenden verschachtelten Schleife die Zeichenkette

`"Hello Again"`

ausgegeben?

```
int count1 = 1, count2;
while (count1 <= 5) {
   count2 = 1;
   while (count2 < 10) {
      System.out.println("Hello Again");
      count2++;
   }
   count1++;
}
```

Die `println` Anweisung ist innerhalb der inneren Schleife zu finden. Die äussere Schleife wird genau `5` mal ausgeführt (`count1` wächst in Einerschritten von `1` bis `5`). Die innere Schleife wird `9` mal ausgeführt (`count2` wächst auch in Einerschritten von `1` bis `9`). Bei jedem Durchlauf der äusseren Schleife wird die innere Schleife komplett abgearbeitet: Deshalb wird die Zeichenkette `"Hello Again"` $5 \times 9 = 45$ mal ausgegeben.

Beispiel 62 *Das Programm* `Multiples` *demonstriert, wie eine verschachtelte* `while`*-Schleife dazu verwendet werden kann, um Vielfache von Zahlen zu berechnen (→ Video K-5.4).*

```
public class Multiples {

        /**
         * Gibt Vielfache von Zahlen aus
         */
        public static void main(String[] args) {

                final int MAX1 = 5, MAX2 = 10;
                int val1 = 1, val2;

                while (val1 <= MAX1){
                        val2 = 1;
                        while (val2 <= MAX2){
                                System.out.print(val1 * val2 + "\t");
                                val2++;
                        }
                        val1++;
                        System.out.println();
                }
        }
}
```

Die Ausgabe des Programmes lautet:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50

Beispiel 63 *Im Programm* `GCD` *nutzen wir eine verschachtelte* `while`*-Schleife dazu, um die Berechnung des grössten gemeinsamen Teilers (ggT) mehrfach durchführen zu können (→ Video K-5.5).*

```
import java.util.Scanner;

public class GCD {

        /**
         * Berechnet den grössten gemeinsamen Teiler zweier Zahlen
         */
        public static void main(String[] args) {

                String another = "Y";
                Scanner scan = new Scanner(System.in);

                int value1, value2;

                while (another.equalsIgnoreCase("Y")) {
                        System.out.print("Erste Zahl: ");
                        value1 = scan.nextInt();
                        System.out.print("Zweite Zahl: ");
                        value2 = scan.nextInt();

                        // Euklids Verfahren zur Berechnung des ggTs
                        while (value1 != value2) {
                                if (value1 > value2)
                                        value1 = value1 - value2;
                                else
                                        value2 = value2 - value1;
                        }
                        int gcd = value1;
                        System.out.println("Der ggT Ihrer Zahlen ist: " + gcd);

                        System.out.println("Weitere Berechnung? (y/n)");
                        another = scan.next();
                }
        }
}
```

Die äussere `while`*-Schleife kontrolliert, ob der Benutzer eine weitere Berechnung durchführen möchte. Hierzu nutzen wir die Methode* `equalsIgnoreCase` *der Klasse*

`String`, welche auf dem Objekt `another` mit Parameter `"Y"` aufgerufen wird. Beachten Sie, dass am Ende der `while`-Schleife dem Benutzer die Möglichkeit gegeben wird, zu entscheiden, ob er eine weitere Berechnung durchführen möchte oder nicht.

Im Block der äusseren Schleife werden zunächst zwei Zahlen vom Benutzer eingelesen. Mit diesen Eingaben wird danach Euklids Verfahren zur Bestimmung des ggTs durchgeführt. Dies erfordert den Einsatz einer weiteren `while`-Schleife. In dieser inneren Schleife wird immer die kleinere der beiden Zahlen von der grösseren subtrahiert, bis schliesslich beide gleich gross sind – dies entspricht dem ggT der beiden Zahlen.

Ein mögliche Ein- und Ausgabe des Programmes wäre:

```
Erste Zahl: 36
Zweite Zahl: 48
Der ggT Ihrer Zahlen ist: 12
Weitere Berechnung? (y/n)
y
Erste Zahl: 15
Zweite Zahl: 3
Der ggT Ihrer Zahlen ist: 3
Weitere Berechnung? (y/n)
n
```

5.4 Sammlung von Objekten

Als Nächstes wollen wir Sammlungen von Objekten generieren, die wir dann zum Beispiel mit `while`-Schleifen bearbeiten können. Hierzu nutzen wir eine nützliche Klasse des Java APIs zur Behandlung von Sammlungen von Objekten: Die Klasse `ArrayList` aus dem *Package* `java.util`[5].

Die Klasse `ArrayList` verwendet das Konzept der *generischen Typen* (engl. *Generics*), die wir im folgenden Exkurs einführen.

[5] Wir werden später in Kapitel 11 auf weitere Möglichkeiten von Sammlungen eingehen.

5.4.1 Exkurs: Generics

Nehmen wir an, wir programmieren ein Spiel, in dem Raketen gewisse Dinge transportieren sollen. In einem ersten Versuch programmieren wir uns eine Klasse `DogRocket` nach folgendem Muster (→Video K-5.6):

```
public class DogRocket {

        // Transportgut
        private Dog cargo;

        /**
         * Konstruktor, getter und setter
         */
        public DogRocket(Dog cargo) {
                this.cargo = cargo;
        }

        public void set(Dog cargo) {
                this.cargo = cargo;
        }

        public Dog get() {
                return this.cargo;
        }
}
```

Das Problem dieser Lösung ist, dass die Klasse `DogRocket` nur Objekte vom Typ `Dog` transportieren kann. Wenn wir etwas anderes transportieren möchten (z.B. ein Objekt vom Typ `Human`), brauchen wir eine neue Klasse (die sehr ähnlich ist zur ersten Klasse):

```
public class HumanRocket {

        // Transportgut
        private Human cargo;

        /**
         * Konstruktor, getter und setter
         */
        public HumanRocket(Human cargo) {
                this.cargo = cargo;
        }

        public void set(Human cargo) {
                this.cargo = cargo;
        }

        public Human get() {
                return this.cargo;
        }
}
```

Diese Vorgehensweise scheint etwas zu unflexibel zu sein, da wir für alle unterschiedlichen Transportgüter je eine eigene Klasse programmieren müssen. Die Lösung für dieses Problem heisst *Generics*. Diese bieten Programmiererinnen die Möglichkeit, Klassen zu programmieren, die von einem Datentyp unabhängig, d.h. generisch sind.

Wollen wir `Rocket` in einen generischen Typ umbauen, so müssen wir an den Stellen, an denen `Dog` bzw. `Human` vorkam, einen *Typstellvertreter*, eine sogenannte *Typvariable*, einsetzen. Der Name der Typvariablen muss in der Klassendeklaration angegeben werden (in spitzen Klammern hinter dem Klassennamen):

```
public class Rocket<T> {

        private T cargo;

        public Rocket(T cargo) {
                this.cargo = cargo;
        }

        public void set(T cargo) {
                this.cargo = cargo;
        }

        public T get() {
                return this.cargo;
        }
}
```

Wir haben die Typvariable `T`[6] definiert und verwenden sie jetzt anstelle von `Dog` bzw. `Human` in der `Rocket`-Klasse. Die Typvariable kann nun überall dort genutzt werden, wo vorher ein herkömmlicher Typ stand.

Um die neue `Rocket`-Klasse nutzen zu können, müssen wir sie zusammen mit einem *Typargument* instanziieren. Dadurch entstehen sogenannte *parametrisierte Typen*:

```
Rocket<Dog> dogRocket = new Rocket<Dog>();
Rocket<Human> humanRocket = new Rocket<Human>();
```

Der konkrete Typ steht immer in spitzen Klammern hinter dem Klassennamen. Die Rakete `dogRocket` ist eine Instanz eines generischen Typs `Rocket<T>` mit dem konkreten Typargument `Dog`. Diese Rakete kann jetzt nur `Dog` Objekte enthalten (die Rakete `humanRocket` nur Objekte vom Typ `Human`).

[6] Typvariablen sind in der Regel einzelne Grossbuchstaben wie z.B. `T` oder `E`. Sie sind nur Platzhalter und keine wirklichen Typen.

Das Schöne an dieser Lösung ist, dass wenn wir z.B. aus `dogRocket` mit der Methode `get()` auf das Transportgut zugreifen, ist es automatisch vom Typ `Dog`.

```
Dog d = dogRocket.get();
```

Ferner kontrolliert der Kompilierer für uns, dass zum Beispiel der parametrisierte Typ `humanRocket` nur Objekte vom richtigen Typ transportieren wird. Zum Beispiel können der Methode `set` nur Objekte vom Typ `Human` übergeben werden:

```
Human h = new Human();
humanRocket.set(h);
```

In folgender Tabelle sind die Begriffe der *Generics* zusammengefasst:

Begriff	**Beispiel**
generischer Typ (engl. *generic type*)	`Rocket<T>`
Typvariable (engl. *formal type parameter*)	`T`
parametrisierter Typ (engl. *parameterized type*)	`Rocket<Human>`
Typargument (engl. *actual type parameter*)	`Human`
Originaltyp (engl. *raw type*)	`Rocket`

5.4.2 Die Klasse `ArrayList`

Ein Objekt vom Typ `ArrayList` speichert eine Liste von Objekten und erlaubt einem mit einem ganzzahligen Index auf die einzelnen Objekte der Liste zuzugreifen.

In folgender Liste sind wichtige Methoden der Klasse `ArrayList` zusammengefasst:

- `ArrayList<E>()`: Der Konstruktor instanziiert eine leere Liste, die Objekte vom Typ `E` aufnehmen kann.
- `boolean add(E obj)`: Fügt das Objekt `obj` vom Typ `E` ans Ende der Liste hinzu und gibt `true` zurück, wenn die Operation erfolgreich war.
- `void add(int i, E obj)`: Fügt das Objekt `obj` vom Typ `E` an Position `i` in der Liste hinzu und verschiebt das aktuell an dieser Position befindliche Element (falls vorhanden) und alle nachfolgenden Elemente nach rechts. Falls der Index `i` nicht gültig ist (`i < 0 || i > size()`), wird die Methode einen Fehler zur Laufzeit generieren.

- `E set(int i, E obj)`: Ersetzt das Objekt an Position `i` mit `obj`. Das ersetzte Objekt wird zurückgegeben. Falls der Index `i` nicht gültig ist, wird auch diese Methode einen Fehler generieren.
- `void clear()`: Entfernt alle Elemente aus der Liste.
- `E remove(int i)`: Entfernt das Element an Position `i` aus der Liste und gibt dieses zurück.
- `E get(int i)`: Gibt das Element an Position `i` aus der Liste zurück, ohne dieses zu entfernen.
- `int indexOf(E obj)`: Gibt den Index des Objektes `obj` zurück (sollte `obj` in der Liste mehrfach auftreten, wird nur der kleinste Index zurückgegeben – existiert `obj` nicht in dieser Liste, wird `-1` zurückgegeben).
- `boolean contains(E obj)`: Gibt `true` zurück, wenn die Liste das Objekt `obj` mindestens einmal enthält.
- `boolean isEmpty()`: Gibt `true` zurück, wenn die Liste leer ist.
- `int size()`: Gibt die Anzahl gespeicherter Objekte in der Liste zurück.

Beachten Sie, dass die Klasse `ArrayList` nur Objekte vom Typ `E` speichern kann. `E` entspricht also einer *Typvariablen*, die erst dann festgelegt wird, wenn ein neues Objekt vom Typ `ArrayList` instanziiert wird. Das heisst, man instanziiert nicht einfach ein Objekt vom Originaltyp `ArrayList` – man instanziiert i.d.R. einen parametrisierten Typ, welcher nur Objekte *eines* Typs aufnehmen kann.

Benötigen wir zum Beispiel eine Liste `names`, welche Objekte vom Typ `String` aufnehmen kann, deklarieren und instanziieren wir diese Liste folgendermassen:

```
ArrayList<String> names = new ArrayList<String>();
```

Benötigen wir eine Liste, welche Objekte vom Typ `PlayerCard` aufnehmen kann, sieht die Deklaration und Instanziierung der Liste folgendermassen aus:

```
ArrayList<PlayerCard> cards = new ArrayList<PlayerCard>();
```

Beachten Sie, dass ein Objekt vom Typ `ArrayList` nur Objekte, aber keine primitiven Daten speichern kann (Sie können also z.B. keine `ArrayList` instanziieren,

die `int` Werte speichert). Hier kommen die Wrapper-Klassen zum Zug. Wollen Sie z.B. eine Liste für ganze Zahlen definieren, deklarieren und instanziieren Sie eine Liste mit

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

Hinweis: Es ist auch möglich ein `ArrayList` Objekt ohne Typangabe zu instanziieren. Diese Liste ist dann prinzipiell offen für jeden Objekttypen. Dieses Vorgehen empfiehlt sich in der Regel nicht, da damit der Vorteil entfällt, dass der Kompilierer überprüft, ob nur Objekte des korrekten Typs eingefügt werden.

Wir instanziieren für folgende Überlegungen eine `ArrayList`, die `String` Objekte aufnehmen kann:

```
ArrayList<String> wishes = new ArrayList<String>();
```

Mit jedem Aufruf von `add` wird das übergebene Element ans Ende der Liste hinzugefügt.

```
wishes.add("Happy");
wishes.add("Birthday");
wishes.add("Stefan");
```

Die Indizes in einer `ArrayList` starten mit `0` (und nicht etwa mit `1`). Die `ArrayList` `wishes` kann man sich also folgendermassen vorstellen (siehe auch Abb. 5.3):

0. `"Happy"`
1. `"Birthday"`
2. `"Stefan"`

Bei einem Aufruf von `add` mit Indexangabe werden das aktuell an dieser Position befindliche Element und alle nachfolgenden Elemente nach unten (oder rechts) verschoben, während `set` das Element tatsächlich ersetzt. Nach folgenden Anweisungen:

```
wishes.add(1, "Joyful");
wishes.set(3, "Daniel");
```

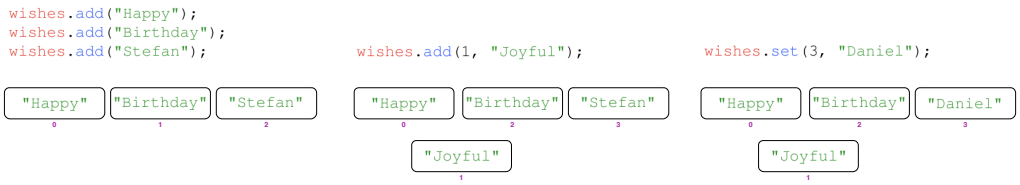

Abbildung 5.3: Das Hinzufügen von Objekten zu Listen mit `add` bzw. `set` illustriert.

hat sich die `ArrayList` `wishes` folgendermassen verändert (siehe auch Abb. 5.3):

0. `"Happy"`
1. `"Joyful"`
2. `"Birthday"`
3. `"Daniel"`

Beispiel 64 *Das folgende Programm* ***`BandMember`*** *instanziiert ein Objekt vom Typ* ***`ArrayList<String>`*** *und weist dieses der Variablen* ***`band`*** *zu. Danach werden die Methoden der Klasse* ***`ArrayList`*** *demonstriert (→ Video K-5.7).*

So wie bei jedem Objekt wird auch bei Objekten der Klasse ***`ArrayList`*** *automatisch die Methode* ***`toString`*** *aufgerufen, wenn man versucht, die Liste als Zeichenkette auszugeben. Die Methode* ***`toString`*** *ist in der Klasse* ***`ArrayList`*** *so programmiert, dass alle Objekte der Liste innerhalb eckiger Klammern als Zeichenkette zurückgegeben werden. Der erste Aufruf der Methode* ***`println`*** *führt also zu folgender Ausgabe:*

```
[Lauener, Mumenthaler, von Siebenthal, Fehlmann, Schmid]
```

Danach werden die Methoden ***`indexOf`****,* ***`remove`****,* ***`set`****,* ***`size`*** *und* ***`get`*** *demonstriert. Die Ausgabe des Programmes sieht folgendermassen aus:*

```
[Lauener, Mumenthaler, von Siebenthal, Fehlmann, Schmid]
[Lauener, von Siebenthal, Fehlmann, Schmid]
[Lauener, Etter, Fehlmann, Stäuble]
Anzahl Bandmitglieder: 4
An Position 0: Lauener
An letzter Position: Stäuble
```

```java
import java.util.ArrayList;

public class BandMember {

        /**
         * Demonstriert den Nutzen einer ArrayList
         */
        public static void main(String[] args) {

                ArrayList<String> band = new ArrayList<String>();

                band.add("Lauener");
                band.add("Mumenthaler");
                band.add("Fehlmann");
                band.add("Schmid");
                band.add(2, "von Siebenthal");
                System.out.println(band);

                int location = band.indexOf("Mumenthaler");
                band.remove(location);
                System.out.println(band);

                band.set(1, "Etter");
                band.set(3, "Stäuble");
                System.out.println(band);

                System.out.println("Anzahl Bandmitglider: "+ band.size());

                System.out.println("An Position 0: " + band.get(0));
                System.out.println("An letzter Position: " + band.get(band.size() - 1));

                int index = 0;
                while (index < band.size()) {
                        System.out.println(index +": " + band.get(index));
                        index++;
                }
        }
}
```

Beachten Sie den Einsatz der `while`-Schleife am Ende des Programmes `BandMember`, welche durch alle Objekte der Liste wandert und diese je auf einer Zeile ausgibt.

```
0: Lauener
1: Etter
2: Fehlmann
3: Stäuble
```

Für dieses "Durchwandern" aller Elemente einer Sammlung gibt es ein mächtiges Konzept in Java: *Iteratoren*.

5.4.3 Iteratoren

Ein *Iterator* ist ein Objekt, mit dem Sie eine Sammlung von Elementen nacheinander besuchen können. Genau genommen werden Iteratoren über die *Schnittstelle* `Iterator` definiert. Schnittstellen werden aber erst in Kapitel 9 eingeführt. Für den Moment ist nur wichtig, dass Sie wissen, dass Sie mit Iteratoren jedes Element einer Sammlung in einer bestimmten Reihenfolge besuchen und nach Bedarf verarbeiten können.

Die Schnittstelle `Iterator` ist im *Package* `java.util` deklariert und muss entsprechend importiert werden. Zudem werden Iteratoren auch als parametrisierte Typen deklariert und instanziiert (also z.B. `Iterator<String> iter`).

Jedes `Iterator` Objekt besitzt zwei wichtige Methoden:

- `boolean hasNext()`: Gibt `true` zurück, wenn es mindestens noch ein weiteres unbesuchtes Element in der Sammlung gibt.
- `E next()`: Gibt das nächste Element der Sammlung zurück.

Die Klasse `ArrayList<E>` bietet uns eine Methode `iterator()` an, welche ein Objekt vom Typ `Iterator<E>` zum Durchwandern der Liste zurückgibt.

Beispiel 65 *Wir können einen Iterator für die Liste* `band` *zum Beispiel folgendermassen referenzieren und verwenden:*

```
Iterator<String> iter = band.iterator();
while (iter.hasNext()) {
        System.out.println(iter.next());
}
```

Auch dieses Code-Fragment wandert durch alle Objekte der Liste und gibt diese je auf einer Zeile aus.

Kapitel 6

Bedingungen und Schleifen

Bis jetzt haben wir zwei Strukturen kennengelernt, welche den Kontrollfluss eines Programmes beeinflussen können: Die `if`- und die `while`-Anweisung. Die erste Kontrollstruktur ist eine sogenannte *Bedingung* (engl. *Conditional*), während die zweite Struktur eine *Schleife* (engl. *Loop*) darstellt.

Obschon man mit `if`- und `while`-Anweisungen sehr viele Situationen bei der Programmierung meistern kann, ist es eine gute Idee, die weiteren Kontrollstrukturen von Java für Bedingungen und Schleifen zu kennen[1].

Zuerst schauen wir uns zwei Alternativen zur `if`-Anweisung an – die `switch`-Anweisung und der sogenannte *Conditional Operator*. Die `switch`-Anweisung ist vor allem in Situationen hilfreich, in denen eine Verzweigung nicht nur zu zwei sondern zu vielen möglichen Optionen führen kann. Der *Conditional Operator* bietet bei Wertzuweisungen, die in einem `if-else` erfragt werden müssen, eine elegante Alternative.

Auch für die `while`-Anweisung betrachten wir zwei Alternativen: Die `do`-Schleife und die `for`-Schleife. Das erste Konzept ist vor allem für Schleifen geeignet, von denen man weiss, dass sie in jedem Fall mindestens einmal durchlaufen werden müssen. Die `for`-Schleife eignet sich sehr gut für Situationen, bei denen von Anfang an klar ist, wie oft der Schleifenrumpf ausgeführt werden soll.

[1] Tatsächlich kann man aber alle weiteren Kontrollstrukturen, die wir in diesem Kapitel betrachten, immer auch mit einer `if`- bzw. `while`-Anweisung programmieren.

6.1 Die switch-Anweisung

Bevor wir die switch-Anweisung besprechen, betrachten wir *verschachtelte* if-Anweisungen, also if- oder else-Anweisungen, welche wiederum if-Anweisungen enthalten. Solche verschachtelte if-Anweisungen erlauben uns, komplexe Entscheidungen zu treffen, die mehr als zwei Optionen umfassen.

Beispiel 66 *Die Klasse* ***MinFinder*** *verwendet verschachtelte* ***if****-Anweisungen, um die kleinste Zahl aus drei eingegebenen Zahlen zu finden (→ Video K-6.1).*

```
import java.util.Scanner;

public class MinFinder {

        /**
         * Findet das Minimum dreier eingegebener Zahlen
         */
        public static void main(String[] args) {
                // die drei Zahlen, aus denen das Minimum gefunden werden soll
                Scanner scan = new Scanner(System.in);
                System.out.print("Zahl 1 eingeben: ");
                int num1 = scan.nextInt();
                System.out.print("Zahl 2 eingeben: ");
                int num2 = scan.nextInt();
                System.out.print("Zahl 3 eingeben: ");
                int num3 = scan.nextInt();

                int min = 0;

                if (num1 < num2) {
                        if (num1 < num3)
                                min = num1;
                        else
                                min = num3;
                } else {
                        if (num2 < num3)
                                min = num2;
                        else
                                min = num3;
                }
                System.out.println("Kleinste Zahl: " + min);
        }
}
```

Mögliche Ein- und Ausgabe:

```
Zahl 1 eingeben: 17
Zahl 2 eingeben: -99
Zahl 3 eingeben: 1098
Kleinste Zahl: -99
```

Betrachten Sie die folgende verschachtelte `if`-Anweisung:

```
if (!full)
        if (pressure <= MAX)
                System.out.println("OK");
        else
                System.out.println("Problem mit dem Druck!");
```

Zu welchem der beiden `if`-Anweisungen gehört die `else`-Anweisung? Die Einrückung erweckt den Eindruck, dass diese zum zweiten `if` gehört und das ist in diesem Fall tatsächlich korrekt. Ein `else` gehört immer zur nächstgelegenen `if`-Anweisung, die noch zu keinem `else` gehört.

Klammern können dazu verwendet werden, um zu spezifizieren zu welcher `if`-Anweisung ein `else` gehören soll. Wollen wir das obige Beispiel so ändern, dass das `else` zur ersten `if`-Anweisung gehört, kann dies folgendermassen gemacht werden (Hinweis: Es würde nicht ausreichen, die Einrückung der Zeilen anzupassen!):

```
if (!full) {
        if (pressure <= MAX)
                System.out.println("OK");
} else
        System.out.println("Problem mit dem Füllung!");
```

Muss der Kontrollfluss einer von vielen verschiedenen Möglichkeiten folgen, führt dies typischerweise zu stark verschachtelten `if`-Anweisungen und eher unübersichtlichen Strukturen im Quellcode. In solchen Fällen ist die `switch`-Anweisung besser geeignet.

Eine `switch`-Anweisung führt dazu, dass der Kontrollfluss aufgrund eines einzigen Wertes nur einem von mehreren möglichen Pfaden folgt. Die `switch`-Anweisung liest zuerst eine Variable aus, um den `switch` Wert zu definieren:

```
switch(v)
```

Die Variable `v`, die in einer `switch`-Anweisung am Anfang in Klammern evaluiert wird, muss zwingend vom Typ `char`, `byte`, `short`, `int` oder eine Zeichenkette (also ein Objekt vom Typ `String`) sein. Danach wird dieser Wert mit den möglichen *Fällen* (engl. *cases*) verglichen. Jeder Fall (mit dem Schlüsselwort `case` programmiert) besitzt zugehörige Anweisungen, die nach einem Doppelpunkt `:` angegeben werden.

Beispiel 67 *Folgendes Code Fragment ist ein Beispiel für eine* `switch`*-Anweisung:*

```
switch(id) {
        case 'q' :
                qCount++;
                break;
        case 'w' :
                wCount++;
                break;
        case 'e' :
                eCount++;
                break;
        default :
                System.out.println("Fehlerhafter Wert!");
}
```

In obigem Beispiel ist der `switch` Wert, der evaluiert werden muss, eine Variable vom Typ `char` mit Bezeichner `id`.

Der Kontrollfluss des Programmes springt nach der Auswertung der Variablen `id` zu dem `case`, dessen Wert der Variablen `id` entspricht. Wenn in unserem Beispiel `id == 'q'` wahr ist, dann wird `qCount` um eins erhöht. Wenn `id == 'w'` wahr ist, wird der Fall `'q'` übersprungen und es wird `wCount` um eins erhöht (analog wird für den Fall, dass `id == 'e'` wahr ist, der Kontrollfluss direkt zum `case 'e'` springen).

Wenn keiner der angegebenen Fälle dem evaluierten Wert entspricht (z.B. wenn `id` aktuell `'A'` speichert), so springt der Kontrollfluss zum optionalen *Standard Fall* (angegeben mit dem reservierten Wort `default`). Existiert kein `default` Fall, wird in einer solchen Situation *keine* Anweisung der gesamten `switch`-Anweisung durchgeführt. Es ist meistens eine gute Idee, einen `default` Fall zu programmieren (auch dann, wenn man davon ausgehen darf, dass dieser Fall eigentlich gar nie eintreten kann).

Wird innerhalb einer Schleife oder einer Bedingung eine `break`-Anweisung angetroffen, springt der Kontrollfluss direkt zur ersten Anweisung unterhalb der Schleife oder der Bedingung (mit anderen Worten, die Kontrollstruktur wird sofort verlassen). In `switch`-Anweisungen bedeutet dies, dass wenn der Kontrollfluss in einem `case` auf eine `break`-Anweisung trifft, die restlichen Teile der gesamten `switch`-Anweisung übersprungen werden. Dies ist insbesondere dann hilfreich, wenn auf jeden Fall nur *genau ein* `case` ausgeführt werden soll.

In unserem Beispiel oben wird bei jedem Programmdurchlauf also *genau ein* Fall

('q' oder 'w' oder 'e') bearbeitet, da am Ende jedes Falles ein break vorhanden ist. Wird am Ende eines Falles aber keine break-Anweisung platziert, springt der Kontrollfluss zum *nächsten* Fall der switch-Anweisung.

Oft möchte man in einer switch-Anweisungen jeweils nur genau einen der vorhandenen Fälle ausführen lassen (und deshalb enthalten switch-Anweisungen recht häufig am Ende jedes Falles eine break-Anweisung). Es gibt aber durchaus Situationen, in denen der Kontrollfluss mit dem nächsten Fall weiterfahren soll.

Beispiel 68 *In folgender* switch*-Anweisung ist kein* break *vorhanden:*

```
switch(input) {
        case 'A' :
                System.out.print("A ");
        case 'B' :
                System.out.print("B ");
        case 'C' :
                System.out.print("C ");
        case 'D' :
                System.out.print("D ");
}
```

Für den Fall, dass input *bei der Ausführung dieses Code Fragmentes* 'A' *speichert, ist die Ausgabe des Programmes:*

```
A B C D
```

Für den Fall, dass input *aktuell* 'C' *speichert, ist die Ausgabe des Programmes hingegen:*

```
C D
```

Beispiel 69 *Im Programm* Report *wird eine schriftliche Beurteilung durch die Formel*

$$\frac{\textit{Erreichte Punkte}}{\textit{Mögliche Punkte}} \times 5 + 1$$

bestimmt. Das Resultat dieser Berechnung ist eine Schulnote zwischen 1 und 6. Die gerundete Schulnote category *wird als Ausdruck für die* switch*-Anweisung verwendet. Diese Anweisung gibt für verschiedene Punktezahlen unterschiedliche Aussagen zur Leistung aus (→ Video K-6.2).*

```java
import java.util.Scanner;

public class Report {

        /**
         * Nimmt eine Anzahl erreichter Punkte entgegen und
         * gibt eine schriftliche Beurteilung aus
         */
        public static void main(String[] args) {

                final int MAX = 35;
                double points;
                int category;

                Scanner scan = new Scanner(System.in);

                System.out.print("Anzahl erreichte Punkte (0 bis " + MAX + "): ");
                points = scan.nextInt();

                while (points < 0 || points > MAX) {
                        System.out.print("Anzahl erreichte Punkte (0 bis " + MAX + "): ");
                        points = scan.nextInt();
                }

                category = (int) Math.round(points / MAX * 5 + 1);

                System.out.print("Das ist ");
                switch (category) {
                case 6 :
                        System.out.println("ausgezeichnet.");
                        break;
                case 5:
                        System.out.println("gut.");
                        break;
                case 4:
                        System.out.println("genügend.");
                        break;
                default:
                        System.out.println("ungenügend.");
                }
        }
}
```

Eine mögliche Ein- und Ausgabe des Programmes lautet:

```
Anzahl erreichte Punkte (0 bis 35): 36
Anzahl erreichte Punkte (0 bis 35): 27
Das ist gut.
```

6.2 Der *Conditional Operator*

Der *Conditional Operator* von Java ist ein *ternärer Operator*, der drei Operanden benötigt. Betrachten Sie beispielsweise folgende Anweisung:

```
(points > MAX) ? points + 1 : points * 2;
```

Vor dem Fragezeichen ? steht eine Boolesche Bedingung[2]. Danach folgen zwei Ausdrücke, die mit einem Doppelpunkt (:) getrennt sind. Der gesamte Operator gibt den Wert des ersten Ausdruckes zurück, wenn die Boolesche Bedingung wahr ist und sonst den zweiten Ausdruck.

Üblicherweise möchte man etwas mit dieser Rückgabe tun, diese z.B. einer Variablen zuweisen:

```
points = (points > MAX) ? points + 1 : points * 2;
```

Man kann diese Anweisung auch in eine `if-else`-Anweisung übersetzen:

```
if (points > MAX)
        points = points + 1;
else
        points = points * 2;
```

Beispiel 70 *In folgendem Programm wird die Ausgabe der Anzahl Versuche mit einem Conditional Operator grammatikalisch korrekt definiert (→Video K-6.3).*

```
import java.util.Scanner;

public class NumberOfAttempts {

        /**
         * Gibt die Anzahl Versuche aus
         */
        public static void main(String[] args) {
                Scanner scan = new Scanner(System.in);
                System.out.print("Anzahl: ");
                int tries = scan.nextInt();

                System.out.println(tries + (tries == 1 ? " Versuch" : " Versuche"));
        }
}
```

[2]Die Klammern sind nicht zwigend nötig, unterstützen aber manchmal die Lesbarkeit.

Bei einer Eingabe von 1 ist die Ausgabe z.B.:

```
Anzahl: 1
1 Versuch
```

Während bei einer Eingabe von 0 oder einer Eingabe, die grösser als 1 ist, die Ausgabe angepasst wird:

```
Anzahl: 17
17 Versuche
```

6.3 Die do-Anweisung

Sie erinnern sich, dass `while`-Anweisungen zuerst eine Boolesche Bedingung evaluieren und danach die Anweisungen in der zugehörigen Blockanweisung so lange wiederholen, bis die Bedingung falsch wird. Eine *`do`-Schleife* funktioniert sehr ähnlich.

Eine `do`-Schleife startet mit dem reservierten Wort `do`. Danach folgt die Anweisung (oder mehrere Anweisungen in einem Block), die ausgeführt werden soll. Mit einer Booleschen Bedingung wird am Ende der `do`-Schleife überprüft, ob die `do`-Anweisung weiter ausgeführt werden soll oder nicht.

Beispiel 71 *Die folgende Schleife gibt die Werte von 1 bis 3 mit einer `do`-Schleife aus:*

```
int count = 1;
do {
        System.out.println(count);
        count++;
} while (count <= 3);
```

Der Unterschied zur `while`-Anweisung ist, dass die Boolesche Bedingung ans Ende der Schleife gesetzt wird. Dies um zu verdeutlichen, dass eine `do`-Schleife immer *mindestens einmal* ausgeführt wird (siehe Abb. 6.1). Im Gegensatz hierzu kann eine `while`-Schleife möglicherweise auch gar nie ausgeführt werden. Wenn Sie also wissen, dass ein bestimmtes Code-Fragment möglicherweise mehrere Male, aber mindestens einmal ausgeführt werden soll, ist eine `do`-Schleife einer `while`-Schleife vorzuziehen.

Beispiel 72 *Beachten Sie folgende Anweisungen:*

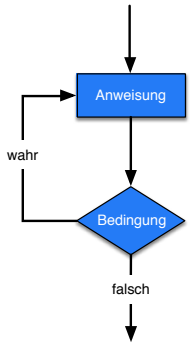

Abbildung 6.1: Die Logik einer do-Schleife.

```
System.out.print("Erreichte Punkte (0 bis 100): ");
points = scan.nextInt();
while (points < 0 || points > 100) {
        System.out.print("Erreichte Punkte (0 bis 100): ");
        points = scan.nextInt();
}
```

Offensichtlich muss sowohl die **print** *Anweisung als auch das Einlesen mit* **nextInt** *in jedem Fall mindestens einmal durchgeführt werden. Mit Hilfe einer* do*-Schleife lässt sich das eleganter programmieren:*

```
do {
        System.out.print("Erreichte Punkte (0 bis 100): ");
        points = scan.nextInt();
} while (points < 0 || points > 100);
```

6.4 Die for-Anweisung

Die while- und do-Anweisungen sind dann nützlich, wenn man initial nicht weiss, wie oft eine bestimmte Schleife durchgeführt werden soll. Die for-Anweisung ist eine weitere Möglichkeit, Anweisungen mehrmals ausführen zu lassen. Diese Schleife ist insbesondere in Situationen geeignet, in denen bereits zu Beginn klar ist, wie oft die Schleife durchlaufen werden soll.

Beispiel 73 *Folgendes Code Fragment gibt die Zahlen 1 bis 3 mit Hilfe einer* for*-Schleife aus:*

```
for (int count = 1; count <= 3; count++)
        System.out.println(count);
```

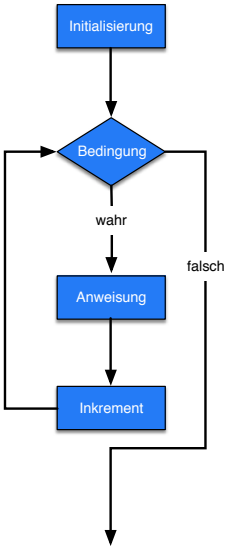

Abbildung 6.2: Die Logik einer `for` Schleife.

Der Kopf einer `for`-Schleife enthält drei durch Semikolon getrennte Teile: Die Initialisierung, die Boolesche Bedingung und das Inkrement:

- Bevor die Schleife ausgeführt wird, wird der erste Teil des Kopfes, die *Initialisierung* der sogenannten *Laufvariablen*, durchgeführt (in obigem Beispiel ist die Laufvariable `count` und diese wird mit `1` initialisiert). Dieser Teil des Schleifenkopfes wird nur *einmal* ausgeführt.
- Der zweite Teil des Kopfes enthält eine *Boolesche Bedingung,* die *vor jedem* Eintritt in die Schleife evaluiert wird. Nur wenn die Boolesche Bedingung (noch) wahr ist, werden die Anweisungen der `for`-Schleife ausgeführt (ähnlich wie bei einer `while`-Schleife).
- Immer wenn der Schleifenblock komplett abgearbeitet ist, wird der dritte Teil des Kopfes – das *Inkrement* – ausgeführt. Das heisst, während die Initialisierung nur einmal ausgeführt wird, wird das Inkrement *nach jedem* Schleifendurchgang ausgeführt.

Der Prozess einer `for`-Schleife ist in Abb. 6.2 illustriert.

In obigem Beispiel wird in der Initialisierung die Laufvariable `count` mit dem Wert `1` initialisiert. Beachten Sie, dass wir nebst der Initialisierung die Variable auch noch deklarieren. Dies ist nicht zwingend. Das heisst, man könnte die Laufvariable `count` auch schon vorher ausserhalb der `for`-Schleife deklarieren:

```
int count;
for (count = 1; count <= 3; count++)
        System.out.println(count);
```

Da in unserem ersten Beispiel die Variable `count` innerhalb der `for`-Schleife deklariert wird, existiert diese Variable auch nur innerhalb der `for`-Schleife (`count` kann also nur innerhalb des Schleifenblocks referenziert werden). Auch wenn die Laufvariable innerhalb des Schleifenblocks referenziert und verwendet werden kann, sollte diese *nicht* innerhalb des Schleifenblocks verändert werden – das Verändern der Laufvariablen sollte ausschliesslich im Inkrement des Schleifenkopfes stattfinden.

Das Inkrement der `for`-Schleife kann die Laufvariable nicht nur inkrementieren, sondern auch dekrementieren.

Beispiel 74 *Folgende `for`-Schleife gibt die Zahlen von `100` bis `1` rückwärts aus:*

```
int count;
for (count = 100; count > 0; count--)
        System.out.println(count);
```

Tatsächlich kann man im Inkrement einer `for`-Schleife beliebige Veränderungen der Laufvariablen ausführen.

Beispiel 75 *Einige Beispiele von möglichen Veränderungen einer Laufvariablen `i` im Inkrement einer `for`-Schleife:*

- *`i += 3` (erhöht die Laufvariable in jedem Durchgang um `3`)*
- *`i *= 2` (verdoppelt die Laufvariable in jedem Durchgang)*
- *`i -= 10` (verkleinert die Laufvariable in jedem Durchgang um `10`)*

Beispiel 76 *Das Programm `Multiples` gibt Vielfache einer vom Benutzer definierten Zahl `num` bis zu einer gewissen Grenze `limit` aus. Im Inkrement der `for`-Schleife wird in diesem Beispiel in jeder Iteration der Wert der eingegebenen Zahl `num` zur Laufvariablen `i` addiert (→ Video K-6.4).*

Eine mögliche Ausgabe des Programmes wäre:

```
Vielfache welcher Zahl? 3
Bis zu welcher Grenze? 40
Vielfache von 3 (bis max. 40):
3 6 9 12 15 18 21 24 27 30 33 36 39
```

```java
import java.util.Scanner;

public class Multiples {

        /**
         * Nimmt eine Zahl und eine Grenze entgegen und
         * gibt alle Vielfache der Zahl bis zur Grenze aus
         */
        public static void main(String[] args){

                Scanner scan = new Scanner(System.in);

                System.out.print("Vielfache welcher Zahl? ");
                int num = scan.nextInt();
                System.out.print("Bis zu welcher Grenze? ");
                int limit = scan.nextInt();

                System.out.println("Vielfache von " + num + " (bis max. " + limit + "):");

                for (int i = num; i <= limit; i += num) {
                        System.out.print(i + "\t");
                }
        }
}
```

Natürlich kann man auch `for`-Schleifen verschachteln. Und auch hier gilt: In jedem Durchlauf der äusseren `for`-Schleife, wird die innere Schleife komplett abgearbeitet.

Beispiel 77 *Das Programm* `Stars` *zeigt ein einfaches Beispiel einer verschachtelten* `for`*-Schleife. Die äussere Schleife wird 10 mal ausgeführt. In jedem Durchlauf einer äusseren Schleife wird mit Hilfe einer inneren* `for`*-Schleife eine Ausgabezeile produziert (→ Video K-6.5).*

```java
public class Stars {

        /**
         * Gibt 10 Zeilen mit Sternen aus, wobei auf
         * der i-ten Zeile i Sterne gezeigt werden
         */
        public static void main(String[] args) {

                final int MAX_ROWS = 10;

                for (int i = 1; i <= MAX_ROWS; i++) {
                        for (int j = 1; j <= i; j++)
                                System.out.print("*");
                        System.out.println();
                }
        }
}
```

Die innere Schleife führt eine variable Anzahl Durchgänge durch, die abhängig vom aktuellen Wert der Laufvariablen **i** *der äusseren Schleife ist – in Durchgang 4 zum Beispiel wird die innere Schleife 4 mal ausgeführt. Jeder Durchgang der inneren Schleife gibt einen Stern * aus (ohne Zeilenumbruch).*

Die Ausgabe des Programmes lautet:

```
*
**
***
****
*****
******
*******
********
*********
**********
```

6.4.1 Die `for`-`each` Schleife

Nehmen wir an, dass wir eine `ArrayList band` instanziiert haben, die Objekte vom Typ `String` aufnehmen kann:

```
ArrayList<String> band = new ArrayList<String>();
```

Sie erinnern sich, wie man mit Hilfe einer `while`-Schleife durch diese Liste wandern kann, um zum Beispiel alle Elemente der Liste auszugeben:

```
int index = 0;
while (index < band.size()) {
        System.out.println(band.get(index));
        index++;
}
```

Das Gleiche kann man mit einer `for`-Schleife folgendermassen lösen:

```
for (int index = 0; index < band.size(); index++)
        System.out.println(band.get(index));
```

Alternativ steht uns auch ein Iterator zur Verfügung, der ebenfalls das Gleiche erledigt:

```
Iterator<String> iter = band.iterator();
while (iter.hasNext()) {
        System.out.println(iter.next());
}
```

Eine Variation der `for`-Schleife, genannt `for-each` Anweisung, ist hilfreich im Zusammenhang mit der Klasse `ArrayList` und mit Arrays (die im nächsten Kapitel eingeführt werden). Eine `for-each` Anweisung kann für jedes Objekt verwendet werden, das die Schnittstelle `Iterable` implementiert. Diese Schnittstelle eliminiert die Notwendigkeit, einen Zähler zu verwenden (wie bei `for`-Schleifen) oder die Methoden `hasNext` und `next` aufzurufen (wie bei Iteratoren).

Mit folgender `for-each` Schleife wird ebenfalls jedes `String` Objekt der Liste `band` besucht und ausgegeben:

```
for (String bandMember : band)
        System.out.println(bandMember);
```

Eine solche Schleife kann folgendermassen interpretiert werden. Für jedes `String` Objekt `bandMember` aus der Liste `band`, gib `bandMember` aus. Die Variable `bandMember` referenziert in jedem Durchlauf automatisch das nächste `String` Objekt der Liste `band`. Das heisst, im ersten Durchlauf entspricht `bandMember` dem Objekt `band.get(0)`, im zweiten Durchlauf referenziert `bandMember` dann `band.get(1)`, usw.

6.5 Daten Vergleichen

Schleifen und Bedingungen basieren auf Booleschen Ausdrücken. Bei fast all unseren Beispielen haben wir jeweils ganze Zahlen miteinander verglichen, um solche Booleschen Ausdrücke zu definieren (z.B. `(i < 20)`). Werden Daten in Booleschen Ausdrücken verglichen, müssen einige Feinheiten des Datenvergleichs gut verstanden werden. Wir betrachten im Folgenden vier Schlüsselsituationen des Datenvergleichs.

6.5.1 Vergleich von Gleitkommazahlen

Werden zwei Gleitkommazahlen (also Variablen oder Werte vom Typ `double`) miteinander verglichen, sollte man wenn möglich auf den Gleichheitsoperator (`==`) verzichten. Das Problem ist, dass zwei Gleitkommazahlen nur dann als gleich angesehen werden, wenn deren unterliegenden binären Repräsentationen exakt identisch sind.

Dies ist in manchen Situationen problematisch – insbesondere wenn einer der beiden zu vergleichenden Variablen das Resultat einer Berechnung ist. Bei mathematischen Berechnungen können nämlich sehr kleine, irrelevante Unterschiede in den Resultaten entstehen:

Beispiel 78 *Folgende Anweisung*

```
System.out.println(2.12 * 2.5);
```

kann zum Beispiel folgende Ausgabe erzeugen

```
5.300000000000001
```

In vielen praktischen Anwendungen würde man zwei Gleitkommazahlen wie `5.3` und `5.300000000000001` wohl als "gleich" betrachten, da diese genügend nah voneinander entfernt liegen (auch wenn diese nicht exakt identisch sind).

Eine bessere Möglichkeit zum Vergleich zweier Gleitkommazahlen `num1` und `num2` ist es, eine Konstante `TOLERANCE` zu definieren:

```
final double TOLERANCE = 0.00000001;
```

Wenn nun die Absolutdifferenz zwischen `num1` und `num2` kleiner als diese Konstante ist, können die zwei Zahlen als "gleich" interpretiert werden:

```
if (Math.abs(num1 - num2) < TOLERANCE)
        System.out.println("Praktisch gleich!");
```

6.5.2 Vergleich von Zeichen

Der Java Zeichensatz basiert auf *Unicode*. In Unicode wird jedem Zeichen ein numerischer Wert zugewiesen. Werden in Java Zeichen – also Daten vom Typ `char` – miteinander verglichen, basieren diese Vergleiche immer auf der zugehörigen numerischen Codierung durch Unicode.

Glücklicherweise sind die Ziffern sowie die Gross- und Kleinbuchstaben in Unicode aufsteigend sortiert codiert:

Zeichen	Unicode Wert
'0' – '9'	48 – 57
'A' – 'Z'	65 – 90
'a' – 'z'	97 – 122

Beispiel 79 *Das Zeichen* `'5'` *ist zum Beispiel kleiner als das Zeichen* `'B'`*, weil dieses im Unicode Zeichensatz vor* `'B'` *steht, das Zeichen* `'Q'` *ist kleiner als das Zeichen* `'a'`*, weil dieses einen kleineren Unicode Wert besitzt.*

6.5.3 Vergleich von Zeichenketten

Der Gleichheitsoperator (`==`) kann zwar auf Objekten vom Typ `String` angewendet werden, man sollte aber darauf verzichten.

Beispiel 80 *Betrachten Sie die folgenden Anweisungen. Welche Ausgabe würde man erwarten?*

```
String name1 = new String("Michael");
String name2 = new String("Michael");
if (name1 == name2)
        System.out.println("Gleiche Namen!");
```

Tatsächlich wird in diesem Beispiel `Gleiche Namen!` *nicht* ausgegeben. Der Gleichheitsoperator (==) gibt nur dann `true` zurück, wenn die beiden `String` Objekte Aliase voneinander sind (d.h. wenn die Referenzvariablen `name1` und `name2` also auf das gleiche Objekt zeigen) – dies ist in unserem Beispiel nicht gegeben.

Beispiel 81 *Folgende Anweisung gibt* `Gleiche Namen!` *aus, da* `name1` *und* `name2` *Aliase sind.*

```
String name1 = new String("Michael");
String name2 = name1;
if (name1 == name2)
        System.out.println("Gleiche Namen!");
```

Zur Lösung dieses Problems bietet die Klasse `String` die Methode `equals` an, die zwei `String` Objekte auf Gleichheit überprüft (diese Methode haben wir bereits kennengelernt). Die Methode `equals` wird auf einem `String` Objekt aufgerufen und erhält als Parameter ein zweites Objekt vom Typ `String`. Die Methode gibt `true` zurück, wenn die beiden Zeichenketten die gleichen Zeichen in der gleichen Reihenfolge beinhalten.

Beispiel 82 *Folgende Anweisung gibt nun also* `Gleiche Namen!` *aus, da* `name1` *und* `name2` *die gleichen Zeichen in der gleichen Reihenfolge gespeichert haben:*

```
String name1 = new String("Michael");
String name2 = new String("Michael");
if (name1.equals(name2))
        System.out.println("Gleiche Namen!");
```

Die Operatoren > und < sind in Java für Zeichenketten nicht definiert und können nicht verwendet werden. Allerdings existiert neben der Methode `equals` in der Klasse `String` auch die Methode `compareTo`, die nach folgendem Prinzip arbeitet:

Der Methodenaufruf

```
name1.compareTo(name2);
```

gibt eine ganze Zahl zurück, welche

- Null ist, wenn die beiden Objekte `name1` und `name2` identisch sind.
- negativ ist, wenn das aufrufende Objekt `name1` *kleiner* ist als das Objekt `name2`, das als Parameter übergeben wird.
- positiv ist, wenn das aufrufende Objekt `name1` *grösser* ist als das Objekt `name2`, das als Parameter übergeben wird.

Beispiel 83 *Folgendes Code Fragment verwendet die Methode* `compareTo` *zum Vergleich von zwei Zeichenketten* `name1` *und* `name2`*:*

```
int compareValue = name1.compareTo(name2);
if (compareValue == 0)
        System.out.println("Gleiche Namen!");
else
        if (compareValue < 0)
                System.out.println(name1 + " ist kleiner!");
        else
                System.out.println(name2 + " ist kleiner!");
```

Die Berechnung der Rückgabe der Methode `compareTo` basiert auf dem Unicode Zeichensatz. Hierbei werden einzelne Zeichen von links nach rechts stellenweise miteinander verglichen, bis ein Unterschied in den Zeichen festgestellt werden kann. Sind alle Zeichen gross oder klein geschrieben, entspricht dies der alphabetischen Ordnung.

Beispiel 84 *Zum Beispiel sind die Rückgaben der folgenden zwei Anweisungen*

```
"abba".compareTo("zumba");
"abba".compareTo("aha");
```

beide negativ, weil in der alphabetischen Ordnung `abba` *vor* `zumba` *und auch vor* `aha` *steht.*

Vergleicht man aber zum Beispiel `"abraham"` mit `"George"`, so wird die Methode `compareTo` angeben, dass `"George"` kleiner ist als `"abraham"`. Sie erinnern sich: Alle Grossbuchstaben haben einen kleineren Unicode Wert als Kleinbuchstaben (es gilt also, dass der Ausdruck `('G' < 'a')` wahr ist). Eine Zeichenkette, die *Präfix* einer anderen Zeichenkette ist, ist immer kleiner als die längere Zeichenkette (z.B. ist `"Haus"` gemäss `compareTo` kleiner als `"Hausdach"`).

6.5.4 Vergleich von Objekten

Der Gleichheitsoperator (`==`) kann nicht nur auf `String`, sondern auf beliebigen Objekten ausgeführt werden. Aber wie bereits bei der Klasse `String` gesehen, gibt dieser Operator nur dann `true` zurück, wenn die beiden Objekte Aliase voneinander sind (wenn die Referenzvariablen also auf das gleiche Objekt zeigen):

```
Dice dice1 = new Dice();
Dice dice2 = dice1;
if (dice1 == dice2)
        System.out.println("Aliase!");
```

Der Operator `==` ist also nicht nützlich, um Objekte *wirklich* miteinander zu vergleichen.

Die Methode `equals` ist für alle Klassen (auch für selbst geschriebene Klassen) automatisch vorhanden und kann deshalb für den Vergleich von Objekten verwendet werden. Dass diese Methode tatsächlich zur Verfügung steht, verdanken wir dem Konzept der Vererbung, das wir in späteren Kapiteln noch genauer betrachten werden.

Allerdings verhält sich die geerbte Methode `equals` genau gleich wie der Gleichheitsoperator (==) und es wird lediglich überprüft, ob die beiden zu vergleichenden Objekte Aliase voneinander sind. Wenn man eine eigene Klasse programmiert, kann man die Methode `equals` aber nach eigenen Bedürfnissen umdefinieren (bekannt als

überschreiben einer geerbten Methode). Das heisst, es liegt im Ermessen der Programmiererin zu definieren, wann zwei Objekte als identisch angesehen werden (in der Klasse `String` zum Beispiel ist die Methode `equals` so umgeschrieben worden, dass die beiden Zeichenketten auf tatsächliche Gleichheit der Zeichen überprüft werden).

Beispiel 85 *Folgendes Code-Fragment demonstriert die Verwendung einer selbst geschriebenen Methode* `equals` *für die Klasse* `Dice`*:*

```
        /**
         * gibt true zurück wenn dieser Würfel, die gleiche
         * Punktezahl besitzt wie otherDice
         */
        public boolean equals(Dice otherDice) {
                if (this.points == otherDice.getPoints())
                        return true;
                else
                        return false;
        }
```

Folgendes Code-Fragment würde nun also `Gleiche Punktezahl!` *ausgeben.*

```
Dice dice1 = new Dice();
Dice dice2 = new Dice();
dice1.setPoints(4);
dice2.setPoints(4);
if (dice1.equals(dice2))
        System.out.println("Gleiche Punktezahl!");
```

Manchmal möchte man Objekte nicht nur auf Gleichheit überprüfen (mit der Methode `equals`), sondern herausfinden, ob ein gegebenes Objekt kleiner oder grösser ist als ein anderes Objekt. Auch hierzu kann man eine Methode programmieren, die jeweils auf Basis einer oder mehreren Variablen einen Vergleich durchführt.

Beispiel 86 *Eine mögliche Methode für die Klasse* `Dice` *ist die folgende Methode* `compareTo` *und nimmt (ähnlich wie die Methode* `equals`*) ein Objekt vom gleichen Typ entgegen (in diesem Fall also ein Objekt* `otherDice` *vom Typ* `Dice`*).*

```
        /**
         * gibt einen Vergleichwert zum Vergleich dieses
         * Würfels mit otherDice zurück
         */
        public int compareTo(Dice otherDice) {
                return this.points - otherDice.getPoints();
        }
```

Grundsätzlich gibt es bei Vergleichen zweier Objekte immer drei mögliche Resultate: ein Objekt ist grösser als, kleiner als oder gleich gross wie ein zweites Objekt. Diese drei Möglichkeiten werden in der Methode `compareTo` mit einer Rückgabe einer ganzen Zahl `i` umgesetzt. Für die zurückgegebene Zahl `i` gilt:

- `i < 0`, wenn das aufrufende Objekt kleiner ist als das Objekt, das als Parameter übergeben wurde.
- `i > 0`, wenn das aufrufende Objekt grösser ist als das Objekt, das als Parameter übergeben wurde.
- `i = 0`, wenn das aufrufende Objekt gleich gross ist wie das Objekt, das als Parameter übergeben wurde.

Die Methode `compareTo` könnte also beispielsweise folgendermassen eingesetzt werden:

```
if (dice1.compareTo(dice2) > 0)
        System.out.println("Würfel 1 gewinnt!");
```

Es ist kein Zufall, dass die eigens geschriebene Methode `compareTo` mit der gleichen Logik vorgeht, wie sie für die Klasse `String` programmiert ist (mit einer ganzzahligen Rückgabe, welche angibt, ob das eine Objekt kleiner, grösser oder gleich dem zweiten Objekt ist). Für bestimmte Abläufe oder Verhalten machen Programmiererinnen gewisse Regeln ab, an die sich alle halten sollten. Dies geschieht mit Hilfe sogenannter *Schnittstellen* – wir kommen in späteren Kapiteln nochmals auf die Methode `compareTo` und das Konzept der Schnittstelle `Comparable` zu sprechen.

Kapitel 7

Arrays

Ein *Feld* bzw. *Array*[1] ist ein einfaches, aber zugleich mächtiges Konzept, das von praktisch allen Programmiersprachen unterstützt wird. Arrays ermöglichen einem die einfache Verwaltung und Organisation von mehreren Variablen.

Im Gegensatz zur Klasse `ArrayList` gehören Arrays zur Programmiersprache Java – entsprechen also nicht einer programmierten Klasse aus dem Java API – wir kommen im letzten Abschnitt dieses Kapitels auf die Unterschiede zwischen einem Array und der Klasse `ArrayList` zu sprechen.

Wenn wir ein Programm schreiben, das eine Menge von Daten verwalten soll (z.B. neun `int` Werte), wäre es nicht sonderlich praktisch, für jeden Wert eine eigene Variable deklarieren zu müssen:

```
int num1, num2, num3, num4, num5, num6, num7, num8, num9;
```

Arrays lösen dieses Problem, indem diese die Deklaration einer einzigen Variablen erlauben, die dann mehrere individuell ansprechbare Variablen eines Typs enthalten kann. Jede Variable wird an einer nummerierten Position im Array gespeichert. Ein Array ist also eine Art Behälter für Variablen eines bestimmten Typs.

7.1 Arrays Deklarieren und Verwenden

In Java sind Arrays Objekte. Das heisst, wir müssen zunächst eine Variable für ein Array deklarieren. Das Array kann danach mit dem `new`-Operator instanziiert

[1] Wir verwenden in diesem Skript den englischen Ausdruck.

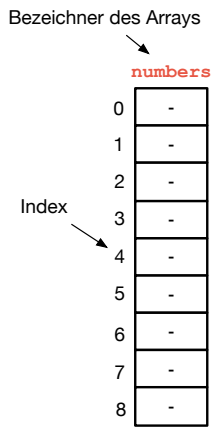

Abbildung 7.1: Ein Array mit Bezeichner `numbers`, das maximal `9` `int` Werte enthalten kann.

werden. Folgende Anweisung deklariert und instanziiert zum Beispiel das Array aus Abb. 7.1[2]:

```
int[] numbers = new int[9];
```

Die Variable `numbers` wird als Array für ganze Zahlen deklariert. Der Typ von `numbers` ist also `int[]` (gesprochen: *int-Array*). Ein Array kann so deklariert werden, dass es primitive Datentypen (z.B. `int` oder `double`) oder Objekte einer bestimmten Klasse (z.B. `String` oder `Dice`) aufnehmen kann. Alle Werte in einem Array müssen aber vom gleichen Typ sein. Das heisst, wir können kein Array deklarieren, das zum Beispiel sowohl Zahlen als auch Zeichenketten aufnehmen kann.

Die Deklaration der Variablen `numbers` vom Typ `int[]` enthält noch keine Angabe über die Grösse des Arrays:

```
int[] numbers;
```

Mit der Instanziierung der Variablen `numbers` durch den `new`-Operator

```
numbers = new int[9];
```

muss die Grösse des Arrays (in diesem Beispiel `9`) zwingend angegeben werden. Der

[2]Diese Visualisierung ist eigentlich nicht ganz korrekt: Ein `int[]` initialisiert alle Variablen im Array immer mit dem Wert `0`. Wir kommen später auf diese automatische Initialisierung zurück.

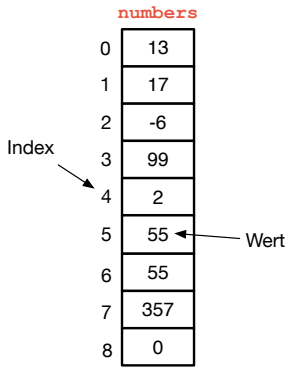

Abbildung 7.2: Ein Array mit Bezeichner `numbers`, das `int` Werte enthält.

Konstruktor wird nun für neun ganze Zahlen Platz reservieren – die Plätze im Array sind indexiert von `0` bis `8` (in Java beginnen Indizes immer mit `0`). Wurde ein Array einer gewissen Grösse instanziiert, kann die Grösse des Arrays nachträglich nicht mehr geändert werden.

Beispiel 87 *In Abb. 7.2 ist nochmals das Array* `numbers` *visualisiert (das Array enthält nun neun* `int` *Werte).*

Um auf einen bestimmten Wert in einem Array zuzugreifen, benötigen wir den Bezeichner des Arrays gefolgt von einem Index innerhalb eckiger Klammern. Um z.B. auf den fünften Wert des Arrays `numbers` zuzugreifen, schreiben wir

```
numbers[4]
```

Der Ausdruck `numbers[4]` verweist auf einen einzelnen `int` Wert, der im Array gespeichert ist. Der Ausdruck `numbers[4]` kann deshalb immer dann verwendet werden, wenn eine `int` Variable oder ein `int` Wert benötigt wird.

Beispiel 88 *Dem Ausdruck* `numbers[4]` *kann also zum Beispiel eine ganze Zahl zugewiesen werden:*

```
numbers[4] = 13;
```

Oder man kann einzelne Array Elemente für Berechnungen verwenden:

```
average = (numbers[0] + numbers[1] + numbers[2]) / 3.0;
```

Oder man kann `numbers[4]` *in* `println` *Anweisungen verwenden:*

```
System.out.println("Wert mit Index 4: " + numbers[4]);
```

Der in eckigen Klammern definierte Index darf auch das Resultat einer Berechnung oder eine Variable/Konstante sein:

```
pick = numbers[rand.nextInt(9)];
last = numbers[MAX];
```

Beispiel 89 *In folgendem Programm* `SampleArray` *wird ein Array mit dem Bezeichner* `list` *instanziiert, das* `10` *ganze Zahlen speichern kann. Dem Array werden danach die Zahlen* `10, 20, ..., 90, 100` *zugewiesen (mit Hilfe einer* `for`*-Schleife). Der Wert des sechsten Elementes (mit Index* `5`*) wird auf* `999999` *geändert (→ Video K-7.1).*

```
public class SampleArray {

        /**
         * Instanziert ein int Array und füllt dieses mit Werten,
         * modifiziert einen Wert und gibt den Inhalt des Arrays aus
         */
        public static void main(String[] args){
                final int LIMIT = 10, MULTIPLE = 10;

                // deklarieren und instanziieren des Arrays
                int[] list = new int[LIMIT];

                //  zuweisen von Werten im Array
                for (int i = 0; i < LIMIT; i++)
                        list[i] = (i + 1) * MULTIPLE;

                list[5] = 999999;  // ändern eines Wertes im Array

                // ausgeben aller Werte des Arrays
                for (int value : list)
                        System.out.print(value + "  ");
        }
}
```

Zur Ausgabe der Array Elemente wird die vereinfachte Syntax für `for`-Schleifen verwendet (die wir bereits bei der Klasse `ArrayList` ähnlich verwendet haben). Die Ausgabe des Programmes lautet:

```
10  20  30  40  50  999999  70  80  90  100
```

7.1.1 Grenzen Finden

Die Konstante `LIMIT` wird im obigen Programm `SampleArray` an zwei Stellen verwendet. Erstens bei der Instanziierung, um die Grösse des Arrays festzulegen und zweitens in der `for`-Schleife, um die Grenze der Laufvariablen `i` zu kontrollieren.

Wird mit dem Operator `[i]` eine Referenz auf ein Element im Array gemacht, so muss der Index `i` grösser-gleich `0` und kleiner als die Grösse des Arrays sein. Wurde beispielsweise ein Array `prices` der Grösse `100` instanziiert, so sind die gültigen Indizes `0` bis `99`. Immer wenn eine Referenz auf ein Element mit einem nicht-gültigen Index gemacht wird, wird ein Laufzeitfehler ausgelöst und das Programm wird nicht ordnungsgemäss terminieren.

In unseren Programmen sollten wir immer sicherstellen, dass wir nur mit gültigen Indizes auf das Array zugreifen. Hierzu bieten uns Arrays die mit Sichtbarkeit `public` deklarierte Konstante `length` an, die jedes Array nach deren Instanziierung besitzt und die tatsächliche Grösse des Arrays speichert (also die maximale Anzahl Plätze im Array)[3].

Da die Konstante `length` mit Sichtbarkeit `public` deklariert ist, kann diese direkt referenziert werden. Z.B. können wir die Grösse des Arrays `prices` durch `prices.length` auslesen (der dort gespeicherte Wert beträgt dann beispielsweise `100`). Der grösste gültige Index im Array `prices` wäre also `prices.length - 1`.

Beispiel 90 *In folgendem Programm wird ein `double[]` mit dem Bezeichner `numbers` der Grösse `5` instanziiert und mit Benutzereingaben gefüllt. Danach werden alle Werte des Arrays in umgekehrter Reihenfolge ausgegeben. Eine mögliche Ein- und Ausgabe lautet wie folgt:*

```
Geben Sie 5 Zahlen ein!
Zahl 1 eingeben: 45
Zahl 2 eingeben: 99.99
Zahl 3 eingeben: -17.1
Zahl 4 eingeben: 0
Zahl 5 eingeben: 8
Ihre Eingabe rückwärts:
8.0 0.0 -17.1 99.99 45.0
```

[3] Der Wert von `length` ist eine Konstante, da Arrays eine fixe Grösse besitzen, die nicht geändert werden kann.

Beachten Sie insbesondere die Verwendung der Konstanten **`length`** *in verschiedenen Anweisungen (→ Video K-7.2).*

```
import java.util.Scanner;

public class Reverse {

        /**
         * Liest vom Benutzer eine Reihe von Zahlen ein, speichert diese
         * in einem Array und gibt sie dann rückwärts wieder aus
         */
        public static void main(String[] args) {
                Scanner scan = new Scanner(System.in);

                double[] numbers = new double[5];
                System.out.println("Geben Sie " + numbers.length + " Zahlen ein!");

                for (int index = 0; index < numbers.length; index++) {
                        System.out.print("Zahl " + (index + 1) + " eingeben: ");
                        numbers[index] = scan.nextDouble();
                }

                System.out.println("Ihre Eingabe rückwärts:");

                for (int index = numbers.length - 1; index >= 0; index--)
                        System.out.print(numbers[index] + "  ");
        }
}
```

Man kann sogenannte *Initialisierungslisten* zur Instanziierung von Arrays verwenden. Die Elemente in einer solchen Liste werden durch Kommas separiert und befinden sich innerhalb geschweifter Klammern. Werden solche Initialisierungslisten verwendet, muss der `new`-Operator nicht verwendet werden (die Grösse des Arrays muss auch nicht angegeben werden, da diese durch die Anzahl Elemente der Liste vorgegeben ist).

Beispielsweise instanziieren wir mit folgender Anweisung ein Array `scores` bestehend aus `10` ganzen Zahlen:

```
int[] scores = {5, 3, 22, 31, 5, 18, 12, 55, 17, 17};
```

Eine Initialisierungsliste kann nur dann verwendet werden, wenn das Array zuerst deklariert worden ist. Zudem darf die Initialisierungsliste nur Elemente enthalten, die dem deklarierten Typen des Arrays entsprechen.

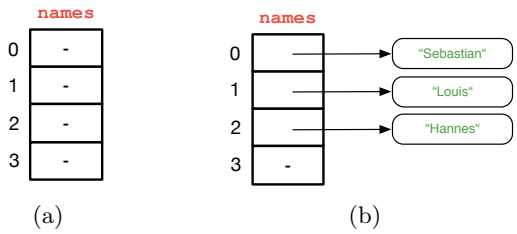

Abbildung 7.3: (a) ein "leeres" `String` Array. (b) ein `String` Array, das drei Referenzen auf `String` Objekte enthält.

7.2 Arrays für Objekte

In den vorangegangenen Beispielen haben wir Arrays verwendet, um primitive Datentypen (z.B. `int` oder `double` Werte) zu verwalten. Arrays können aber auch Objekte beinhalten. Wenn wir Objekte in einem Array speichern, so ist jedes Element des Arrays ein separates Objekt. Beachten Sie z.B. folgende Deklaration und Instanziierung:

```
String[] names = new String[4];
```

Die Variable `names` ist ein Array von Referenzen auf `String` Objekte. Der `new`-Operator instanziiert zwar das Array und reserviert Platz für vier Objekte vom Typ `String`, aber es werden *keine* Objekte vom Typ `String` instanziiert – es wird nur das Array instanziiert, das maximal vier `String` Objekte aufnehmen kann.

Nach der Instanziierung sieht das Array also wie in Abb. 7.3 (a) aus. Erst wenn einige `String` Objekte instanziiert und dem Array `names` hinzugefügt werden – zum Beispiel mit

```
String name1 = new String("Sebastian");
names[0] = name1;
names[1] = new String("Louis");
names[2] = "Hannes";
```

sieht das Array wie in Abb. 7.3 (b) aus. Beachten Sie insbesondere, dass in `names[3]` immer noch nichts referenziert wird (etwas genauer: `names[3]` zeigt auf das leere Objekt `null`). Jedes Objekt, das in einem Array enthalten ist, muss also separat instanziiert werden.

Etwas anders sieht es bei Arrays für primitive Datentypen aus. Bei der Instanziierung eines `int[]` zum Beispiel wird das ganze Array mit `0` gefüllt – bei einem `boolean[]` sind alle Einträge standardmässig `false`. Es ist aber guter Programmierstil, auch bei diesen Arrays jedem Element explizit einen Wert zuzuweisen.

Arrays für Objektvariablen können ebenfalls mit Initialisierungslisten instanziiert werden:

```
String[] names = {"Sebastian", "Louis", "Hannes"};
```

Dieses Array entspricht nicht ganz dem Array aus dem obigen Beispiel – was ist der Unterschied?

Beispiel 91 *Ein* `Team` *Objekt repräsentiert einen Sportclub und beinhaltet den Namen des Teams und die aktuelle Punktezahl des Teams als Instanzvariablen. Der Name des Teams* `teamName` *wird als Parameter über den Konstruktor initialisiert, während die Punktezahl* `currentPoints` *standardmässig mit* `0` *initialisiert wird (und mit der Methode* `addPoints` *verändert werden kann).*

```
public class Team {

        private String teamName;
        private int currentPoints;

        /**
         * instanziiert ein neues Team
         */
        public Team(String teamName) {
                this.teamName = teamName;
                this.currentPoints = 0;
        }

        /**
         * Zeichenkettenrepräsentation dieses Teams
         */
        public String toString() {
                return this.teamName + ": " + this.currentPoints + " Punkte";
        }

        /**
         * fügt diesem Team Punkte hinzu
         */
        public void addPoints(int points) {
                this.currentPoints += points;
        }
}
```

Betrachten Sie nun das Programm **`TeamArray`**. *Wir instanziieren das Array* **`teams`** *für* **`Team`** *Objekte mit einer Initialisierungsliste. Die* **`Team`** *Objekte werden direkt in der Initialisierungsliste mit dem* **`new`**-*Operator instanziiert (→Video K-7.3).*

```
public class TeamArray {

        /**
         * Demonstration eines Objekt-Arrays
         */
        public static void main(String[] args) {

                Team[] teams = {
                                new Team("FC Foo"), new Team("FC Bar"),
                                new Team("AC Qoo"), new Team("SC Tar")
                };
                // hinzufügen von Punkten
                teams[0].addPoints(3);
                teams[1].addPoints(5);
                teams[2].addPoints(7);
                teams[3].addPoints(1);

                for (Team team : teams)
                        System.out.println(team);
        }
}
```

Danach rufen wir für jedes Objekt im Array **`teams`** *die Methode* **`addPoints`** *auf und aktualisieren so für jedes Objekt den Punktestand. Beachten Sie insbesondere, wie wir* **`addPoints`** *auf einzelnen Objekten des Typs* **`Team`** *im Array* **`teams`** *aufrufen, z.B. mit*

```
teams[0].addPoints(3);
```

Mit einer **`for`**-**`each`** *Schleife geben wir den Inhalt des Arrays* **`teams`** *aus:*

```
FC Foo: 3 Punkte
FC Bar: 5 Punkte
AC Qoo: 7 Punkte
SC Tar: 1 Punkte
```

Im folgenden Beispiel betrachten wir ein Programm, in dem ein Objekt ein Array aus anderen Objekten speichert (→Video K-7.4).

Beispiel 92 *Wir verwenden für dieses Beispiel die folgende (etwas vereinfachte) Klasse* **`PlayerCard`**. *Die Klasse beinhaltet zwei Instanzvariablen, einen Konstruktor und eine Methode* **`toString`**.

```java
public class PlayerCard {

        // Instanzvariablen für PlayerCard Objekte
        private String playerName;
        private int cardNumber;

        /**
         * Konstruktor für PlayerCard Objekte
         */
        public PlayerCard(String playerName, int cardNumber) {
                this.playerName = playerName;
                this.cardNumber = cardNumber;
        }

        /**
         * Zeichenkettenrepräsentation für PlayerCard Objekte
         */
        public String toString() {
                return this.playerName + ", " + this.cardNumber;
        }
}
```

Die Klasse `PlayerCardCollection` *beinhaltet ein Array* `collection`*, das Objekte vom Typ* `PlayerCard` *aufnehmen kann. Die Klasse* `PlayerCardCollection` *beinhaltet zudem einen Zähler* `count` *für die Anzahl vorhandener Spielerkarten im Array* `collection` *und die aktuelle Grösse* `capacity` *des Arrays. Das Array* `collection` *wird im Konstruktor* `PlayerCardCollection` *instanziiert (mit initialer Grösse* `100`*). Da sich noch keine Karten in der Sammlung befinden, ist zu diesem Zeitpunkt* `count = 0`*.*

Wird eine Spielerkarte zur Sammlung hinzugefügt (via Methode `addPlayerCard`*), wird ein neues* `PlayerCard` *Objekt instanziiert und die zugehörige Referenz im Array* `collection` *gespeichert (und zwar an der nächsten freien Position im Array – dies ist die Position mit Index* `count`*). Danach muss der Zähler* `count` *um eins erhöht werden.*

Beim Hinzufügen überprüfen wir zuerst, ob wir die maximale Kapazität des Arrays erreicht haben[4]*. Ist die aktuelle Kapazität der Sammlung erreicht, wird die mit Sichtbarkeit* `private` *deklarierte Support-Methode* `increaseSize` *aufgerufen. Diese instanziiert zuerst ein neues Array* `temp`*, das doppelt so gross ist wie die aktuelle* `collection` *und danach wird jedes* `PlayerCard` *Objekt der aktuellen Sammlung einzeln in dieses neue Array kopiert. Letztendlich wird der Instanzvariablen* `collection` *dieses neue Array zugewiesen.*

[4]Sonst laufen wir Gefahr, zur Laufzeit einen Fehler zu erhalten.

```java
public class PlayerCardCollection {

	// Array aus PlayerCard Objekten
	private PlayerCard[] collection;
	// Anzahl PlayerCard Objekte und maximale Kapazität des Arrays
	private int count, capacity = 100;

	/**
	 * Instanziiert eine leere Sammlung
	 */
	public PlayerCardCollection() {
		this.collection = new PlayerCard[this.capacity];
		this.count = 0;
	}

	/**
	 * Fügt der Sammlung eine PlayerCard hinzu
	 * (vergrössert wenn nötig vorher das Array collection)
	 */
	public void addPlayerCard(String playerName, int cardNumber) {
		if (this.count == this.capacity)
			this.increaseSize();

		this.collection[count] = new PlayerCard(playerName, cardNumber);
		this.count++;
	}

	/**
	 * Gibt eine Zeichenkettenrepräsentation der Sammlung zurück
	 */
	public String toString() {
		String collectionAsString = "Anzahl Karten: " + this.count + "\n";
		collectionAsString += "\nMeine Karten:\n";
		for (int i = 0; i < this.count; i++)
			collectionAsString += this.collection[i] + "\n";

		return collectionAsString;
	}

	/**
	 * Vergössert die Kapazität der Sammlung indem ein Array mit doppelter Grösse
	 * instanziiert wird und die bestehende Sammlung in dieses Array kopiert wird
	 */
	private void increaseSize()   {
		PlayerCard[] temp = new PlayerCard[this.capacity * 2];
		for (int i = 0; i < this.collection.length; i++)
			temp[i] = this.collection[i];

		this.collection = temp;
		this.capacity = this.collection.length;
	}
}
```

Mit dieser Strategie können wir so viele Spielerkarten speichern, wie wir wollen. Der Benutzer der Klasse `PlayerCardCollection` *braucht sich keine Sorgen um die*

Grösse des Arrays zu machen, da die Klasse die Verantwortung hierfür übernimmt.

Die Methode **`toString`** *der Klasse* **`PlayerCardCollection`** *gibt eine Zusammenfassung über die Sammlung in Zeichenkettenform zurück – zudem werden die einzelnen Spielerkarten der Sammlung zurückgegeben (unter Benutzung der Methode* **`toString`** *der Klasse* **`PlayerCard`***).*

Betrachten Sie als Nächstes die Klasse **`PlayerCardApp`***, die eine* **`main`** *Methode enthält, in der eine Sammlung von Spielerkarten* **`myCards`** *instanziiert wird. Danach kann der Benutzer entweder Spielerkarten hinzufügen (mit der Methode* **`addPlayerCard`***), die Sammlung ausgegeben (mit der Methode* **`toString`***) oder das Programm beenden.*

```
import java.util.Scanner;

public class PlayerCardApp {

        /**
         * Ermöglicht die Verwaltung von Spielerkarten
         */
        public static void main(String[] args) {

                // Sammlung der Karten
                PlayerCardCollection myCards = new PlayerCardCollection();

                Scanner scan = new Scanner(System.in);
                int decision;

                do {
                        System.out.println("Was möchten Sie tun? 1=Karte Hinzufügen; "
                                        + "2=Karten ausgeben; 0=EXIT");
                         decision = Integer.parseInt(scan.nextLine());
                         switch(decision) {
                         case 1 : // Hinzufügen einer neuen Karte
                                 System.out.print("Spielername: ");
                                 String playerName = scan.nextLine();
                                 System.out.print("Kartennummer: ");
                                 int cardNumber = Integer.parseInt(scan.nextLine());
                                 myCards.addPlayerCard(playerName, cardNumber);
                                 break;
                         case 2 : // Ausgeben der Sammlung
                                 System.out.println(myCards);
                                 break;
                         case 0 :
                                 System.out.println("Programm wird beendet");
                                 break;
                         default : System.out.println("Ungültige Eingabe");
                         }
                } while (decision != 0);
        }
}
```

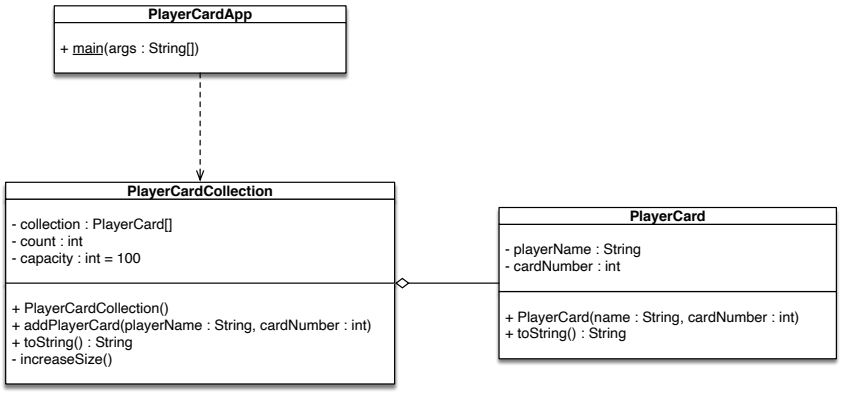

Abbildung 7.4: Die Klasse `PlayerCardApp` *verwendet* die Klasse `PlayerCardCollection`, welche eine *Aggregation* aus `PlayerCard` Objekten ist.

Eine mögliche Ein- und Ausgabe des Programmes lautet:

```
Was möchten Sie tun? 1=Karte Hinzufügen; 2=Karten ausgeben; 0=EXIT
1
Spielername: Karl
Kartennummer: 77
Was möchten Sie tun? 1=Karte Hinzufügen; 2=Karten ausgeben; 0=EXIT
1
Spielername: Ferdinand
Kartennummer: 17
Was möchten Sie tun? 1=Karte Hinzufügen; 2=Karten ausgeben; 0=EXIT
2
Anzahl Karten: 2

Meine Karten:
Karl, 77
Ferdinand, 17

Was möchten Sie tun? 1=Karte Hinzufügen; 2=Karten ausgeben; 0=EXIT
0
Programm wird beendet
```

Die Klasse `PlayerCardApp` verwendet die Klasse `PlayerCardCollection` und diese besteht z.T. aus `PlayerCard` Objekten. Eine Beziehung "besteht aus" nennt man *Aggregation* und wird in UML mit einem Pfeil mit leerer Raute dargestellt. In Abb. 7.4 ist das UML Klassendiagramm für das oben definierte Programm gezeigt.

7.3 Programmparameter

Der formale Parameter `args` der Methode `main` in Java Programmen ist immer ein Array aus `String` Objekten (`String[] args`). Bisher haben wir den Parameter `args` ignoriert – jetzt wollen wir diesen verwenden.

Wird ein Java Programm gestartet, wird immer als Erstes die `main` Methode aufgerufen. Der `String[]` Parameter, der üblicherweise den Bezeichner `args` erhält, repräsentiert sogenannte *Programmparameter*, die dem Programm beim Start übergeben werden können. Diese Technik ermöglicht es also, beim Start einer Anwendung gewisse Vorgaben in ein Programm einzulesen (z.B. die aktuell gültige Konfiguration oder Ähnliches).

Beispiel 93 *Das Programm* `Welcome` *benutzt Programmparameter, um eine spezifische Ausgabe zu machen (→Video K-7.5).*

```
public class Welcome {

        /**
         * Gibt eine länderspezifische Begrüssung aus
         */
        public static void main(String[] args) {

                String language = args[0];

                switch (language) {
                case "German" :
                        System.out.println("Hallo zusammen!");
                        break;
                case "English" :
                        System.out.println("Hi everyone!");
                        break;
                case "French" :
                        System.out.println("Bonjour ensemble!");
                        break;
                default :
                        System.out.println("Keine gültige Sprachauswahl");
                }
        }
}
```

Werden beim Starten dieses Programmes Programmparameter mitgegeben, so werden diese automatisch im Array `args` *gespeichert. In Abb. 7.5 wird die Klasse* `Welcome` *kompiliert und viermal mit unterschiedlichen Argumenten ausgeführt*[5].

[5] In *Eclipse* können Programmparameter unter `Run` > `Run Configurations...` im Reiter `Arguments` angegeben werden.

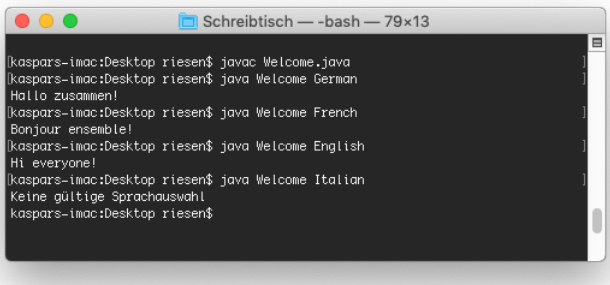

Abbildung 7.5: Die Klasse `Welcome` wird kompiliert und dreimal mit unterschiedlichen Programmparametern ausgeführt.

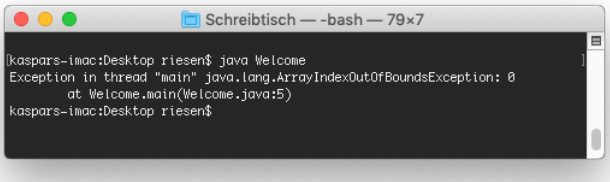

Abbildung 7.6: Der Versuch auf `args[0]` zuzugreifen scheitert.

Wird keine Zeichenkette beim Starten des Programmes `Welcome` übergeben, würde das Array `args` nicht genügend (bzw. keine) Elemente enthalten, was zu einem Laufzeitfehler (mit dem Namen `ArrayIndexOutOfBoundsException`) führt (siehe Abb. 7.6).

Der Parameter der `main` Methode ist immer ein Array aus `String` Objekten. Möchte man einem Programm beim Aufruf numerische Parameter mitgeben, müssen diese in der Methode `main` konvertiert werden.

Beispiel 94 *Folgendes Programm erwartet zwei ganze Zahlen als Programmparameter, addiert diese und gibt das Resultat der Berechnung aus (→ Video K-7.6):*

```
public class Addition {

        /**
         * Addiert zwei ganze Zahlen
         */
        public static void main(String[] args) {
                int op1 = Integer.parseInt(args[0]);
                int op2 = Integer.parseInt(args[1]);
                System.out.println(op1 + " + " + op2 + " = " + (op1 + op2));
        }
}
```

7.4 Variable Länge von Parameterlisten

Angenommen wir wollen eine Methode programmieren, der man drei Zahlen als Parameter übergeben kann und die dann den Mittelwert dieser Zahlen berechnet und zurückgibt. Die erste Methode `computeAvg` in der Klasse `Util` erfüllt diese Aufgabe (→Video K-7.7):

```
public class Util {

        /**
         * Berechnet den Mittelwert dreier Zahlen und gibt diesen zurück
         */
        public double computeAvg(int i1, int i2, int i3){
                double sum = i1 + i2 + i3;
                return sum / 3.0;
        }

        /**
         * Berechnet den Mittelwert beliebig vieler Zahlen und gibt diesen zurück
         */
        public double computeAvg(int ... values){
                if (values.length == 0)
                        return 0;
                else {
                        double sum = 0.;
                        for (int value : values)
                                sum += value;
                        return sum / values.length;
                }
        }
}
```

Folgender Aufruf wäre nun also möglich:

```
Util util = new Util();
double avg = util.computeAvg(42, 69, 37);
```

Wollen wir aber eine Methode programmieren, der man *beliebig* viele Zahlen als Parameter übergeben kann und die dann den Mittelwert dieser Zahlen berechnet, benötigen wir ein anderes, flexibleres Konzept.

Java erlaubt die Definition von Parameterlisten mit variabler Länge. Betrachten Sie die zweite Methode `computeAvg`. Die drei Punkte `...` nach der Angabe eines Datentyps, also zum Beispiel

```
(int ... values)
```

geben an, dass die Methode eine variable Anzahl Parameter des vorangestellten Typs akzeptieren kann (in diesem Fall Werte vom Typ `int`). Die an die Methode übergebenen Werte werden dann zur Laufzeit automatisch in einem Array mit Bezeichner `values` gespeichert. Auf dieses Array kann man dann innerhalb der Methode zugreifen.

Wir können nun der Methode `computeAvg` eine beliebige Anzahl `int` Werte übergeben (auch Null – deshalb überprüfen wir zu Beginn der Methode, ob die Länge des Arrays `values` grösser als Null ist). Folgende Aufrufe sind jetzt also beispielsweise möglich:

```
util.computeAvg(42, 69, 37);
util.computeAvg(2, 4, 6, 8, 10, 12, 14);
util.computeAvg();
```

Diese Syntax kann für jeden Datentypen verwendet werden. Die folgende Methode akzeptiert z.B. beliebig viele Objekte vom Typ `PlayerCard` und gibt diese danach der Reihe nach aus:

```
public void printPlayerCards(PlayerCard ... cards) {
        for (PlayerCard card : cards)
                System.out.println(card);
}
```

Beachten Sie, dass eine Methode, die eine variable Anzahl Parameter akzeptiert, zusätzlich auch noch "normale" Parameter verarbeiten kann (die variable Anzahl Parameter muss hierbei aber am Schluss der Parameterliste stehen und eine Methode kann nur *eine* Liste mit variablen Parametern entgegennehmen).

Beispielsweise akzeptiert die folgende Methode einen `int` Wert, ein `String` Objekt und danach beliebig viele `double` Werte (die zur Laufzeit in einem Array mit Bezeichner `numbers` gespeichert werden):

```
public void dummy(int count, String name, double ... numbers) { }
```

Konstruktoren können natürlich auch mit variabler Anzahl Parameter programmiert werden.

Beispiel 95 *Der Konstruktor der Klasse* **`Band`** *speichert die erste Zeichenkette in der Instanzvariablen* **`bandName`** *und die weiteren Parameter in einem Array* **`members`** *(→ Video K-7.8).*

```
public class Band {

        private String bandName;
        private String[] members;

        /**
         * Instanziiert eine Band mit den Namen der
         * Band und der Bandmitglieder
         */
        public Band(String bandName, String ... members) {
                this.bandName = bandName;
                this.members = members;
        }

        /**
         * Gibt eine Zeichenkettenrepräsentation der Band zurück
         */
        public String toString(){
                String band = "Band: " + this.bandName +"\n";
                for (String name : this.members)
                        band += name + "\n";
                return band;
        }
}
```

Das Programm **`BandTest`** *instanziiert zwei* **`Band`** *Objekte, indem unterschiedliche Anzahlen* **`String`** *Objekte an den Konstruktor* **`Band`** *übergeben werden.*

```
public class BandTest {

        /**
         * Instanziiert zwei Bands mit dem gleichen Konstruktor
         * mit unterschiedlichen Anzahlen Parameter
         */
        public static void main(String[] args) {

                Band band1 = new Band("One-Man-Show", "Kim");
                Band band2 = new Band("Greatest Band", "Ray", "Mac", "Sandy",
                                "Johnny", "Bonny");

                System.out.println(band1);
                System.out.println(band2);
        }
}
```

Die Ausgabe des Programmes **`BandTest`** *lautet:*

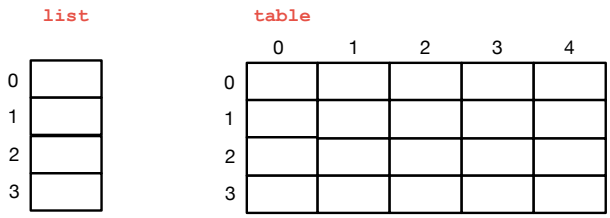

Abbildung 7.7: Ein- vs. zweidimensionale Arrays.

```
Band: One-Man-Show
Kim

Band: Greatest Band
Ray
Mac
Sandy
Johnny
Bonny
```

7.5 Zweidimensionale Arrays

Die Arrays, die wir bis jetzt betrachtet haben, waren alle *eindimensional*. *Zweidimensionale* Arrays besitzen Werte in zwei Dimensionen. Man kann sich zweidimensionale Arrays auch als Tabellen mit Zeilen und Spalten vorstellen (siehe Abb. 7.7).

Ein zweidimensionales Array wird deklariert und instanziiert, indem die Grössen beider Dimensionen separat definiert werden (beachten Sie, dass die Anzahl Zeilen und Spalten in zweidimensionalen Arrays nicht gleich gross sein müssen). Folgende Anweisung instanziiert ein Array für ganze Zahlen mit `4` Zeilen und `5` Spalten:

```
int[][] table = new int[4][5];
```

In zweidimensionalen Arrays benötigen wir zwei Indizes, um einen Wert anzusprechen (ein Index für die Zeile und ein zweiter Index für die Spalte). Folgende Anweisung weist der Variablen `value` den Wert zu, der im Array `table` in der dritten Zeile und der vierten Spalte steht:

```
int value = table[2][3];
```

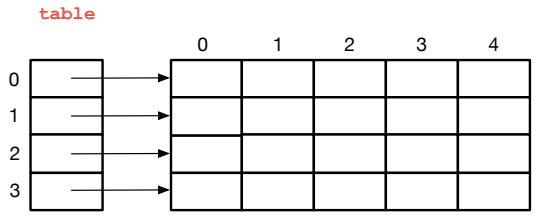

Abbildung 7.8: Ein zweidimensionales Array ist ein Array aus Arrays.

Die Visualisierung in Abb. 7.7 ist nicht ganz korrekt. Ein zweidimensionales Array ist nämlich ein Array aus Arrays. Das heisst, die einzelnen Elemente im Array sind wiederum Arrays (siehe Abb. 7.8). Deshalb referenziert die Variable `table.length` die Anzahl Zeilen des zweidimensionalen Arrays `table` und mit `table[0].length` referenzieren wir die Anzahl Spalten in Zeile `0` (oder genauer: Die Grösse des Arrays, das sich an Position `0` im Array `table` befindet).

Beispiel 96 *In der Klasse* `Table` *wird ein zweidimensionales Array* `table` *der Dimension* `10` $\times$ `10` *für ganze Zahlen instanziiert, mit Werten gefüllt und ausgegeben (→ Video K-7.9):*

```
public class Table {

        /**
         * Demonstration eines 2D Arrays
         */
        public static void main (String[] args){
                // 2D Array der Grösse 10x10
                int[][] table = new int[10][10];

                // Füllen der Tabelle
                // table.length = Anzahl "Zeilen"
                // table[row].length = Anzahl Elemente in "Zeile" mit Index row
                for (int row = 0; row < table.length; row++)
                        for (int col = 0; col < table[row].length; col++)
                                table[row][col] = row * 10 + col;

                // Ausgeben der Tabelle
                for (int row = 0; row < table.length; row++){
                        for (int col = 0; col < table[row].length; col++)
                                System.out.print(table[row][col] + "\t");
                        System.out.println();
                }
        }
}
```

In obigem Programm wird das Array `table` mit einer verschachtelten `for`-Schleife mit Werten gefüllt. An Position `table[i][j]` steht nach Abarbeitung dieser Schlei-

fen der Wert $10i + j$ (also z.B. `table[3][5] = 35`). Die äussere Schleife wird verwendet, um von Zeile zu Zeile zu springen und wird durch `table.length` (= Anzahl Zeilen) kontrolliert. Die inneren Schleifen springen von Spalte zu Spalte einer gegebenen Zeile. Die Anzahl Spalten in Zeile `row` kann aus `table[row].length` ausgelesen werden.

Wie bei eindimensionalen Arrays können auch bei zweidimensionalen Arrays Initialisierungslisten zur Instanziierung verwendet werden.

Beispiel 97 *Die Resultate von vier Sportmannschaften sind in der Initialisierungsliste für das Array* **`results`** *zusammengefasst. Jede Zeile in* **`results`** *enthält die Anzahl Siege (Spalte 0), die Anzahl Unentschieden (Spalte 1) und die Anzahl Niederlagen (Spalte 2) für jedes der vier Teams. Das heisst,* **`results`** *besitzt vier Zeilen und drei Spalten (→ Video K-7.10).*

```
public class LeagueTable {

        /**
         * Berechnet die Punkte für jedes Team basierend auf einer
         * Tabelle mit den Anzahl Siegen, Unentschieden und Niederlagen
         */
        public static void main(String[] args) {

                final int POINTS_FOR_WIN = 3, POINTS_FOR_TIE = 1;

                Team[] teams = {
                                new Team("FC Foo"), new Team("FC Bar"),
                                new Team("AC Qoo"), new Team("SC Tar")
                };

                // Anzahl Siege, Unentschieden und Niederlagen der vier Teams
                int[][] results = { {1, 2, 0},
                                    {2, 0, 1},
                                    {1, 1, 1},
                                    {1, 0, 2} };

                final int NUM_OF_TEAMS = results.length;
                int points;

                // Punkteberechnung
                for (int i = 0; i < NUM_OF_TEAMS; i++) {
                        points = results[i][0] * POINTS_FOR_WIN;
                        points += results[i][1] * POINTS_FOR_TIE;
                        teams[i].addPoints(points);
                }

                for (int i = 0; i < NUM_OF_TEAMS; i++)
                        System.out.println(teams[i]);
        }
}
```

Das Programm `LeagueTable` *berechnet innerhalb einer* `for`*-Schleife die Anzahl Punkte für jedes der vier Teams. Wir verwenden also nochmals die Klasse* `Team` *aus einem der obigen Beispiele. Beachten Sie den Einsatz der Konstanten: Siege ergeben in diesem Beispiel je drei und Unentschieden je einen Punkt. Für jedes Objekt im Array* `teams` *wird dann die Methode* `addPoints` *aufgerufen und die berechnete Punktezahl übergeben.*

Mit Hilfe einer weiteren `for`*-Schleife werden alle* `Team` *Objekte ausgegeben. Die Ausgabe dieses Programmes lautet:*

```
FC Foo: 5 Punkte
FC Bar: 6 Punkte
AC Qoo: 4 Punkte
SC Tar: 3 Punkte
```

7.6 Vergleich von Arrays und `ArrayList`

In vorherigen Kapiteln haben wir die Klasse `ArrayList` des Java APIs verwendet, um eine Sammlung von Daten zu verwalten. Tatsächlich verwendet die Klasse `ArrayList` zur internen Verwaltung der Daten ein Array, wie wir es in diesem Kapitel kennen gelernt haben.

Die wichtigsten Unterschiede der beiden Behälter sind in folgender Liste zusammengefasst:

- Eine `ArrayList` ist eine *mit Java geschriebene* Klasse, während ein Array *Teil der Sprache* Java ist.
- Eine `ArrayList` passt ihre Grösse automatisch den tatsächlich vorhandenen Elementen an. Mit Hilfe der Methode `size` kann die aktuelle Anzahl gespeicherter Elemente erfragt werden. Ein Array hingegen hat eine fixe Grösse, die bei der Instanziierung festgelegt werden muss. Diese maximale Kapazität kann über die Konstante `length` erfragt werden. Die Anzahl tatsächlich vorhandener Elemente im Array müssen wir dabei selber verwalten (z.B. mit einer Variablen `count`).
- Das Zugreifen auf oder das Hinzufügen von Elementen in einer `ArrayList` erfolgt über Methoden (z.B. `add` oder `get`). Bei Arrays erfolgt der Zugriff mit eckigen Klammern `[]` und einem Index.

- Bei einer `ArrayList` wird das hinzuzufügende Element automatisch ans Ende der Liste angehängt. In einem Array müssen wir den nächsten freien Platz selber verwalten.
- Eine `ArrayList` kann nur Objekte speichern, während ein Array sowohl primitive Datentypen als auch Objektvariablen speichern kann.
- Eine `ArrayList` verwaltet die Daten in einer eindimensionalen Liste, während ein Array auch mehrdimensional instanziiert werden kann.

Kapitel 8

Eigene Klassen Programmieren (Teil 2)

Die Erstellung von Software umfasst vier grundlegende Tätigkeiten, die selten bis nie streng linear ablaufen – sie überschneiden sich, beeinflussen einander gegenseitig und meistens ist der gesamte Prozess iterativ (das heisst, es braucht mehrere Durchgänge durch die vier Tätigkeiten – siehe Abb. 8.1):

- **Festlegen der Anforderungen**: Die Anforderungen geben an, welche Aufgaben ein Programm erfüllen muss. Wichtig ist dabei, dass definiert wird, *was* die Software tun kann, aber noch nicht *wie* dies die Software tun wird. Es ist eine relativ schwierige Aufgabe, detaillierte, eindeutige und vollständige Anforderungen an eine Software festzulegen.

- **Erstellen eines Entwurfs**: Ein Softwareentwurf legt fest, *wie* ein Programm seine Anforderungen erfüllen soll. Der Softwareentwurf legt insbesondere fest, wie die Lösung in handhabbare Teile zerlegt werden kann und was jeder Teil tun wird. In der objektorientierten Entwicklung bedeutet dies, dass wir festlegen

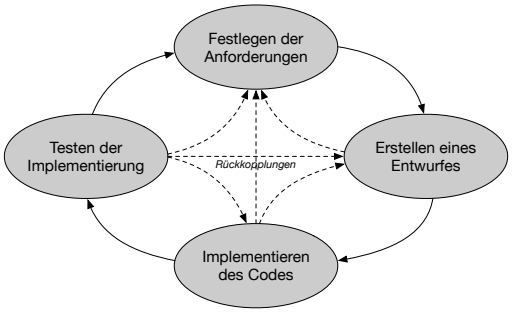

Abbildung 8.1: Vier grundlegende Tätigkeiten der Software-Entwicklung.

müssen, welche Klassen und Objekte benötigt werden und wie diese zusammenwirken sollen. Ferner muss entworfen werden, wie die einzelnen Methoden ihre Aufgaben erfüllen werden.

- **Implementieren des Codes**: Die Implementierung ist der Prozess der Umsetzung eines Entwurfs in Quellcode. Dieser Schritt ist eigentlich der am wenigsten kreative Schritt, da fast alle wichtigen Entscheidungen während der Anforderungs- und Entwurfsphase getroffen werden. Die Implementierung sollte sich auf die Codierungsdetails konzentrieren, einschliesslich Stilrichtlinien und Dokumentation.

- **Testen der Implementierung**: Beim Testen wird versucht sicherzustellen, dass das Programm das beabsichtigte Problem unter allen in den Anforderungen spezifizierten Einschränkungen korrekt und vollständig löst. Ein Programm muss gründlich getestet werden, um Fehler zu finden. Werden Fehler gefunden, muss die Ursache des Problems ermittelt und behoben werden.

Der Fokus dieses Kapitels liegt auf dem Entwurf von Klassen und zeigt einige Besonderheiten und Werkzeuge von Java hierzu.

8.1 Klassen und Objekte Identifizieren

Eine der fundamentalen Aufgaben der objektorientierten Programmierung ist das Bestimmen der Klassen, aus denen das gesamte Programm bestehen soll. Die Klassen bestimmen letztendlich die Objekte, die das komplette System ausmachen werden. Man muss deshalb sehr sorgfältig abwägen, wie die vielfältigen Elemente und Anforderungen an eine Problemstellung in einem Programm repräsentiert werden sollen.

Oftmals werden die Anforderungen an ein Programm in einem Dokument zusammengefasst – in der *funktionalen Spezifikation*. In folgendem Beispiel ist ein sehr kleiner Ausschnitt einer einfachen Spezifikation gezeigt:

Beispiel 98 *Ein **Benutzer** kann der **Anwendung** beliebig viele **Spielerkarten** hinzufügen. Eine **Spielerkarte** repräsentiert dabei einen **Fussballspieler**, der durch verschiedene **Attribute** beschrieben werden kann (z.B. durch dessen **Namen**, **Club**, **Nationalität** und **Position**). Die bereits eingegebenen **Spielerkarten** sollen in einer **Sammlung** gespeichert werden. Diese **Sammlung** kann durchsucht werden. Das heisst, der **Benutzer** kann einen beliebigen **Begriff** eingeben und es werden ihm alle **Spielerkarten** angezeigt, welche diesen **Begriff** in irgendeinem **Attribut** beinhalten. Ausserdem kann der **Benutzer** ...*

In obigem Beispiel sind alle Substantive markiert. Tatsächlich stehen Substantive in einer funktionalen Spezifikation oftmals für die zu erstellenden Objekte (bzw. Klassen). Natürlich werden nicht alle Substantive der Spezifikation zu einer eigenen Klasse führen. Die Identifikation der Substantive kann aber ein guter Startpunkt sein, um über die Typen der Objekte, mit denen das Programm konzipiert und organisiert werden soll, nachzudenken.

Ein in Mehrzahl formuliertes Substantiv in der Spezifikation (z.B. *Spielerkarten*) kann darauf hinweisen, dass eine Klasse nötig ist, die für das Konzept einer *Spielerkarte* stehen soll (z.B. mit einer Klasse `PlayerCard`). Klassen erhalten oftmals Namen in Einzahl. Eine Klasse repräsentiert ja nur das Konzept, aus dem wir dann so viele Instanzen – also Objekte – wie nötig instanziieren können. Auch wenn nur ein einziges Objekt eines bestimmten Typs in einem System nötig sein wird, kann dieses oftmals am besten als eigene Klasse programmiert werden (z.B. eine Klasse `PlayerCardCollection`, die genau *eine* Sammlung von Spielerkarten repräsentiert).

Es ist manchmal schwierig zu entscheiden, wann ein bestimmtes Konzept tatsächlich als eigene Klasse programmiert werden soll und wann ein primitiver Datentyp (oder ein `String`) ausreichend sein könnte. Zum Beispiel könnten wir initial davon ausgehen, dass die Adresse eines Mitarbeitenden als eine Reihe von Instanzvariablen erstellt werden könnte. Vielleicht merkt man aber in weiteren Diskussionen, dass die Adresse flexibler und reichhaltiger sein sollte und dass ein Mitarbeitender evtl. mehrere Adressen besitzt. Ist dies der Fall, sollten wir die Adresse vielleicht eher als eigene Klasse `Address` programmieren, die dann alle nötigen Variablen und Funktionalitäten selbständig verwaltet.

Zusätzlich zu den Klassen, welche die Objekte der Problemdomäne repräsentieren, brauchen wir wahrscheinlich auch Klassen, die die Arbeit mit den Objekten unterstützen. Also zum Beispiel die Interaktion zwischen den Objekten ermöglichen oder gewisse Hilfsaufgaben für die Objekte erledigen.

8.2 Enum-Typen

Ein sogenannter *Enum-Typ* `enum` definiert einen neuen Datentyp ähnlich wie eine Klasse. Ein `enum` gibt dabei aber alle möglichen Werte vor, die eine Variable dieses Typs annehmen darf, indem diese durch die Programmiererin im Quellcode aufgezählt werden. Die Werte dieser Aufzählung sind beliebige Bezeichner.

Beispiel 99 *Im* `enum Position` *werden drei Werte aufgezählt:*

```
public enum Position {

        Stürmer, Verteidiger, Torwart

}
```

Die aufgezählten Bezeichner repräsentieren automatisch instanziierte Objekte des entsprechenden Enum-Typen. Zum Beispiel sind `Stürmer` oder `Torwart` beides instanziierte Objekte der Klasse `Position`.

Nehmen wir nun an, dass wir eine Variable `pos` vom Typ `Position` deklarieren:

```
Position pos;
```

Die möglichen Werte (`Stürmer, Verteidiger, Torwart`) sind Referenzen auf `Position` Objekte, die als statische Variablen innerhalb der `Position` Klasse definiert sind (ähnlich wie zum Beispiel die statische Konstante `PI` in der Klasse `Math`). Deshalb können wir nun folgende Zuweisung machen:

```
pos = Position.Torwart;
```

Einem Objekt, dessen Typ einem `enum` entspricht, können nur Werte aus der entsprechenden Aufzählung zugewiesen werden. Man sagt, ein `enum` ist *typsicher* (engl. *type safe*). Ein Versuch, einer Variablen vom Typ `Position` einen nicht aufgelisteten Bezeichner zuzuweisen, wird mit einem Kompilierfehler enden.

Jedes `enum` Objekt besitzt automatisch die folgenden zwei Methoden:

- `String name()`: Gibt den Bezeichner des `enum` Objektes als Zeichenkette zurück (also zum Beispiel `"Torwart"`).
- `int ordinal()`: Gibt die Position des `enum` Objektes innerhalb der Aufzählung zurück. Der erste Bezeichner der Aufzählung erhält die Ordinalzahl `0`, der zweite Bezeichner besitzt die Ordinalzahl `1`, usw.

Nach der Zuweisung `pos = Position.Torwart;`

entspricht die Ausgabe von

```
System.out.println(pos.name() + " " + pos.ordinal());
```

zum Beispiel: `Torwart 2`

Jedes `enum` besitzt zudem zwei statische Methoden (die Sie direkt auf dem `enum` aufrufen können):

- `static E valueOf(String s)`: Nimmt eine Zeichenkette `s` entgegen und gibt das Objekt vom `enum` Typen `E` aus der Aufzählung zurück, dessen Bezeichner mit `s` übereinstimmt.
- `static E[] values()`: Gibt alle Objekte dieser Aufzählung in einem Array zurück.

In folgendem Programm sind die Methoden eines `enum` zusammengefasst (→Video K-8.1).

```
public class EnumDemo {

        /**
         * Illustriert den Gebrauch eines Enum
         */
        public static void main(String[] args) {

                Position pos;
                pos = Position.Torwart; // typsichere Zuweisung

                // die Methoden name und ordinal
                System.out.println(pos.name());
                System.out.println(pos.ordinal());

                // mit valueOf aus einer Zeichenkette das entsprechende Enum herleiten
                String input = "Stürmer";
                pos = Position.valueOf(input);

                System.out.println(pos.name());
                System.out.println(pos.ordinal());

                // mit values alle Objekte des Enum in ein Array auslesen
                Position[] positions = Position.values();
                for (Position p : positions)
                        System.out.println(p.ordinal() + ": " + p.name());
        }
}
```

Die Ausgabe des Programmes lautet:
`Torwart`

```
2
Stürmer
0
0: Stürmer
1: Verteidiger
2: Torwart
```

In folgendem Beispiel definieren wir ein weiteres `enum` – dieses Mal vom Typ `Club` (→Video K-8.2). Wir verwenden auch hier das reservierte Wort `enum` und wir zählen auch hier alle zulässigen Werte des Typs auf. Dieses Beispiel zeigt, dass man einem `enum` neben der Aufzählung der zulässigen Werte auch noch weitere, beliebige Variablen und Methoden hinzufügen kann. Zusätzlich definieren wir in diesem `enum` zum Beispiel zwei Instanzvariablen, einen Konstruktor und noch zwei weitere Methoden.

```java
public enum Club {

        FCBarcelona("FC Barcelona", 1899),
        BSCYB("BSC Young Boys", 1898),
        FCBayernMünchen("FC Bayern München", 1900);

        private String clubName;
        private int year;

        /**
         * Konstruktor für die drei aufgezählten Club Objekte
         */
        private Club(String clubName, int year) {
                this.clubName = clubName;
                this.year = year;
        }

        /**
         * Getter für den Clubnamen
         */
        public String getClubName() {
                return this.clubName;
        }

        /**
         * Getter für das Gründungsjahr
         */
        public int getYear() {
                return this.year;
        }
}
```

Jeder Bezeichner der Aufzählung in unserem `enum` ruft automatisch den Konstruktor

auf und übergibt diesem die in Klammern angegebenen Parameter. Die Parameter werden dann für jedes der Objekte in der Instanzvariablen `clubName` bzw. `year` gespeichert. Im Objekt `BSCYB` speichert also die Variable `clubName` nun die Zeichenkette `"BSC Young Boys"` und in `year` wird der Wert `1898` referenziert.

Der Konstruktor instanziiert automatisch für alle Bezeichner der Aufzählung je ein Objekt. Zusätzliche Objekte dieses Typs dürfen aber nicht instanziiert werden. Das heisst, der Konstruktor darf nicht auch sonst noch aufgerufen werden. Deshalb muss der Konstruktor eines `enum` mit der Sichtbarkeit `private` deklariert werden.

Mit den beiden `enum` `Position` und `Club` könnten wir nun unsere Beispielklasse `PlayerCard` erweitern (→Video K-8.2 (Fortsetzung)):

```
public class PlayerCard {

        // Instanzvariablen für PlayerCard Objekte
        private String playerName;
        private int cardNumber;
        private Position position;
        private Club club;

        /**
         * Konstruktor für PlayerCard Objekte
         */
        public PlayerCard(String playerName, int cardNumber,
                        Position position, Club club) {
                this.playerName = playerName;
                this.cardNumber = cardNumber;
                this.position = position;
                this.club = club;
        }

        /**
         * Zeichenkettenrepräsentation für PlayerCard Objekte
         */
        public String toString() {
                String cardAsString = "Spielername:\t" + this.playerName + "\n";
                cardAsString += "Clubname:\t" + this.club.getClubName() + "\n";
                cardAsString += "Position:\t" + this.position.name() + "\n";
                cardAsString += "KartenNr:\t" + this.cardNumber + "\n";
                return cardAsString;
        }
}
```

8.3 Klassen mit Statischen Variablen/Methoden

Wir haben statische Methoden bereits in einigen Fällen verwendet. Zum Beispiel sind alle Methoden der Klasse `Math` statisch. Eine statische Methode wird mit dem Klassennamen statt über den Bezeichner eines Objektes aufgerufen.

Beispiel 100 *Die Methode* `roll` *der Klasse* `Dice` *ist nicht statisch. Bevor wir diese Methode benutzen können, müssen wir ein Objekt vom Typ* `Dice` *instanziieren:*

```
Dice dice = new Dice();
dice.roll();
```

Die Methode `sqrt` *der Klasse* `Math` *hingegen ist eine statische Methode. Wir rufen diese mit dem Klassennamen auf:*

```
Math.sqrt(9);
```

8.3.1 Statische Variablen

Bis hierhin haben wir zwei Kategorien von Variablen programmiert: lokale Variablen, die innerhalb einer Methode deklariert sind (und somit nur innerhalb der entsprechenden Methode verwendet werden können) und Instanzvariablen, die innerhalb einer Klasse, aber nicht innerhalb einer Methode deklariert werden.

Man sagt "Instanzvariable", da jede Instanz der Klasse eine eigene Version der Variablen besitzt. Das heisst, jedes Objekt erhält separaten Speicherplatz für jede Variable, so dass jede Instanz unterschiedliche Werte für die Variablen speichern kann (z.B. besitzt jedes Objekt vom Typ `PlayerCard` einen eigenen Namen `playerName`).

Eine *statische Variable* (manchmal auch *Klassenvariable* genannt), wird von allen Instanzen der entsprechenden Klasse geteilt. Es existiert also nur *eine Kopie* einer statischen Variablen für *alle* Objekte dieser Klasse. Das heisst, wenn man den Wert einer statischen Variablen ändert, so ändert der Wert dieser Variable für *alle* Objekte der Klasse.

Das reservierte Wort zur Deklaration statischer Variablen ist `static`. Eine statische Variable vom Typ `int` (mit Bezeichner `globalCount`) wird beispielsweise folgendermassen deklariert (und auch gleich mit dem Wert `0` initialisiert):

```
public static int globalCount = 0;
```

Konstanten, also Variablen, die mit dem Modifikator `final` deklariert sind, werden recht häufig mit den Modifikatoren `public` und `static` deklariert. Konstanten werden typischerweise von verschiedenen Klassen benutzt und ein direkter Zugriff ohne Instanziierung ist deshalb oftmals sinnvoll. Ausserdem kann der Wert einer Konstanten nicht verändert werden – deshalb reicht es aus, nur eine einzige Kopie dieser Variablen für alle Objekte zu speichern.

Beispiel 101 *In folgender Klasse ist der Einsatz einer statischen Variablen demonstriert (→Video K-8.3). Jedes Mal wenn der Konstruktor* **`SimplePlayerCard`** *aufgerufen wird, wird die statische Variable* **`globalCount`** *um 1 erhöht. Der aktuelle Wert der statischen Variablen* **`globalCount`** *wird dann auf der nächsten Zeile der Instanzvariablen* **`cardNumber`** *zugewiesen. Das heisst, das erste Objekt vom Typ* **`SimplePlayerCard`**, *das instanziiert wird, speichert in der Variablen* **`cardNumber`** *den Wert 1, das zweite Objekt speichert in* **`cardNumber`** *den Wert 2, etc.*

```
public class SimplePlayerCard {

        // globaler Zähler für Karten
        public static int globalCount = 0;

        // Instanzvariablen für SimplePlayerCard Objekte
        private String playerName;
        private int cardNumber;

        /**
         * Konstruktor für SimplePlayerCard Objekte
         */
        public SimplePlayerCard(String playerName) {
                this.playerName = playerName;
                globalCount++;
                this.cardNumber = globalCount;
        }

        /**
         * Zeichenkettenrepräsentation für SimplePlayerCard Objekte
         */
        public String toString() {
                String cardAsString = "Spielername:\t" + this.playerName + "\n";
                cardAsString += "Karten-Nr:\t" + this.cardNumber + "\n";
                return cardAsString;
        }
}
```

Die Ausgabe des Programmes folgenden **`PlayerCardTest`**

```java
public class PlayerCardTest {

        /**
         * Demonstriert den Einsatz einer statischen Variablen
         */
        public static void main(String[] args) {

                ArrayList<SimplePlayerCard> cards = new ArrayList<SimplePlayerCard>();
                cards.add(new SimplePlayerCard("John"));
                cards.add(new SimplePlayerCard("Jim"));
                cards.add(new SimplePlayerCard("Mike"));

                System.out.println("Anzahl Karten: " + SimplePlayerCard.globalCount);

                System.out.println("Aktuelle Karten:");
                for (SimplePlayerCard pc : cards)
                        System.out.println(pc);
        }
}
```

ist demnach:

```
Anzahl Karten: 3
Aktuelle Karten:
Spielername: John
Karten-Nr: 1

Spielername: Jim
Karten-Nr: 2

Spielername: Mike
Karten-Nr: 3
```

Beachten Sie insbesondere, dass die Kartennummern jetzt automatisch initialisiert (und hochgezählt) werden und wie wir (mit dem Klassennamen) auf die statische Variable `globalCount` *zugreifen.*

8.3.2 Statische Methoden

Wenn wir in unserem Programm gewisse grundlegende Funktionalitäten benötigen, ist es oftmals sinnvoll, diese in einer separaten Klasse als *statische Methoden* (manchmal *Klassenmethoden* genannt) zu programmieren. Eine Methode wird mit dem Modifikator `static` als statisch deklariert. Um diese Funktionalitäten dann nutzen zu können, muss kein Objekt aus der Klasse instanziiert werden, da eine statische

Methode über den Klassennamen angesprochen werden kann.

Die Methoden der Klasse `Math` führen zum Beispiel grundlegende mathematische Berechnungen auf Werten durch, die als Parameter übergeben werden. Es gibt keinen Grund, hierfür extra ein Objekt vom Typ `Math` zu instanziieren, da wir lediglich die Funktionen (z.B. `sqrt`) nutzen wollen.

Wir haben bereits sehr oft eine statische Methode verwendet, nämlich `main`. Die Methode `main` eines Java Programmes wird beim Programmstart immer als Erstes aufgerufen. Die Methode `main` muss zwingend `static` deklariert werden, damit diese vom System aufgerufen werden kann, ohne die entsprechende Klasse vorher instanziieren zu müssen (sonst wäre ja diese Instanziierung der erste Aufruf, was ein Widerspruch wäre zur Aussage, dass `main` immer als Erstes aufgerufen wird).

Da statische Methoden nicht im Kontext eines Objektes operieren, können diese *nicht* auf Instanzvariablen zugreifen (diese existieren nur in der entsprechenden Instanz der Klasse). Der Kompilierer wird eine Fehlermeldung erzeugen, wenn Sie innerhalb einer statischen Methode versuchen, auf eine nicht-statische Variable zuzugreifen.

Wir programmieren nun eine Klasse mit einer statischen Methode und kommen hierzu nochmals auf die Klasse `Scanner` zurück, die wir bereits verwendet haben, um Eingaben von der Tastatur entgegennehmen zu können.

Die Methode `hasNext` der Klasse `Scanner` gibt immer dann `true` zurück, wenn es noch Eingaben zu bearbeiten gibt. Wird mit einem Objekt vom Typ `Scanner` eine Eingabe von der Tastatur eingelesen, so gibt die Methode `hasNext` ständig `true` zurück (Eingaben von der Tastatur werden immer als "nicht-fertig" interpretiert) – deshalb mussten wir in unseren Beispielen jeweils spezielle Wächterwerte definieren, die uns das Ende einer Benutzereingabe angeben.

Die Klasse `Scanner` bzw. deren Methode `hasNext` ist speziell interessant, wenn Eingaben bearbeitet werden sollen, die ein definiertes Ende besitzen, also z.B. Dateien oder Zeichenketten.

Gegeben sei eine Textdatei, die unter dem Namen `players.txt` gespeichert ist. In dieser Datei sind die Namen von Fussballspielern auf separaten Zeilen gespeichert:

```
John
Jim
Mike
```

Diese Datei befinde sich in einem Ordner `resources` im Projektordner `src`, in dem wir auch unsere Java Klassen gespeichert haben.

Die Klasse `CardReader` besteht aus einer statischen Konstanten `PATH_NAME` und einer statischen Methode `readPlayerFile` (→Video K-8.4). Diese Klasse soll uns die grundlegende Funktionalität bieten, Informationen zu Spielern aus einer Datei zu lesen.

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

public class CardReader {

        // Pfad und Dateiname
        private final static String PATH_NAME = "src/resources/players.txt";

        /*
         * Statische Methode zum Einlesen der Spielerinformationen.
         * Gibt eine Liste mit allen eingelesenen Spielerkarten zurück
         */
        public static ArrayList<SimplePlayerCard> readPlayerFile()
                        throws FileNotFoundException {

                // Sammlung der Karten
                ArrayList<SimplePlayerCard> cards = new ArrayList<SimplePlayerCard>();

                // Einlesen der Datei
                Scanner fileReader = new Scanner(new File(CardReader.PATH_NAME));
                while (fileReader.hasNext()){
                        String playerName = fileReader.nextLine();
                        SimplePlayerCard playerCard = new SimplePlayerCard(playerName);
                        cards.add(playerCard);
                }
                return cards;
        }
}
```

In der Methode `readPlayerFile` wird zunächst ein neues Objekt `cards` vom Typ `ArrayList` instanziiert (diese Liste soll dann die instanziierten Objekte vom Typ `SimplePlayerCard` aufnehmen). Die Variable `fileReader` wird als `Scanner` Objekt instanziiert, das auf einer Datei arbeitet. Statt den Parameter `System.in` übergeben wir dem Konstruktor in diesem Fall ein neu instanziiertes Objekt vom Typ `File`

(aus dem Java API – genauer aus dem *Package* `java.io`). Der Konstruktor von `File` wiederum nimmt den Namen einer Datei (inkl. Pfad zur Datei) entgegen (also z.B. `"src/resources/players.txt"`). Das Objekt `fileReader` ist nun bereit, Eingaben aus der definierten Datei zu lesen.

Falls aus irgendwelchen Gründen Probleme beim Auffinden oder Öffnen der Datei auftauchen sollten, so wird der Versuch, ein `File` Objekt zu instanziieren, scheitern und es entsteht ein Laufzeitfehler. Aus diesem Grund muss dem Methodenkopf von `readPlayerFile` die Klausel `throws FileNotFoundException` hinzugefügt werden (das Thema `Exception` wird im letzten Kapitel des Skriptes behandelt).

Die `while`-Schleife wird nun so lange wiederholt, wie die Methode `hasNext` des Objektes `fileReader` `true` zurückgibt. Das heisst, so lange sich noch ungelesene Zeilen in der Datei befinden, läuft diese Schleife weiter. Jeder Durchgang durch die Schleife liest mit

```
String playerName = fileReader.nextLine();
```

eine ganze Zeile aus der Datei und speichert den gelesenen Namen in der lokalen Variablen `playerName`. Mit dieser Variablen kann nun ein neues `SimplePlayerCard` Objekt instanziiert werden, welches dann der zuvor instanziierten Liste `cards` hinzugefügt wird. Die gefüllte Liste wird schliesslich durch die Methode zurückgegeben.

In der Klasse `PlayerCardTest` wird die statische Methode `readPlayerFile` zu Testzwecken aufgerufen.

```
import java.io.FileNotFoundException;
import java.util.ArrayList;

public class PlayerCardTest {

        /**
         * Demonstriert den Aufruf einer statischen Methode
         */
        public static void main(String[] args) throws FileNotFoundException {

                ArrayList<SimplePlayerCard> cards = CardReader.readPlayerFile();

                System.out.println("Aktuelle Karten:");
                for (SimplePlayerCard pc : cards)
                        System.out.println(pc);
        }
}
```

Beachten Sie insbesondere, dass für die Verwendung der Funktionalität kein Objekt instanziiert werden muss – der Aufruf von `readPlayerFile` erfolgt auf dem Klassennamen `CardReader`[1]. Die Rückgabe der Methode `readPlayerFile` wird der Variablen `playerCards` vom Typ `ArrayList` zugewiesen. Der Inhalt der Liste wird dann mit einer `for-each` Schleife ausgegeben.

Hinweis: in UML werden die Bezeichner für statische Variablen und statische Methoden unterstrichen dargestellt.

8.4 Beziehungen zwischen Klassen

Klassen bzw. die instanziierten Objekte in einem objektorientierten Softwaresystem stehen in verschiedenen Beziehungen zueinander. Drei wichtige Beziehungstypen sind *Abhängigkeit*, *Aggregation* und *Vererbung.* (engl. *dependency*, *aggregation*, *inheritence*). Gegeben seien zwei Klassen `A` und `B`, dann widerspiegeln diese drei Beziehungstypen folgende Situationen:

- Abhängigkeit: `A` *benutzt* `B`
- Aggregation: `A` *besteht (zum Teil) aus* `B`
- Vererbung: `A` *ist ein* `B`

Die ersten beiden Beziehungstypen (Abhängigkeit und Aggregation) werden wir in diesem Abschnitt besprechen – das Konzept der Vererbung wird im nächsten Kapitel gesondert behandelt.

8.4.1 Abhängigkeit

Klasse `A` ist zum Beispiel abhängig von Klasse `B`, wenn eine oder mehrere Methoden der Klasse `A` eine oder mehrere Methoden der Klasse `B` aufrufen.

Sind die aufgerufenen Methoden statisch deklariert, so können diese via Klassenname `B` aufgerufen werden. In folgenden Klassen ist dies illustriert – innerhalb der Methode `methodA` wird die statische Methode `methodB` über den Klassennamen `B` aufgerufen[2].

[1] Da es Probleme beim Lesen einer Datei geben kann (in der aufgerufenen Methode `readPlayerFile`), muss auch der Methodenkopf von `main` mit der Klausel `throws FileNotFoundException` erweitert werden.

[2] Dies entspricht genau dem Vorgehen, das wir in der Klasse `PlayerCardTest` angewendet haben, um die Methode `readPlayerFile` der Klasse `CardReader` aufzurufen: `CardReader.readPlayerFile();`

```
public class A {

        /*
         * Verwendung einer statischen Methode
         */
        public void methodA() {
                B.methodB();
        }
}
```

```
public class B {

        /**
         * Statische Methode
         */
        public static void methodB() {
                // tut etwas
        }
}
```

Ist die aufgerufene Methode `methodB` aber nicht statisch deklariert, muss `A` Zugriff auf ein instanziiertes Objekt vom Typ `B` haben. Eine Möglichkeit, in `methodA` eine Referenz auf ein Objekt vom Typ `B` zu erhalten, ist in folgenden Klassen gezeigt. Die Methode `methodA` erhält ein instanziiertes Objekt `b` vom Typ `B` als Parameter. Mit Hilfe des formalen Parameters `b` kann nun die Methode `methodB` aufgerufen werden.

```
public class A {

        /*
         * Verwendung einer Methode aus Klasse B
         */
        public void methodA(B b) {
                b.methodB();
        }
}
```

```
public class B {

        /**
         * Methode methodB
         */
        public void methodB() {
                // tut etwas
        }
}
```

Ein Spezialfall dieses Szenarios ist, dass eine Methode einer bestimmten Klasse ein

Objekt der *gleichen* Klasse als Parameter erwartet. In diesem Fall ist die Klasse von sich selbst abhängig.

Beispiel 102 *Die Methode* `concat` *der Klasse* `String` *ist ein Beispiel für eine solche Abhängigkeit. Die Methode wird auf einem Objekt* `str1` *vom Typ* `String` *aufgerufen und erwartet als Parameter ein anderes Objekt* `str2` *ebenfalls vom Typ* `String`*:*

```
str3 = str1.concat(str2);
```

Das `String` *Objekt, auf dem die Operation durchgeführt wird (hier* `str1`*), erweitert sich mit der Zeichenkette* `str2`*. Das Resultat von* `concat` *ist ein neues* `String` *Objekt, das in diesem Fall der Variablen* `str3` *zugewiesen wird.*

Beispiel 103 *Wir programmieren in diesem Beispiel eine eigene Klasse, die von sich selbst abhängig ist. Die Klasse* `RationalNumber` *repräsentiert eine rationale Zahl (→ Video K-8.5).*

```
public class RationalNumber {

        private int numerator, denominator; // Zähler und Nenner

        /**
         * Konstruktor
         */
        public RationalNumber(int numerator, int denominator) {
                this.numerator = numerator;
                this.denominator = denominator;
        }

        /**
         * Addiert diese rationale Zahl mit dem Parameter und
         * gibt das Resultat als rationale Zahl zurück.
         */
        public RationalNumber add(RationalNumber otherNumber) {
                int commonDenominator = this.denominator * otherNumber.getDenominator();
                int numerator1 = this.numerator * otherNumber.getDenominator();
                int numerator2 = otherNumber.getNumerator() * this.denominator;
                int sum = numerator1 + numerator2;
                return new RationalNumber(sum, commonDenominator);
        }

        /**
         * Getter für den Zähler
         */
        public int getNumerator() {
                return this.numerator;
        }
```

```
        /**
         * Getter für den Nenner
         */
        public int getDenominator() {
                return this.denominator;
        }

        /**
         * Einfache Zeichenkettenrepräsentation
         */
        public String toString() {
                return  this.numerator + "/" + this.denominator;
        }
}
```

Der Konstruktor **`RationalNumber`** *erwartet zwei ganze Zahlen (den Zähler und den Nenner, die als Instanzvariablen gespeichert werden). Neben den Gettern und einer Methode* **`toString`** *besitzt die Klasse* **`RationalNumber`** *insbesondere die Funktionalität, sich selber zu einer anderen rationalen Zahl zu addieren – mit der Methode* **`add`**. *Die Methode* **`add`** *erweitert zunächst diese rationale Zahl und die als Parameter erhaltene rationale Zahl, so dass die beiden Brüche gleichnamig sind (also den gleichen Nenner besitzen). Danach werden die erweiterten Zähler addiert und schliesslich wird mit den gleichnamigen Nenner eine neue rationale Zahl instanziiert, die zurückgegeben wird.*

Das Programm **`RationalTester`** *instanziiert drei Objekte vom Typ* **`RationalNumber`** *(*`r1`*,* `r2` *und* `r3`*) und führt danach einige Additionen mit diesen Zahlen durch. Dies geschieht, indem die Methode* **`add`** *z.B. auf dem Objekt* `r1` *aufgerufen wird und als Parameter das Objekt* `r2` *erhält:* `r1.add(r1);`

```
public class RationalTester{

        /**
         * Instanziiert drei rationale Zahlen und führt einige Addtionen aus
         */
        public static void main(String[] args){
                RationalNumber r1 = new RationalNumber(3, 8);
                RationalNumber r2 = new RationalNumber(1, 3);
                RationalNumber r3 = new RationalNumber(3, 4);

                System.out.println("Erste Rationale Zahl r1: " + r1);
                System.out.println("Zweite Rationale Zahl r2: " + r2);
                System.out.println("Zweite Rationale Zahl r3: " + r3);

                System.out.println("r1 + r1  = " + r1.add(r1));
                System.out.println("r1 + r2  = " + r1.add(r2));
                System.out.println("r1 + r3 = " + r1.add(r3));
                System.out.println("r2 + r3 =" + r2.add(r3));
```

```
        }
}
```

Die Ausgabe des Programmes `RationalTester` *lautet:*

```
Erste Rationale Zahl r1: 3/8
Zweite Rationale Zahl r2: 1/3
Zweite Rationale Zahl r3: 3/4
r1 + r1 = 48/64
r1 + r2 = 17/24
r1 + r3 = 36/32
r2 + r3 = 13/12
```

8.4.2 Aggregation

Ein Objekt, das in Teilen aus anderen Objekten besteht, nennen wir ein *aggregiertes Objekt*. Ein Aggregiertes Objekt ist natürlich abhängig von seinen Teil-Objekten. Die Methoden des aggregierten Objektes rufen nämlich oftmals die Methoden der Objekte auf, aus denen es besteht. Das heisst, Aggregation ist eine spezielle Form der Abhängigkeit.

In der objektorientierten Programmierung ist ein aggregiertes Objekt ein Objekt, das Referenzen auf ein anderes oder mehrere andere Objekte als Instanzvariablen besitzt. Wir haben im letzten Kapitel bereits eine Aggregation programmiert: Die Klasse `PlayerCardCollection` besteht aus einer Menge von `PlayerCard` Objekten (organisiert in einem Array).

Betrachten Sie als weiteres Beispiel die folgende Klasse `A`. Diese besitzt als Instanzvariable eine Referenz auf ein Objekt vom Typ `B`. Wird es jetzt in einer Methode von `A` (z.B. in der Methode `methodA`) nötig, eine Methode aus `B` zu benutzen, kann diese Instanzvariable verwendet werden[3]:

```
public class A {

        private B b;

        /*
         * Verwendung einer Methode aus Klasse B
         */
```

[3]Was in dieser Illustration allerdings nicht gezeigt wird, ist, wo das Objekt `b` tatsächlich instanziiert worden ist – dies könnte zum Beispiel im Konstruktor von `A` geschehen.

```
        public void methodA() {
                this.b.methodB();
        }
}
```

```
public class B {

        /**
         * Methode methodB
         */
        public void methodB() {
                // tut etwas
        }
}
```

Beispiel 104 *Die folgende Klasse* `PlayerCard` *ist recht ähnlich zu unserer vorherigen Version dieser Klasse. Der Unterschied ist, dass statt mit einer Zeichenkette* `playerName`*, eine Instanzvariable* `player` *vom Typ* `Player` *deklariert ist. Das heisst, jedes* `PlayerCard` *Objekt besteht – neben dem* `int` *Wert für die Kartennummer – aus einem* `Player` *Objekt.*

```
public class PlayerCard {

        // globaler Zähler für Karten
        public static int globalCount = 0;

        // Instanzvariablen für PlayerCard Objekte
        private Player player;
        private int cardNumber;

        /**
         * Konstruktor für PlayerCard Objekte
         */
        public PlayerCard(Player player) {
                this.player = player;
                globalCount++;
                this.cardNumber = globalCount;
        }

        /**
         * Zeichenkettenrepräsentation für PlayerCard Objekte
         */
        public String toString() {
                String cardAsString = "Spieler:\t" + this.player + "\n";
                cardAsString += "Karten-Nr:\t" + this.cardNumber + "\n";
                return cardAsString;
        }
}
```

Die Klasse `Player` *repräsentiert einen Spieler. Jedes Objekt dieses Typs speichert*

einen Namen (als `String`*), die Position des Spielers (mit dem* `enum` `Position`*) und einen Jahrgang (→Video K-8.6).*

```
public class Player {

        // Instanzvariablen für Player Objekte
        private String playerName;
        private Position position;
        private int year;

        /**
         * Konstruktor für Player Objekte
         */
        public Player(String playerName, Position position, int year) {
                this.playerName = playerName;
                this.position = position;
                this.year = year;
        }

        /**
         * Zeichenkettenrepräsentation für Player Objekte
         */
        public String toString() {
                return this.playerName + " mit Jahrgang " + this.year
                                + " spielt als " + this.position.name() + ".";

        }
}
```

Je komplexer ein Objekt ist, desto grösser ist die Wahrscheinlichkeit, dass dieses als Aggregation entworfen wird. In UML wird Aggregation mit einer Linie und einer leeren Raute gekennzeichnet (die Raute ist hierbei auf der Seite der Aggregation – siehe Abb. 8.2).

In unserem Beispiel ist ein Objekt vom Typ `PlayerCardCollection` eine Aggregation aus mehreren `PlayerCard` Objekten, welche wiederum Aggregationen von `Player` Objekten sind. Ein Objekt vom Typ `Player` besteht aus dem `enum` `Position` (beachten Sie auch, wie in UML ein `enum` gekennzeichnet wird: `<<enumeration>>`).

Wie Sie sehen, macht man die Aggregationen aus `String` Objekten nicht explizit in UML Diagrammen sichtbar. `String` Objekte sind so fundamental, dass diese in UML wie primitive Datentypen behandelt werden.

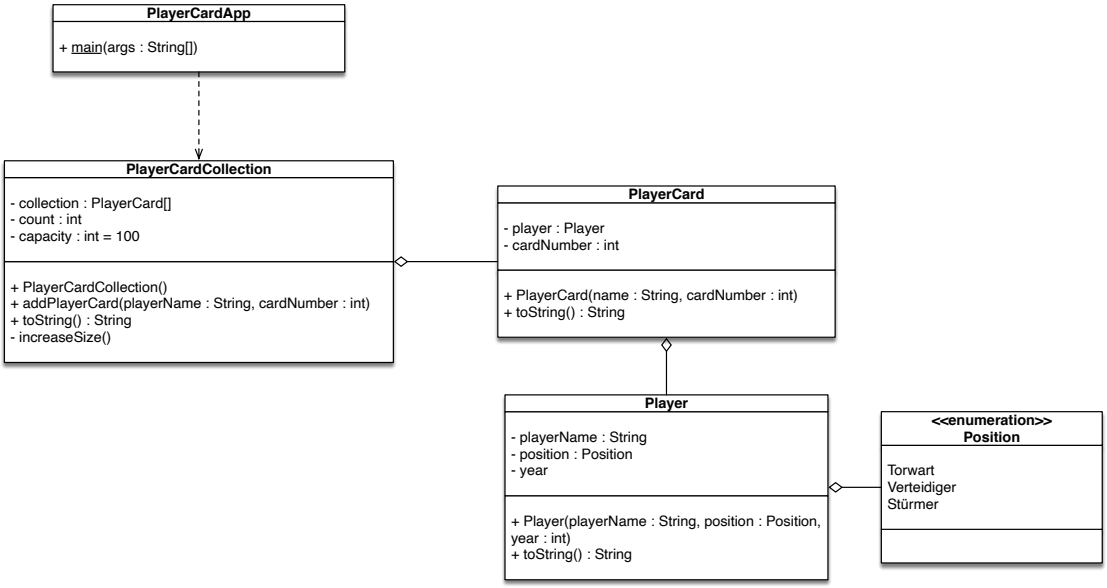

Abbildung 8.2: Ein UML Klassendiagramm, das mehrere Aggregationen zeigt.

Kapitel 9

Schnittstellen und Vererbung

9.1 Schnittstellen

Bisher haben wir die Terminologie *Schnittstelle* (engl. *Interface*) verwendet, um die Menge aller mit Sichtbarkeit `public` deklarierten Methoden zu beschreiben (also die Methoden, die man auch von ausserhalb der Klasse aufrufen kann). Wir formalisieren nun das Konzept einer Schnittstelle mit einem speziellen Element der Programmiersprache Java.

Die Struktur einer Schnittstelle erinnert an eine Klasse (oder an ein `enum`), nur steht an Stelle von `class` (bzw. `enum`) das Schlüsselwort `interface`. Ein `interface` kann Konstanten (also Variablen deklariert mit dem Modifikator `final`) und/oder abstrakte Methoden enthalten. Eine *abstrakte Methode* besteht nur aus dem Methodenkopf abgeschlossen mit einem Semikolon (besitzt also keinen Methodenrumpf).

Beispiel 105 *Nachfolgend ist ein Beispiel einer Schnittstelle* ***`Buyable`*** *mit einer abstrakten Methode* ***`getPrice`*** *gezeigt.*

```
public interface Buyable {

        /**
         * Gibt den Preis eines käufliches Objektes zurück
         */
        double getPrice();

}
```

Eine abstrakte Methode kann in einer Schnittstelle explizit mit den Modifikatoren `public` und `abstract` deklariert werden: `public abstract double getPrice();`

Alle Methoden in Schnittstellen sind automatisch abstrakt und haben die Sichtbarkeit `public`. Deshalb gibt man in Schnittstellen beides oftmals nicht an. Eine Schnittstelle kann man – im Gegensatz zu Klassen – nicht instanziieren. Eine Schnittstelle darf deshalb keinen Konstruktor deklarieren.

Man sagt: Eine Klasse *implementiert* eine Schnittstelle, wenn diese alle abstrakten Methoden der Schnittstelle mit einem Methodenrumpf ergänzt. Eine implementierende Klasse muss immer *alle* Methoden der Schnittstelle programmieren (ansonsten wird der Kompilierer eine Fehlermeldung erzeugen). Eine Klasse, die ein bestimmtes `interface` implementiert, benutzt das reservierte Wort `implements` gefolgt vom Bezeichner der Schnittstelle im Klassenkopf. Der folgende Klassenkopf gibt also zum Beispiel an, dass die Klasse `Pasta` die Schnittstelle `Buyable` implementiert:

```
public class Pasta implements Buyable
```

Beispiel 106 *Nachfolgend ist die Klasse* `Pasta` *gezeigt, die die Schnittstelle* `Buyable` *implementiert (→ Video K-9.1).*

```
public class Pasta implements Buyable {

        private String name;
        private int boilingTime;
        private double price;

        /**
         * Instanziiert ein neues Pasta Objekt mit Namen, Kochdauer und Preis
         */
        public Pasta(String name, int boilingTime, double price) {
                this.name = name;
                this.boilingTime = boilingTime;
                this.price = price;
        }

        /**
         * Implementiert die Methode getPrice
         */
        public double getPrice() {
                return this.price;
        }

        /**
         * Zeichenkettenrepräsentation eines Pasta Objektes
         */
        public String toString() {
                return "Pasta "+ this.name + " - Kochzeit: " +
                                this.boilingTime + ", Preis: " + this.price;
        }
}
```

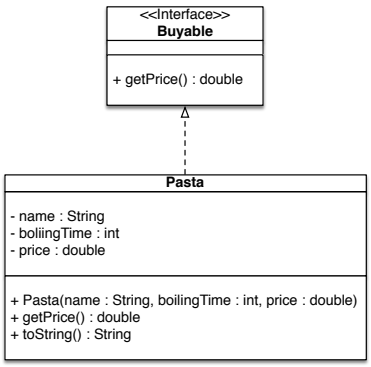

Abbildung 9.1: Ein UML Diagramm, das zeigt, dass die Klasse `Pasta` die Schnittstelle `Buyable` implementiert.

Die Methoden der Schnittstelle (in diesem Fall die Methode `getPrice`) sind nun implementiert. Das heisst, der von der Schnittstelle vorgegebene Methodenkopf erhält nun einen Methodenrumpf.

Die Verwendung einer Schnittstelle garantiert, dass die im `interface` deklarierten Methoden in allen implementierenden Klassen vorhanden sind. Es besteht aber keine Einschränkung, dass nicht noch weitere Methoden programmiert werden dürfen. Die Klasse `Pasta` zum Beispiel *muss* die Methode `getPrice` enthalten, *darf* daneben aber beliebig erweitert werden.

In UML wird eine Schnittstelle wie eine Klasse dargestellt, mit dem Unterschied, dass `<<Interface>>` oberhalb des Bezeichners der Schnittstelle steht. Eine gestrichelte Linie mit einer geschlossenen Pfeilspitze wird von der implementierenden Klasse zur Schnittstelle gezeichnet (siehe Abb. 9.1).

Mehrere Klassen können die gleiche Schnittstelle implementieren. Jede dieser Klassen kann alternative Versionen der durch die Schnittstelle vorgegebenen Methoden implementieren.

Beispielsweise könnten wir eine Klasse `Beef` programmieren, die ebenfalls die Schnittstelle `Buyable` implementiert (→Video K-9.2). In diesem Beispiel ist die Methode `getPrice` anders programmiert als in der Klasse `Pasta`: Eine Schnittstelle gibt nur vor, *dass* eine Methode vorhanden sein muss, gibt aber nicht an, *wie* die Methode tatsächlich programmiert werden soll.

Schnittstellen dienen u.a. dazu, unterschiedlichen Objekten ein gemeinsames Verhal-

ten oder eine gemeinsame Sicht zu geben. Offensichtlich sind Pasta und Rindfleisch zwei unterschiedliche Objekte, aber sie besitzen eine Gemeinsamkeit: Sie sind beide *käuflich*. Wir formalisieren diese Tatsache, indem die Klassen `Pasta` und `Beef` eine gemeinsame Schnittstelle `Buyable` implementieren.

```
public class Beef implements Buyable {

        private final double PRICE = 9.9; // Preis pro 100g
        private int weight;

        /**
         * Instanziiert ein neues Beef Objekt mit Gewicht in g
         */
        public Beef(int weight) {
                this.weight = weight;
        }

        /**
         * Implementiert die Methode getPrice
         */
        public double getPrice() {
                return this.weight * this.PRICE / 100;
        }
}
```

Beispiel 107 *In der Klasse* ***`ShoppingCart`*** *deklarieren wir fünf Objekte als statische Variablen – drei Objekte vom Typ* ***`Pasta`*** *und zwei Objekte vom Typ* ***`Beef`*** *(→ Video K-9.3).*

In der Methode ***`main`*** *wird dann eine Liste* ***`purchase`*** *instanziiert, welche Objekte vom Typ* ***`Buyable`*** *aufnehmen kann*[1]*. Danach fügen wir der Liste* ***`purchase`*** *mit der Methode* ***`add`*** *zwei Objekte vom Typ* ***`Pasta`*** *und zwei Objekte vom Typ* ***`Beef`*** *hinzu. Ohne die Idee der Schnittstellen könnten wir keine solche Liste erstellen, da eine* ***`ArrayList`*** *immer nur Objekte eines Typs aufnehmen kann.*

Als nächstes wollen wir für ***`purchase`*** *die Preissumme berechnen. Hierzu programmieren wir eine statische Methode* ***`priceSum`****, welche eine Liste* ***`purchase`*** *als Parameter entgegennimmt. Der Methode* ***`priceSum`*** *kann es egal sein, was sich tatsächlich in der Liste befindet – wichtig ist nur, dass die entsprechenden Klassen die Schnittstelle* ***`Buyable`*** *implementieren. Dies garantiert nämlich, dass der Zugriff auf die benötigte Methode* ***`getPrice`*** *funktioniert (da diese Methode von der Schnittstelle vorgegeben ist).*

[1] Etwas genauer: Objekte, deren Klassen die Schnittstelle `Buyable` implementieren.

```java
import java.text.DecimalFormat;
import java.util.ArrayList;

public class ShoppingCart {

        // alle möglichen Produkte (alle Buyable)
        private static Pasta spagetti = new Pasta("Spagetti", 10, 2.50);
        private static Pasta penne = new Pasta("Penne", 11, 2.90);
        private static Pasta linguine = new Pasta("Linguine", 9, 2.90);
        private static Beef beef1 = new Beef(125);
        private static Beef beef2 = new Beef(490);

        /**
         * Fügt Buyable Objekte einer Liste hinzu und nutzt dann
         * die statische Methode priceSum zur Berechnung des Gesamtpreises
         */
        public static void main(String[] args) {
                DecimalFormat fmt = new DecimalFormat("CHF 0.00");

                ArrayList<Buyable> purchase = new ArrayList<Buyable>();
                purchase.add(beef2);
                purchase.add(linguine);
                purchase.add(beef1);
                purchase.add(penne);

                double priceSum = priceSum(purchase);
                System.out.println("Gesamtpreis: " + fmt.format(priceSum));
        }

        /**
         * Berechnet die Preissumme von Buyable Objekten
         */
        public static double priceSum(ArrayList<Buyable> purchase){
                double sum = 0.0;
                for (Buyable b : purchase) // b ist eine polymorphe Variable
                        sum += b.getPrice();
                return sum;
        }
}
```

Beachten Sie, dass im obigen Beispiel die Variable `b` vom Typ `Buyable` zur Laufzeit einem Objekt vom Typ `Pasta` oder vom Typ `Beef` entsprechen kann. Man sagt, die Variable `b` ist *polymorph* – wir kommen am Ende dieses Kapitels nochmals auf Polymorphismus zu sprechen.

Mit Hilfe von Schnittstellen können wir ganz unterschiedliche Sichten auf ein Objekt beschreiben. Jede Schnittstelle ermöglicht eine neue Sicht auf das Objekt – eine Art Rolle. Implementiert eine Klasse diverse Schnittstellen, können ihre Instanzen in verschiedenen Rollen auftreten.

Um anzugeben, dass eine Klasse mehrere Schnittstellen implementiert, werden diese mit Kommas separiert hinter `implements` aufgezählt:

```
public class Pasta implements Buyable, Cookable
```

Ein weiterer grosser Vorteil von Schnittstellen ist, dass sie Klassen austauschbar machen. Nehmen wir an, Sie haben die Aufgabe eine Klasse `PlayerCardCollection` zu programmieren. Als Erstes sollten Sie sich klar machen, welche Funktionalitäten eine solche Klasse gegen aussen anbieten sollte und hierzu definieren Sie zunächst eine Schnittstelle. Folgende Schnittstelle gibt z.B. vor, dass eine implementierende Klasse die Methoden `addPlayerCard` und `search` anbieten muss.

```
import java.util.ArrayList;

public interface CardCollection {

        /*
         * Hinzufügen einer Karte
         */
        void addPlayerCard(PlayerCard playerCard);

        /*
         * Suchen einer Karte
         */
        ArrayList<PlayerCard> search(String searchString);

}
```

Wie diese Methoden im Detail implementiert werden sollen, ist damit noch nicht gegeben. Man könnte die Sammlung von Karten nun intern mit Hilfe eines Arrays `PlayerCard[]` (in einer Klasse `PlayerCardArray`) oder aber mit der Klasse `ArrayList` umsetzen (in einer Klasse `PlayerCardList`). Wichtig ist dabei, dass die Klasse die Schnittstelle implementiert. Also bspw.

```
public class PlayerCardArray implements CardCollection
```

Die Details der Implementationen spielen dank der Schnittstelle keine (grosse) Rolle mehr. In unserem Beispielprogramm `PlayerCardApp` müssen wir uns bei der Instanziierung der Sammlung entscheiden, welche Version wir nun tatsächlich benutzen wollen. Das heisst, wir müssen definieren, mit welcher Klasse wir die Schnittstelle `CardCollection` tatsächlich instanziieren wollen:

```
CardCollection cards = new PlayerCardArray();
```

oder

```
CardCollection cards = new PlayerCardList();
```

Die Variable `cards` ist also wiederum polymorph. Dank der Schnittstelle `CardCollection` können wir die tatsächlich implementierende Klasse beliebig ändern oder austauschen – wichtig ist nur, dass die ausgetauschte Klasse die Schnittstelle `CardCollection` implementiert.

9.1.1 Schnittstellen des Java APIs

Das Java API beinhaltet neben Klassen auch Schnittstellen. Einige dieser Schnittstellen werden nachfolgend kurz vorgestellt.

Die Schnittstelle `Comparable`

Die Schnittstelle `Comparable` ist zum Beispiel im *Package* `java.lang` definiert. Diese Schnittstelle gibt nur die Methode `compareTo` vor, welche ein Objekt als Parameter entgegennimmt und einen Wert vom Typ `int` zurückgibt.

Die Absicht dieser Schnittstelle ist es, dass implementierende Klassen einen Vergleich von Objekten anbieten müssen. Die Methode `compareTo` kann man auf einem Objekt `obj1` aufrufen und dabei ein zweites Objekt `obj2` des gleichen Typs als Parameter übergeben:

```
obj1.compareTo(obj2);
```

Wie der Dokumentation der Schnittstelle zu entnehmen ist, soll die Zahl, die `compareTo` zurückgibt, kleiner als `0` sein, wenn `obj1` kleiner ist als `obj2`. Wenn beide Objekte gleich gross sind, soll `0` zurückgegeben werden und wenn `obj1` grösser ist als `obj2`, wird eine (beliebige) Zahl grösser `0` zurückgegeben.

Grundsätzlich erwartet die Methode `compareTo` ein Objekt als Parameter. Mit spitzen Klammern hinter dem Bezeichner der Schnittstelle `Comparable` können wir die Schnittstelle parametrisieren und damit festlegen, von welchem Typ der Objektparameter sein soll (ähnlich wie wir die `ArrayList` parametrisieren):

```
public class Pasta implements Comparable<Pasta>
```

Die Methode `compareTo` erwartet nun ein Objekt vom Typ `Pasta` als Parameter.

Beispiel 108 *Es liegt an der Programmiererin jeder Klasse, zu entscheiden, wann ein Objekt kleiner als, grösser als oder gleich gross ist wie ein anderes Objekt. Folgende Version vergleicht zwei Objekte vom Typ* `Pasta` *zum Beispiel auf Basis der Kochzeit:*

```
public int compareTo(Pasta other) {
        return this.boilingTime - other.boilingTime;
}
```

Ist die Kochzeit dieses Objektes z.B. länger als die Kochzeit des Objektes `other`*, gibt die Methode eine ganze Zahl grösser* `0` *zurück (umgekehrt wird eine negative, ganze Zahl und bei gleicher Kochzeit wird* `0` *zurückgegeben – genau wie es die Idee der Schnittstelle vorgibt).*

Folgende Alternative führt den Vergleich zweier `Pasta` *Objekte auf dem Preis aus:*

```
public int compareTo(Pasta other) {
 if (this.price == other.price)
        return 0;
 else
        if (this.price > other.price)
                return 1;
        else
                return -1;
}
```

Eine weitere Alternative gibt in jedem Fall `0` *zurück – dies bedeutet, dass alle Objekte vom Typ* `Pasta` *als gleich angesehen werden.*

```
public int compareTo(Pasta other) {
        return 0;
}
```

Hinweis: Im Java API existiert eine Klasse `Collections`, welche u.a. eine statische Methode `sort` beinhaltet. Als Parameter erwartet diese Methode eine Sammlung von Objekten, deren Klasse die Schnittstelle `Comparable` implementiert. Die Sammlung kann zum Beispiel in einer `ArrayList` organisiert sein. Die Methode sortiert die Sammlung aufsteigend (und zwar gemäss der Definition von `compareTo`).

Beispiel 109 *Nehmen wir an, dass wir eine* `ArrayList pastaList` *mit* `Pasta` *Objekten zur Verfügung haben. Da die Objekte vom Typ* `Pasta` *die Schnittstelle* `Comparable` *implementieren, ist folgender Aufruf gültig:*

```
Collections.sort(pastaList);
```

Der Methode `sort` können unterschiedlich typisierte Listen übergeben werden. Das heisst, die Klasse der Objekte in der Liste ist austauschbar, solange diese die Schnittstelle `Comparable` implementieren.

Die Schnittstellen `Iterator` und `Iterable`

Eine weitere Schnittstelle des Java APIs ist `Iterator`. Sie wird von Klassen implementiert, die eine Sammlung von Objekten Element für Element durchwandern können. Beispielsweise implementiert die Klasse `Scanner` die Schnittstelle `Iterator`.

Die beiden wichtigsten Methoden der Schnittstelle `Iterator` sind `hasNext` und `next`. Beide Methoden benötigen keine Parameter. Die Methode `hasNext` gibt `true` zurück, wenn es noch unbesuchte Elemente in der Sammlung gibt, und `next` gibt das nächste Objekt der Sammlung zurück. Es obliegt der Programmiererin der Klasse, die die Schnittstelle `Iterator` implementiert, zu entscheiden, in welcher Reihenfolge die Objekte von der Methode `next` geliefert werden.

Hinweis: Die Methode `next` im Sinne der Schnittstelle entfernt das Objekt nicht aus der zugrunde liegenden Sammlung. Die Schnittstelle `Iterator` gibt aber auch eine Methode namens `remove` vor, die keine Parameter annimmt und einen Rückgabetyp `void` hat. Ein Aufruf dieser Methode soll das Objekt, das zuletzt von der Methode `next` besucht wurde, aus der Sammlung entfernen.

Eine andere Schnittstelle des Java APIs, `Iterable`, legt fest, dass ein Objekt einen `Iterator` bereitstellen muss (beispielsweise implementiert die Klasse `ArrayList` die Schnittstelle `Iterable`). Die Schnittstelle `Iterable` schreibt hierzu eine Methode `iterator` vor, die ein Objekt vom Typ `Iterator` zurückgibt. Jedes Objekt vom Typ `Iterable` kann zudem mit einer `for-each` Schleife durchwandert werden.

Beachten Sie den Unterschied: Ein `Iterator` schreibt Methoden vor, die ein Durchwandern durchführen können, während ein `Iterable` auf Anfrage ein Objekt vom Typ `Iterator` liefern wird.

9.2 Vererbung

Vererbung ist eines der Kernkonzepte der objektorientierten Programmierung. Im Allgemeinen können neue Klassen via Vererbung schneller und einfacher erstellt werden, als wenn diese immer wieder von Grund auf neu programmiert werden. Bedenken Sie beim Programmieren immer, dass einige der Klassen, die Sie programmieren müssen, vielleicht bereits existieren könnten. Manchmal bilden diese die Anforderungen der neuen Klassen nicht perfekt ab, aber sie sind möglicherweise ähnlich genug, um als Basis für eine neue Klasse dienen zu können.

Das Wort *Klasse* (`class`) lehnt sich an die Idee der *Klassifizierung* von Objekten auf Basis derer Charakteristiken an. Zum Beispiel teilen alle Bücher gewisse Charakteristiken (Titel, Autor(en), Anzahl Seiten, etc.). Eine Teilmenge von Büchern sind z.B. Sachbücher, Lexikone oder Romane. Alle Lexikone sind Bücher – ein Lexikon besitzt aber auch eigene charakteristische Merkmale, die es von anderen Büchern (z.B. einer Gedichtsammlung) unterscheidet.

Übersetzen wir diese Idee in die Programmierung: Gegeben sei eine Klasse `Book` (mit Variablen, die das Buch beschreiben, und Methoden, die z.B. dessen Status verändern können). Eine Klasse `Dictionary` kann nun aus der Klasse `Book` abgeleitet werden. In Java kann man dies mit dem Schlüsselwort `extends` angeben:

```
public class Dictionary extends Book
```

`Dictionary` besitzt automatisch alle Variablen und Methoden, die wir bereits in `Book` programmiert haben. In der Klasse `Dictionary` können wir also alle Variablen und Methoden der Klasse `Book` verwenden (ohne sie nochmals in `Dictionary` zu deklarieren). Zusätzliche Variablen und Methoden, die nur für die Klasse `Dictionary` nötig sind, können einfach hinzugefügt werden.

Die Klasse, die benutzt wird, um eine neue Klasse abzuleiten, wird *Superklasse* genannt. Die abgeleitete Klasse nennt man *Subklasse*.

Der Prozess des Vererbens etabliert eine *ist-ein-Beziehung* zwischen zwei Klassen (Z.B.: Ein Lexikon *ist-ein* Buch). Für jede Klasse `X`, die aus `Y` abgeleitet wird, muss man sagen können "`X` ist ein `Y`". Falls eine solche Aussage keinen Sinn machen sollte, ist die Beziehung zwischen `X` und `Y` möglicherweise nicht geeignet für eine Vererbung.

In der folgenden Klasse `Book` speichern wir den Titel und die Anzahl Seiten dieses Buches als Instanzvariablen `title` und `pages` (→Video K-9.4). Beide Instanzvariablen werden via Konstruktor initialisiert und es gibt einen *Getter* für `title` und einen *Setter* für `pages` sowie eine Methode `toString`.

```
public class Book {

        protected String title;
        protected int pages;

        /**
         * Konstruktor für Book Objekte
         */
        public Book(String title, int pages) {
                this.title = title;
                this.pages = pages;
        }

        /**
         * Getter für den Titel des Buches
         */
        public String getTitle() {
                return this.title;
        }

        /**
         * Getter für die Anzahl Seiten des Buches
         */
        public void setPages(int pages) {
                this.pages = pages;
        }

        /**
         * Zeichenkettenrepräsentation des Buches
         */
        public String toString() {
                return this.title + ", " + this.pages + " Seiten";
        }
}
```

Jede Methode oder Variable, die in der Superklasse mit der Sichtbarkeit `public` deklariert wird, kann in der Subklasse verwendet werden. Andererseits werden Methoden oder Variablen, die mit `private` deklariert sind, *nicht* vererbt.

Wenn wir aber eine Instanzvariable mit `public` modifizieren würden, so dass die Subklasse darauf zugreifen kann, verletzen wir das Prinzip der Kapselung (nur das Objekt soll auf seine Instanzvariablen zugreifen können). Deshalb bietet Java einen dritten Sichtbarkeitsmodifikator an: `protected`.

Die Variablen `title` und `pages` in der Klasse `Book` sind mit Sichtbarkeit `protected` deklariert. Diese Sichtbarkeit erlaubt es, einen Teil der Kapselung zu bewahren. So ist der Zugriff auf eine mit `protected` deklarierte Variable oder Methode nur innerhalb des *Packages* möglich, in dem sich die Klasse selbst befindet – `protected` kann als Mittelweg zwischen `public` und `private` angesehen werden:

- `private`: Zugriff ausschliesslich aus der eigenen Klasse.
- `protected`: Zugriff aus Klassen aus dem gleichen *Package*.
- `public`: Zugriff aus jeder Klasse.

Ein Mittelweg zwischen `private` und `protected` wäre eigentlich für die Vererbung optimal, um die Variablen nur für die Subklassen sichtbar zu machen (ist in Java aber nicht gegeben).

Die folgende Klasse `Dictionary` ist abgeleitet aus der Klasse `Book` (→Video K-9.5). Damit erbt die Klasse `Dictionary` alle Methoden und Instanzvariablen von `Book`. Es ist also so, als ob diese Variablen und Methoden in `Dictionary` deklariert wären.

```
public class Dictionary extends Book {

        private int definitions;

        /**
         * Konstruktor nimmt den Titel, die Anzahl Seiten und Definitionen entgegen
         */
        public Dictionary(String title, int pages, int definitions) {
                super(title, pages); // Aufruf des Konstruktors der Superklasse
                this.definitions = definitions;
        }

        /**
         * Berechnet die durchschnittliche Anzahl Definitionen pro Seite
         * (verwendet sowohl geerbte als auch nicht geerbte Variablen)
         */
        public double definitionsPerPage(){
                return (double) this.definitions / this.pages;
        }

        /**
         * Die geerbte Methode toString wird überschrieben
         */
        public String toString(){
                String s = super.toString(); // Aufruf der Methode der Superklasse
                return s + " und " + this.definitions + " Definitionen";
        }
}
```

Beachten Sie zum Beispiel, die Methode `definitionsPerPage`. Diese Methode berechnet die durchschnittliche Anzahl Definitionen pro Seite (die Klasse `Dictionary` deklariert hierzu eine Instanzvariable `definitions` vom Typ `int` für die Anzahl Definitionen). Mit `this.pages` wird in der Methode `definitionsPerPage` die Variable `pages` referenziert, obschon diese *nicht* in `Dictionary`, sondern in der Superklasse `Book` deklariert ist.

Vererbung ist eine "Einbahnstrasse". Die Klasse `Book` kann keine Variablen oder Methoden verwenden, die in der Klasse `Dictionary` definiert sind (z.B. können wir in einem Objekt vom Typ `Book` die Methode `definitionsPerPage` nicht verwenden). Diese Restriktion macht Sinn, da die Subklasse eine spezifischere Version der Superklasse ist. Ein Lexikon hat Seiten, da alle Bücher Seiten haben, aber obschon ein Lexikon Definitionen beinhaltet, enthalten nicht alle Bücher Definitionen.

In der Klasse `Library` wird ein neues Objekt `lexicon` der Klasse `Dictionary` instanziiert und danach werden auf dem Objekt drei Methoden aufgerufen: `getTitle`, `definitionsPerPage` und die Methode `toString`. Die Methode `getTitle` ist geerbt, die anderen beiden Methoden sind in der Klasse `Dictionary` deklariert (→Video K-9.6).

```
public class Library {

        /**
         * Illustriert den Gebrauch einer Subklasse
         */
        public static void main(String[] args) {
                DecimalFormat fmt = new DecimalFormat("0.0");

                // Instanziieren einer Subklasse
                Dictionary lexicon = new Dictionary("Das Wörterbuch", 514, 57899);

                // Aufruf einer geerbten Methode
                System.out.println("Titel des Buches: " + lexicon.getTitle());

                // Aufruf einer nicht geerbten Methode
                System.out.println("Anzahl Definitionen pro Seite: " +
                                        fmt.format(lexicon.definitionsPerPage()));

                // Aufruf einer überschriebenen Methode (toString)
                System.out.println(lexicon);
        }
}
```

Die Ausgabe des Programmes lautet:

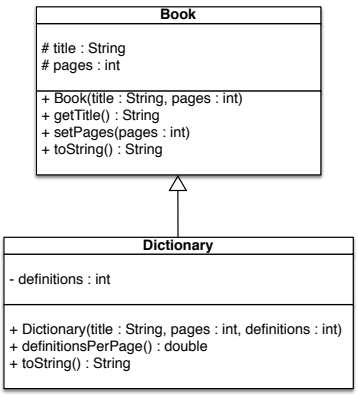

Abbildung 9.2: UML Diagramm zur Vererbung: Die Klasse `Dictionary` erbt von der Klasse `Book`.

```
Titel des Buches: Das Wörterbuch
Anzahl Definitionen pro Seite: 112.6
Das Wörterbuch, 514 Seiten und 57899 Definitionen
```

In Abb. 9.2 ist die Vererbungshierarchie zwischen `Book` und `Dictionary` in UML gezeigt. Mit einem Pfeil mit weissem Kopf zeigt man von der Subklasse zur Superklasse. Die Sichtbarkeit `protected` wird in UML mit einem vorangestellten # markiert.

Im Folgenden betrachten wir vier Aspekte der Vererbung, die wir im obigen Beispiel noch nicht besprochen haben: Den Konstruktor einer Subklasse, das Überschreiben von Methoden, den Aufbau von Klassenhierarchien und Polymorphismus.

9.2.1 Der Konstruktor einer Subklasse

Der "Bauplan" einer Subklasse basiert auf dem Plan der Superklasse. Wird ein Objekt einer Subklasse instanziiert, so werden die Baupläne beider Klassen (also Sub- und Superklasse) benötigt. Das heisst, dass die Klasse `Book` nötig ist, um ein Objekt vom Typ `Dictionary` zu instanziieren.

Beachten Sie, dass es keinen Sinn machen würde, den Konstruktor der Superklasse an eine Subklasse zu vererben. Der Konstruktor einer Subklasse ist aber dafür verantwortlich, dass als Erstes der Konstruktor der Superklasse aufgerufen wird. Dies geschieht mit folgender Anweisung:

```
super();
```

In der ersten Zeile eines Konstruktors einer Subklasse sollte immer dieser Aufruf des Konstruktors der Superklasse programmiert werden. Falls diese Anweisung nicht vorhanden ist, wird Java den Konstruktor der Superklasse automatisch aufrufen. Diese Regelung garantiert, dass eine Superklasse zuerst seine Instanzvariablen initialisiert, bevor die Subklasse daraus abgeleitet wird.

Falls der Konstruktor der Superklasse Parameter erwartet, können Sie sich nicht mehr auf den automatischen Aufruf des Superkonstruktors verlassen. In diesem Fall müssen Sie `super` selber aufrufen und die nötigen Parameter an den Superkonstruktor mitgeben.

Im obigen Beispiel wird im Konstruktor `Dictionary` als Erstes der Konstruktor der Superklasse `Book` aufgerufen. In diesem Fall erwartet der Konstruktor `Book` zwei Parameter vom Typ `String` und `int`, um den Titel und die Anzahl Seiten des Buches zu initialisieren. Sind in der Subklasse noch weitere Initialisierungen nötig, können diese *nach* dem Aufruf des Superkonstruktors programmiert werden.

```
/**
 * Konstruktor nimmt den Titel, die Anzahl Seiten und Definitionen entgegen
 */
public Dictionary(String title, int pages, int definitions) {
        super(title, pages); // Aufruf des Konstruktors der Superklasse
        this.definitions = definitions;
}
```

9.2.2 Überschreiben von Geerbten Methoden

Wenn eine Subklasse eine Methode deklariert, die den gleichen Methodenkopf besitzt wie eine Methode in der Superklasse, dann sagen wir, dass die Methode der Subklasse die Methode der Superklasse *überschreibt* (engl. *method overriding*).

Die Methode `toString` der Klasse `Book` steht durch Vererbung auch in Objekten des Typs `Dictionary` zur Verfügung. Im obigen Beispiel haben wir die Methode `toString` in `Dictionary` aber überschrieben (das heisst, neu definiert). In einer Subklasse haben wir also immer die Möglichkeit, eine geerbte Methode an die tatsächlichen Gegebenheiten anzupassen.

Die Zeichenkette, welche die Methode `toString` in Objekten vom Typ `Dictionary`

zurückgeben soll, beinhaltet nicht nur den Titel und die Anzahl Seiten sondern auch noch die Anzahl Definitionen. Da wir auf den Titel und die Anzahl Seiten zugreifen können, als wären diese in `Dictionary` deklariert, könnten wir in der neuen Version der Methode `toString` folgende `return` Anweisung programmieren:

```
return this.title + ", " + this.pages + " Seiten und " +
 this.definitions + " Definitionen";
```

Die Klasse eines Objektes bestimmt jeweils, welche Methode ausgeführt wird (die Originalversion oder die überschreibende Version). Wenn die Methode `toString` auf einem Objekt vom Typ `Book` ausgeführt wird, so wird die Originalversion ausgeführt. Wird die gleiche Methode auf einem Objekt vom Typ `Dictionary` ausgeführt, so wird die überschreibende Version der Methode ausgeführt.

Mit der Referenz `super` können wir immer auf die Methoden der Superklasse zugreifen. Folgende Anweisung ruft zum Beispiel die Methode `toString` einer Superklasse auf: `super.toString();`

Aufrufe von Methoden der Superklasse sind besonders beim Überschreiben von Methoden hilfreich. Zum Beispiel nutzen wir dies in der gezeigten Methode `toString` in der Klasse `Dictionary`.

```
        /**
         * Die geerbte Methode toString wird überschrieben
         */
        public String toString(){
                String s = super.toString(); // Aufruf der Methode der Superklasse
                return s + " und " + this.definitions + " Definitionen";
        }
```

Zuerst wird mit `super.toString()` die Originalversion der Methode `toString` aufgerufen. Die Rückgabe wird der Variablen `s` zugewiesen, dann mit den spezifischen Informationen des `Dictionary` konkateniert und schliesslich zurückgegeben.

In Java stammen alle Klassen direkt oder indirekt von der Klasse `Object` ab[2]. Wenn eine bestimmte Klasse keine Angabe macht, von welcher Klasse diese erbt (d.h. kein `extends` deklariert), so wird diese automatisch von der Klasse `Object` abgeleitet.

[2]Die Klasse `Object` gehört zum *Package* `java.lang` des Java APIs.

Da jede Klasse direkt (oder indirekt über mehrere Stufen hinweg) von der Klasse `Object` erbt, stehen in allen Klassen die mit Sichtbarkeit `public` deklarierten Methoden der Klasse `Object` zur Verfügung. Zwei wichtige Methoden der Klasse `Object` sind:

- `boolean equals(Object o)`: Gibt `true` zurück, wenn das aufrufende Objekt und der Parameter `o` Aliase sind.
- `String toString()`: Gibt den Klassennamen des Objektes gefolgt von einer ID zurück.

Üblicherweise wird die Methode `toString` überschrieben, um eine geeignete Zeichenkettenrepräsentation des Objektes zu definieren. Auch `equals` wird oft überschrieben, um spezifischere Gleichheitstests zu definieren. Z.B. können zwei Objekte als gleich angesehen werden, wenn die Werte aller (oder bestimmter) Instanzvariablen übereinstimmen.

Eine Methode kann mit dem Modifikator `final` deklariert werden. Z.B. mit

```
public final void message()
```

Eine Subklasse erbt die Methode zwar, kann in diesem Fall die Methode aber nicht überschreiben. Diese Technik kann dazu verwendet werden, um sicherzustellen, dass alle Subklassen die gleiche Version einer Methode verwenden. Der Modifikator `final` kann auch auf ganze Klassen angewendet werden. Z.B. mit

```
public final class Parameters
```

Aus einer solchen Klasse kann *keine* andere Klasse durch Vererbung abgeleitet werden.

9.2.3 Klassenhierarchien

In Java ist nur die *Einfachvererbung* (engl. *single inheritance*) erlaubt. Einfachvererbung schreibt vor, dass eine Klasse von maximal einer Klasse direkt erben kann. Umgekehrt können aber mehrere Klassen von der gleichen Klasse abgeleitet werden. Ferner kann eine Subklasse `B` einer Superklasse `A` wiederum Superklasse einer Klasse `C` sein. Vererbung entwickelt sich deshalb oftmals zu ganzen *Klassenhierarchien*.

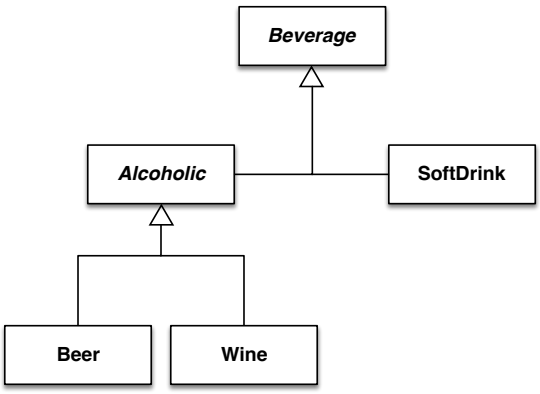

Abbildung 9.3: Eine einfache Klassenhierarchie für Getränke.

In Abb. 9.3 ist ein Beispiel einer Klassenhierarchie für verschiedene Getränke gezeigt.

Der Vererbungsmechanismus in Java ist *transitiv*. Das heisst, Variablen und Methoden werden entlang einer Kette von Super- zu Subklassen weitervererbt (jede geerbte Variable oder Methode wird wiederum weitervererbt). Das bedeutet also, dass eine geerbte Eigenschaft vom direkten Vorgänger stammen kann oder möglicherweise von einer Superklasse, die mehrere Stufen oberhalb der betrachteten Klasse liegt.

Beachten Sie: Es gibt kein Maximum für die Anzahl Kinder (Subklassen) oder für die Anzahl der Stufen in einer Hierarchie und es gibt i.a. keine "beste" Organisation für eine Klassenhierarchie – die Entscheidung, wie eine Hierarchie aufgebaut wird, liegt bei der Programmiererin.

Abstrakte Klassen dienen als Platzhalter in einer Klassenhierarchie. Wie der Name impliziert, ist eine abstrakte Klasse eine abstrakte Entität, die üblicherweise ungenügend definiert ist, um selber nützlich zu sein[3]. Stattdessen ist eine abstrakte Klasse eine Teilbeschreibung eines Konzeptes, die dann von den abgeleiteten Klassen geerbt wird. Mit dem Modifikator `abstract` wird eine abstrakte Klasse deklariert.

Wir betrachten als Beispiel das Konzept eines *Getränkes* (also zum Beispiel ein Bier, ein Glas Wein oder eine Cola). Wir definieren für das allgemeine Konzept eine Klasse `Beverage`. Diese Klasse repräsentiert kein konkretes Getränk und soll nicht instanziiert werden können und deshalb ist die Klasse `abstract` deklariert (→Video K-9.7).

[3]In UML werden die Bezeichner von abstrakten Klassen und abstrakten Methoden kursiv dargestellt.

```java
public abstract class Beverage {

        protected String name;
        protected int cl;

        /**
         * Getter für den Namen des Getränkes
         */
        public String getName() {
                return this.name;
        }

        /**
         * Getter für das Volumen des Getränkes
         */
        public int getCl() {
                return this.cl;
        }

        /**
         * Abstrakte Methode (muss von konkreten Subklassen ausformuliert werden):
         * Gibt den Alkoholgehalt eines Getränkes zurück
         */
        public abstract double getAlcoholContent();

        /**
         * Soll wahr zurückgeben, falls dieses Getränk ein Bier ist (auch abstrakt)
         */
        public abstract boolean isBeer();

        /**
         * Zeichenkettenrepräsentation des Getränkes (auch abstrakt)
         */
        public abstract String toString();

}
```

Die abstrakte Klasse `Beverage` dient als "Anker" oder Basisklasse für sämtliche Getränke der Klassenhierarchie aus Abb. 9.3 und enthält Informationen, die auf alle Typen von Getränken anwendbar sind: Den Namen des Getränkes als `String` und das Volumen in Zentilitern als `int`. Beide Instanzvariablen sind mit Sichtbarkeit `protected` deklariert, so dass diese an alle Subklassen vererbt werden.

Eine abstrakte Klasse enthält üblicherweise eine oder mehrere *abstrakte Methoden* (Methodenköpfe ohne Methodenrümpfe):

```java
public abstract double getAlcoholContent();
```

Im Gegensatz zu einer Schnittstelle ist der Modifikator `abstract` in einer abstrakten

Klasse für jede abstrakte Methode notwendig. Eine abstrakte Klasse darf – neben abstrakten Methoden – auch nicht abstrakte Methoden enthalten.

Neben den konkreten *Getter* Methoden für die zwei Instanzvariablen enthält die Klasse `Beverage` z.B. noch drei abstrakte Methoden: `getAlcoholContent`, die keine Parameter erwartet und einen Wert vom Typ `double` zurückgibt, die Methode `isBeer`, die einen `boolean` zurückgibt und die Methode `toString` (für eine Zeichenkettenrepräsentation eines Getränkes). Die Subklassen von `Beverage` werden dann ihre eigenen spezifischen Definitionen dieser Methoden zur Verfügung stellen.

Abstrakte Klassen besitzen den Standarkonstruktor und man darf in einer abstrakten Klasse auch einen eigenen Konstruktor definieren. Dieser kann aber *nicht* verwendet werden, um ein konkretes Objekt aus der abstrakten Klasse zu erzeugen (da man abstrakte Klassen *nicht* instanziieren kann). Der Konstruktor wird ausschliesslich von konkreten Subklassen der abstrakten Klasse verwendet, wenn diese instanziiert werden.

Die Klasse `Alcoholic` erbt von `Beverage` und ist ebenfalls `abstract` deklariert (→Video K-9.7). Das heisst, auch aus dieser Klasse sollen keine Objekte instanziiert werden können.

```
import java.text.DecimalFormat;

public abstract class Alcoholic extends Beverage {

        public static DecimalFormat fmt = new DecimalFormat("#0.0%");
        protected double alcoholContent;

        /**
         * Getter für Alkoholgehalt
         */
        public double getAlcoholContent() {
                return this.alcoholContent;
        }

        /**
         * Zeichenkettenrepräsentation des Getränkes
         */
        public String toString(){
                return this.name + ", " + this.cl + "cl, " +
                                "Vol. " + fmt.format(this.getAlcoholContent());
        }
}
```

Trotzdem werden nun die beiden abstrakten Methoden `getAlcoholContent` und

`toString` ausformuliert. Das ist nicht zwingend – solange die Subklassen einer abstrakten Klasse auch `abstract` sind, müssen die abstrakten Methoden nicht unbedingt ausformuliert werden – die Methode `isBeer` bleibt z.B. abstrakt.

In der Methode `getAlcoholContent` wird die Instanzvariable `alcoholContent` zurückgegeben, welche alle alkoholischen Getränke zur Verfügung stellen sollen. Die Zeichenkettenrepräsentation in der Methode `toString` gibt den Namen und das Volumen des Getränkes zusammen mit dem Alkoholgehalt zurück (der Alkoholgehalt wird mit dem statischen Objekt `fmt` vom Typ `DecimalFormat` formatiert).

Die Klasse `SoftDrink` erbt auch von `Beverage`, ist aber nicht abstrakt (→Video K-9.7).

```
public class SoftDrink extends Beverage {

        private boolean sparkling;

        /**
         * Konstruktor für nicht-alkoholische Getränke
         */
        public SoftDrink(String name, int cl, boolean sparkling){
                this.name = name;
                this.cl = cl;
                this.sparkling = sparkling;
        }

        /**
         * Bei SoftDrinks ist der Alkoholgehalt standardmässig 0.0
         */
        public double getAlcoholContent(){
                return 0.0;
        }

        /**
         * SoftDrink != Beer
         */
        public boolean isBeer() {
                return false;
        }

        /**
         * Zeichenkettenrepräsentation des Getränkes
         */
        public String toString(){
                return this.name + ", " + this.cl + "cl, " +
                                (this.sparkling ? "mit " : "ohne ") + "Kohlensäure, " +
                                "Alkoholfrei";
        }
}
```

Wir fügen die Information, ob das Getränk kohlensäurehaltig ist oder nicht, als Instanzvariable `sparkling` vom Typ `boolean` hinzu. Zudem programmieren wir für diese Klasse einen Konstruktor, der die drei Instanzvariablen (`name`, `cl` und `sparkling`) gemäss den Parametern initialisiert.

Da wir `SoftDrink` als nicht abstrakte Klasse programmieren, zwingt uns der Kompilierer die in `Beverage` definierten abstrakten Methoden `getAlcoholContent`, `isBeer` und `toString` zu programmieren. Die Methode `getAlcoholContent` gibt einfach `0.0` zurück, `isBeer` gibt `false` zurück und die Methode `toString` definiert eine Zeichenkettenrepräsentation für alkoholfreie Getränke.

Die Klasse `Beer` ist eine konkrete Subklasse der abstrakten Klasse `Alcoholic` (→Video K-9.7). Deshalb definieren wir auch hier einen Konstruktor, der den Namen des Biers und die Zentiliter gemäss den Parametern initialisiert. Der geerbten Variablen `alcoholContent` wird die Konstante `ALC_CONTENT` zugewiesen.

```
public class Beer extends Alcoholic {

        private static final double ALC_CONTENT = 0.045;

        /**
         * Konstruktor für ein Bier
         */
        public Beer(String name, int cl){
                this.name = name;
                this.cl = cl;
                this.alcoholContent = Beer.ALC_CONTENT;
        }

        /**
         * Beer!
         */
        public boolean isBeer() {
                return true;
        }
}
```

Der Unterschied der Klasse `Wine` zur Klasse `Beer` ist, dass der Konstruktor `Wine` den Alkoholgehalt als Parameter erwartet und den übergebenen Wert dann in der Instanzvariablen `alcoholContent` speichert (→Video K-9.7). Zudem gibt die Methode `isBeer` hier natürlich `false` zurück.

Beachten Sie, dass die beiden Klassen `Beer` und `Wine` die von `Beverage` vorgegebenen abstrakten Methoden `getAlcoholContent` und `toString` nicht implementieren

müssen, da diese bereits in der direkten Superklasse `Alcoholic` implementiert sind (aber man könnte beide Methoden – bei Bedarf – überschreiben).

```
public class Wine extends Alcoholic {

        /**
         * Konstruktor für einen Wein
         */
        public Wine(String name, int cl, double alcoholContent){
                this.name = name;
                this.cl = cl;
                this.alcoholContent = alcoholContent;
        }

        /**
         * Wine != Beer
         */
        public boolean isBeer() {
                return false;
        }
}
```

9.2.4 Polymorphismus

Wenn wir eine Variable mit einem bestimmten Klassennamen deklarieren, so kann diese benutzt werden, um Objekte dieser Klasse zu referenzieren (das haben wir bisher immer so gemacht).

Beispiel 110 *Wir deklarieren zum Beispiel eine Variable* `beverage` *vom Typ* `Beverage`*:*

```
Beverage beverage;
```

Wäre die Klasse `Beverage` *nicht abstrakt, so könnten wir nun ein Objekt vom Typ* `Beverage` *instanziieren und der Variablen* `beverage` *zuweisen:*

```
beverage = new Beverage();
```

Durch die Vererbungshierarchie können wir der Variablen `beverage` *aber auch beliebige Objekte eines Typs einer Subklasse von* `Beverage` *zuweisen. Zum Beispiel:*

```
beverage = new Softdrink("Cola", 33, true);
```

Diese Zuweisungen funktionieren durch die gesamte Klassenhierarchie. Da die Klasse `Alcoholic` *von* `Beverage` *erbt und die Klasse* `Wine` *von* `Alcoholic` *abgeleitet ist,*

ist die folgende Zuweisung ebenso gültig:

```
beverage = new Wine("Bordeaux", 19, 0.168);
```

Die Variable `beverage` ist somit *polymorph*, da diese Objekte verschiedener Typen referenzieren kann (zum Beispiel Objekte vom Typ `SoftDrink` oder `Wine`[4]).

In der Klasse `Bar` befindet sich die Methode `main` zum Testen der Klassenhierarchie ausgehend von `Beverage` (→Video K-9.8). Zuerst wird ein Array `beverages` instanziiert, das sechs Objekte vom Typ `Beverage` (bzw. Subklassen von `Beverage`) aufnehmen kann. Danach wird das Array mit Objekten vom Typ `SoftDrink`, `Wine` und `Beer` gefüllt (alles direkte oder indirekte Subklassen von `Beverage`, weshalb diese Zuweisungen zulässig sind). Das Array `beverages` wird also mit polymorphen Referenzen gefüllt.

```
public class Bar {

        /**
         * Testet eine Klassenhierarchie ausgehend von Beverage
         */
        public static void main(String[] args) {

                // Array für verschiedene Beverage Objekte
                Beverage[] beverages = new Beverage[6];
                beverages[0] = new SoftDrink("Cola", 33, true);
                beverages[1] = new SoftDrink("Orangensaft", 25, false);
                beverages[2] = new Wine("Bordeaux", 10, 0.168);
                beverages[3] = new Wine("Chardonnay", 10, 0.152);
                beverages[4] = new Beer("Felsenau Junker", 33);
                beverages[5] = new Beer("Quöllfrisch Lager", 50);

                Beverage beverage; // polymorphe Variable

                // Ausgeben aller Beverage Objekte
                System.out.println("Alle Getränke:");
                for (int i = 0; i < beverages.length; i++){
                        beverage = beverages[i];
                        System.out.println(beverage);
                }

                // Ausgaben aller Biere
                System.out.println("\nAlle Biere:");
                for (Beverage b : beverages)
                        if (b.isBeer())
                                System.out.println(b);
        }
}
```

[4]Das gleiche Prinzip haben wir bei Schnittstellen gesehen: Die Variable `b` vom Typ `Buyable` kann ein Objekt vom Typ `Pasta` oder `Beef` referenzieren, da beide Klassen `Buyable` implementieren.

Mit einer ersten `for`-Schleife gehen wir schliesslich durch das Array `beverages` hindurch und geben jedes Objekt `beverage` aus. Beachten Sie, dass hierzu in jedem Objekt `beverage` die `toString` Methode aufgerufen wird. Wird diese Methode von einem Objekt des Typs `Wine` ausgeführt, unterscheidet sich diese z.B. von Objekten des Typs `SoftDrink` – Der Aufruf ist aber immer der gleiche.

In der zweiten `for`-Schleife rufen wir für alle Objekte in `beverages` die Methode `isBeer` auf – Vererbung garantiert uns, dass *alle* Getränkearten diese Methode anbieten werden. Nur wenn die Methode `isBeer` `true` zurückgibt, wird das entsprechende Getränk ausgeben.

Wir könnten z.B. auch `b.getAlcoholContent()` für jedes Objekt im Array `beverages` aufrufen – diese Methode muss nämlich auch in jeder konkreten Subklasse von `Beverage` vorhanden sein.

Die Ausgabe des Programms lautet:

```
Alle Getränke
Cola, 33cl, mit Kohlensäure, Alkoholfrei
Orangensaft, 25cl, ohne Kohlensäure, Alkoholfrei
Bordeaux, 10cl, Vol. 16.8%
Chardonnay, 10cl, Vol. 15.8%
Felsenau Junker, 33cl, Vol. 4.5%
Qöllfrisch Lager, 50cl, Vol. 4.5%

Alle Biere
Felsenau Junker, 33cl, Vol. 4.5%
Qöllfrisch Lager, 50cl, Vol. 4.5%
```

Kapitel 10

Algorithmen und Methoden (Teil 1)

Jede Funktion, die ein Programm erfüllen soll, muss in irgendeiner Klasse als Methode programmiert sein. Manchmal ist es schwierig, zu entscheiden, welche Klasse die Verantwortung für eine bestimmte Funktionalität übernehmen soll. In diesem Fall lohnt es sich, verschiedene Möglichkeiten und deren Vor- und Nachteile durchzuspielen – manchmal entdeckt man in solchen Fällen auch, dass es nötig ist, eine weitere Klasse zu definieren, welche die Verantwortung übernehmen soll.

Hat man die Klassen identifiziert und ihnen die grundlegenden Verantwortungen zugewiesen, geht es darum, die Methoden zu entwerfen. Das heisst, man muss entscheiden, wie genau die Klasse ihre Funktionalität anbieten soll. Einige Methoden können sehr schnell und ohne grosse Planung programmiert werden (z.B. *Getter* und *Setter*), während andere Methoden sorgfältig geplant werden müssen.

10.1 Methoden Planen

Ein *Algorithmus* ist eine Schritt-für-Schritt Anleitung zur Lösung eines Problems (ein Algorithmus ist eine Art Rezept). Jede Methode programmiert einen Algorithmus, der bestimmt, wie die Methode ein bestimmtes Ziel erreichen oder eine bestimmte Verantwortung übernehmen kann.

Komplexere Algorithmen sollten vor der Implementierung geplant werden – dies kann zum Beispiel mit *Pseudocode* geschehen. Pseudocode ist ein Mischung von Programmieranweisungen und Prosa. Pseudocode soll genügend Struktur liefern, damit nachvollzogen werden kann, wie die Methode vorgehen soll, ohne auf syntak-

tische Details einer spezifischen Programmiersprache einzugehen. Im Vordergrund steht bei Pseudocode also das Erstellen eines Plans oder einer Anleitung, wie ein bestimmtes Problem gelöst werden soll.

10.1.1 Teilen-und-Beherrschen

Als Grundsatz gilt, dass Methoden klein und übersichtlich gehalten werden sollten, so dass diese als einfache Entität erfasst werden können. Es kann aber vorkommen, dass eine Funktionalität, die angeboten werden soll, komplex ist. Solch komplexes Verhalten in einer einzigen Methode zu programmieren, führt oftmals dazu, dass man zu grosse und unübersichtliche Methoden erstellt.

In solchen Fällen lohnt sich das Aufteilen der Funktionalität in mehrere kleine Aufgaben, die leicht zu erfüllen sind (bekannte Strategie unter dem Namen *Teilen-und-Beherrschen*). Die Vorteile von Teilen-und-Beherrschen beim Entwurf von Methoden sind:

- Bessere Verständlichkeit
- Grössere Wahrscheinlichkeit für Wiederverwendbarkeit
- Einfacheres Testen
- Einfachere Arbeitsaufteilung

Wir betrachten in diesem Abschnitt ein Beispiel, das das Vorgehen des Methodenentwurfs illustrieren soll. Wir schreiben ein Programm, das *Englisch* in sogenanntes *Pig Latin* übersetzen soll. Pig Latin ist eine Sprache, in der jedes Wort eines Satzes nach folgenden Regeln modifiziert wird.

- Bei Wörtern, die mit einem Konsonanten beginnen, wird der erste Buchstabe des Wortes ans Ende gestellt und mit einem *ay* ergänzt. Das Wort *happy* wird somit zu *appyhay* oder das Wort *birthday* wird zu *irthdaybay*.
- Kombinationen von Konsonanten wie bspw. “ch” oder “st” am Anfang eines Wortes werden zusammen ans Ende des Wortes gerückt und mit einem *ay* ergänzt (*grapefruit* wird zu *apefruitgray*).
- Wörter die mit einem Vokal beginnen, werden mit einem *yay* am Ende des Wortes ergänzt (*enough* wird zu *enoughyay*).

Algorithm 1 Pig-Latin Translator

```
while der Benutzer einen Satz übersetzen möchte do
    sentence = Satz vom Benutzer einlesen
    result = ""
    for jedes Wort w aus sentence do
        if w beginnt mit einem Vokal then
            result += w + "yay "
        else
            if w beginnt mit einem Doppelkonsonanten then
                result += w[2:] + w[:2] + "ay "
            else
                result += w[1:] + w[:1] + "ay "
            end if
        end if
    end for
    print(result)
end while
```

Der Pseudocode in Alg. 1 könnte zur Lösung unseres Übersetzungsproblems verwendet werden. Wie wir an Alg. 1 sehen, ist die Aufgabe einen ganzen englischen Satz in Pig Latin zu übersetzen nicht ganz trivial. Eine Methode, die diesen Service komplett implementiert, wird sehr gross und unübersichtlich werden. Eine bessere Lösung ist es, die Aufgaben und Verantwortungen des Algorithmus auf mehrere Methoden aufzuteilen. Wir identifizieren folgende Verantwortungen und Aufgaben:

1. Interaktion mit dem Benutzer zur Eingabe eines Satzes, zur Anzeige der Übersetzung und zum Beenden des Programmes.
2. Einen Satz in Pig Latin übersetzen. Diese Aufgabe besteht aus folgenden Teilaufgaben:
   (a) Überprüfung, ob ein Wort mit einem Vokal beginnt
   (b) Überprüfung, ob ein Wort mit einem Doppelkonsonanten beginnt
   (c) Ein einzelnes Wort in Pig Latin übersetzen

Wir starten mit Verantwortung 1. Hierzu definieren wir die Klasse `PigLatin` und die Methode `main`, die vom Benutzer einen Englischen Satz erfragt, diesen übersetzen lässt und die Übersetzung anschliessend ausgibt. Am Schluss dieses Prozesses erhält der Benutzer die Möglichkeit, einen weiteren Satz übersetzen zu lassen bzw. das Programm zu beenden (→Video K-10.1).

Wichtig: Die Methode `main` ist dabei nur die Schnittstelle zum Benutzer und sozusagen der Treiber des Programmes – das heisst, die Verantwortungen 2. (a), (b) und (c) sind allesamt ausgelagert.

```java
import java.util.Scanner;

public class PigLatin {

        /**
         * Liest einen oder mehrere Sätze ein und übersetzt diese(n) in Pig Latin
         */
        public static void main(String[] args){
                String sentence, result, another;

                Scanner scan = new Scanner(System.in);

                do {
                        System.out.println("Enter a sentence (no punctuation):");
                        sentence = scan.nextLine();

                        System.out.println();
                        result = PigLatinTranslator.translate(sentence);
                        System.out.println("That sentence in Pig Latin is:");
                        System.out.println(result);

                        System.out.println();
                        System.out.print("Translate another sentence (y/n)? ");
                        another = scan.nextLine();
                } while(another.equalsIgnoreCase("y"));
        }
}
```

Die ganze Arbeit des Übersetzens erledigen wir in der Klasse `PigLatinTranslator` und deren statischen Methode `translate`, der wir den eingelesenen Satz als Parameter übergeben (Verantwortung 2):

```java
PigLatinTranslator.translate(sentence);
```

Die Methode `translate` ersetzt die Eingabe `sentence` zunächst mit einer Kopie aus Kleinbuchstaben. Danach wird aus `sentence` mit Hilfe eines neuen `Scanner` Objektes jedes einzelne Wort des Satzes an die Support-Methode `translateWord` übergeben (Verantwortung 2 (c)). Diese Methode übersetzt das einzelne Wort in Pig Latin und gibt das Resultat zurück.

Die Methode `translateWord` verwendet wiederum zwei weitere Support-Methoden, nämlich `beginsWithVowel` und `beginsWithBlend`. Diese beiden Methoden geben `true` zurück, wenn das übergebene Wort mit einem Vokal bzw. mit einem Doppelkonsonanten beginnt (Verantwortungen 2 (a) und (b)). Je nach dem was diese beiden Methoden zurückgeben, wird dann eine der drei Regeln zur Modifikation des Wortes angewendet.

```java
import java.util.Scanner;

public class PigLatinTranslator {

        /**
         *   Übersetzt einen Satz in Pig Latin.
         */
        public static String translate(String sentence) {
                String result = "";
                sentence = sentence.toLowerCase();
                Scanner scan = new Scanner(sentence);

                while (scan.hasNext()){
                        result += translateWord(scan.next());
                        result += " ";
                }
                return result;
        }

        /**
         *  Übersetzt ein Wort in Pig Latin.
         */
        private static String translateWord(String word) {
                String result = "";

                if (beginsWithVowel(word))
                        result = word + "yay";
                else
                        if (beginsWithBlend(word))
                                result = word.substring(2) + word.substring(0,2) + "ay";
                        else
                                result = word.substring(1) + word.charAt(0) + "ay";

                return result;
        }

        /**
         * Beginnt das Wort mit einem Vokal?
         */
        private static boolean beginsWithVowel(String word){
                String vowels = "aeiou";

                char letter = word.charAt(0);

                return (vowels.indexOf(letter) != -1);
        }

        /**
         * Beginnt das Wort mit einem Doppelkonsonanten?
         */
        private static boolean beginsWithBlend(String word) {
                return ( word.startsWith ("bl") || word.startsWith ("sc") ||
                                word.startsWith ("br") || word.startsWith ("sh") ||
                                word.startsWith ("ch") || word.startsWith ("sk") ||
                                word.startsWith ("cl") || word.startsWith ("sl") ||
                                word.startsWith ("cr") || word.startsWith ("sn") ||
```

```
                                word.startsWith ("dr") || word.startsWith ("sm") ||
                                word.startsWith ("dw") || word.startsWith ("sp") ||
                                word.startsWith ("fl") || word.startsWith ("sq") ||
                                word.startsWith ("fr") || word.startsWith ("st") ||
                                word.startsWith ("gl") || word.startsWith ("sw") ||
                                word.startsWith ("gr") || word.startsWith ("th") ||
                                word.startsWith ("kl") || word.startsWith ("tr") ||
                                word.startsWith ("ph") || word.startsWith ("tw") ||
                                word.startsWith ("pl") || word.startsWith ("wh") ||
                                word.startsWith ("pr") || word.startsWith ("wr") );
        }
}
```

Eine mögliche Ein- und Ausgabe:

```
Enter a sentence (no punctuation):
Functionalities should be decomposed into several methods

That sentence in Pig Latin is:
unctionalitiesfay ouldshay ebay ecomposedday intoyay everalsay ethodsmay

Translate another sentence (y/n)? n
```

10.1.2 Parameter an Methoden Übergeben

Ein weiterer wichtiger Aspekt bei der Planung und Programmierung von Methoden ist die Parameterübergabe. Parameterübergaben sind implizite Zuweisungen: Dem formalen Parameter (der Variablen, die im Methodenkopf deklariert ist) wird eine Kopie des Wertes des tatsächlichen Parameters zugewiesen.

Im Fall von primitiven Datentypen ist der formale Parameter dabei eine separate Kopie des Wertes, der übergeben wurde. Das heisst, Änderungen, die am formalen Parameter gemacht werden, haben keinen Einfluss auf den Wert des tatsächlichen Parameters. Geht der Kontrollfluss zurück zur aufrufenden Methode, besitzt der tatsächliche Parameter immer noch den gleichen Wert wie vorher.

Wenn wir einer Methode aber ein Objekt als Parameter übergeben, übergeben wir lediglich eine Referenz auf dieses Objekt (und nicht das Objekt selber). Der Wert, der also in den formalen Parameter kopiert wird, ist eine Adresse. Das heisst, in diesem Fall werden der formale und der tatsächliche Parameter Aliase voneinander. Wenn wir den Status des Objektes dann innerhalb der aufgerufenen Methode ändern, so ändern wir auch den Status des tatsächlichen Parameters.

Beispiel 111 *Das folgende Programm illustriert diese Unterschiede bei der Parameterübergabe an Methoden (→ Video K-10.2).*

```
public class ParameterTester {

        /**
         * Illustriert die Effekte bei Änderungen an formalen
         * Parametern auf die tatsächlichen Parameter
         */
        public static void main(String[] args){
                double num1 = 0.75;
                RationalNumber num2 = new RationalNumber(3, 4);
                RationalNumber num3 = new RationalNumber(1, 2);

                System.out.println("Initiale Werte:");
                System.out.println("num1\tnum2\tnum3");
                System.out.println(num1 + "\t" + num2 + "\t" + num3 + "\n");

                // Übergeben der drei Werte an eine Methode als tatsächliche Parameter
                changeValues(num1, num2, num3);

                System.out.println("Werte am Ende von main:");
                System.out.println("num1\tnum2\tnum3");
                System.out.println(num1 + "\t" + num2 + "\t" + num3 + "\n");
        }

        /**
         * Die formalen Parameter f2 und f3 sind Objektvariablen, während
         * f1 eine primitive Variable ist. Zu Testzwecken weisen wir f1 einen
         * neuen Wert zu, in Objekt f2 ändern wir den Wert via Setter und f3
         * weisen wir ein neu instanziiertes Objekt zu.
         */
        private static void changeValues(double f1, RationalNumber f2,
                        RationalNumber f3)   {
              System.out.println("Werte bei Start der Methode:");
              System.out.println("f1\tf2\tf3");
              System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");

              // modifizieren der Werte
              f1 = -999; // Zuweisung
              f2.setNumerator(1); // Ändern des Objektes via Setter
              f3 = new RationalNumber(7, 8); // Zuweisung eines neuen Objektes

              System.out.println("Werte beim Verlassen der Methode:");
              System.out.println("f1\tf2\tf3");
              System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");
           }
}
```

Zwei der tatsächlichen Parameter, die an `changeValues` *übergeben werden, sind Objekte der Klasse* `RationalNumber` *(aus einem Beispiel aus einem vorherigen Kapitel). Der erste tatsächliche Parameter ist ein primitiver Datentyp vom Typ* `double`*. Beim Aufruf der Methode* `changeValues` *werden nun die drei tatsächlichen Para-*

meter `num1`, `num2` *und* `num3` *in die formalen Parameter* `f1`, `f2` *und* `f3` *kopiert.*

Innerhalb der Methode `changeValues` *werden die drei formalen Parameter verändert. Der Variablen* `f1` *wird ein neuer* `double` *Wert zugewiesen, die Variable* `f2` *wird mit der Methode* `setNumerator` *verändert, während der letzten Variablen* `f3` *ein neu instanziiertes* `RationalNumber` *Objekt zugewiesen wird. Danach kehrt der Kontrollfluss zurück zu* `main` *und die Werte der tatsächlichen Parameter werden erneut ausgegeben.*

Das Programm führt zu folgender Ausgabe:

```
Initiale Werte:
num1      num2      num3
0.75      3/4       1/2

Werte bei Start der Methode:
f1        f2        f3
0.75      3/4       1/2

Werte beim Verlassen der Methode:
f1        f2        f3
-999.0    1/4       7/8

Werte am Ende von main:
num1      num2      num3
0.75      1/4       1/2
```

Die `double` *Variable* `num1` *wird also nicht verändert, da die Änderung an einer echten Kopie* `f1` *von* `num1` *vorgenommen wurde. Der letzte Parameter* `num3` *referenziert immer noch das Originalobjekt mit dem Originalwert. Dies kommt daher, weil das neue* `RationalNumber` *Objekt, das in der Methode* `changeValues` *instanziiert wurde, nur dem formalen Parameter* `f3` *zugewiesen worden ist. Kehrt die Methode zurück, wird* `f3` *wieder "zerstört" (d.h. an den Garbage Collector übergeben). Die einzige permanente Änderung ist die, die am zweiten tatsächlichen Parameter* `num2` *vorgenommen wurde. Die Änderung an* `f2` *wirkt sich auch auf dem kopierten Objekt* `num2` *aus (da diese Aliase voneinander sind).*

In Abb. 10.1 ist das Verhalten der formalen und tatsächlichen Parameter illustriert.

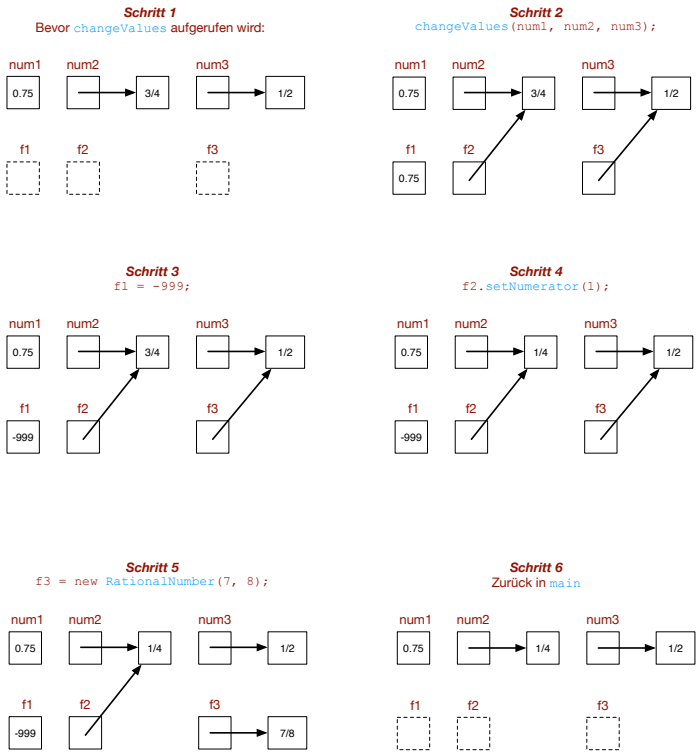

Abbildung 10.1: Formale und Tatsächliche Parameter.

10.1.3 Methoden Überladen

In Java kann man in einer Klasse den gleichen Bezeichner für unterschiedliche Methoden verwenden. Diese Technik heisst *Methodenüberladung* (engl. *method overloading*). Methodenüberladung ist hilfreich, wenn man die gleiche Methode für unterschiedliche Parameter programmieren möchte.

Wird eine Methode durch deren Bezeichner aufgerufen, wird der Kontrollfluss an diese Methode übergeben. Java muss bei jedem Aufruf einer Methode fähig sein, zu entscheiden, wohin der Kontrollfluss springen soll. Wenn also durch Methodenüberladung ein Methodenname für zwei oder mehr Methoden identisch ist, brauchen wir zusätzliche Informationen, um eindeutig bestimmen zu können, welche Methode tatsächlich ausgeführt werden soll.

In Java geschieht dies über die Parameter. Alle Methoden, die den gleichen Bezeichner haben, müssen sich zwingend bezüglich den formalen Parametern unterscheiden (also zum Beispiel eine andere Anzahl Parameter besitzen oder andere Datentypen verwenden).

Wir können in einer Klasse beispielsweise folgende Methode programmieren:

```
public int sum(int num1, int num2) {
        return num1 + num2;
}
```

In der gleichen Klasse könnten wir dann eine zweite Methode mit gleichem Bezeichner aber mit anderen Datentypen für die Parameter programmieren:

```
public double sum(double num1, double num2) {
        return num1 + num2;
}
```

Oder eine dritte Methode mit gleichem Bezeichner aber einer anderen Anzahl Parameter:

```
public int sum(int num1, int num2, int num3) {
        return num1 + num2 + num3;
}
```

Wird jetzt die Methode `sum` aufgerufen, wird Java überprüfen, wie viele tatsächliche Parameter übergeben werden bzw. welche Datentypen die Parameter besitzen. Dann wird die Methode ausgeführt, bei der die tatsächlichen Parameter mit den formalen Parametern übereinstimmen. Beispielsweise wird mit dem Aufruf

```
sum(25, 69, 13);
```

die dritte Version der Methode `sum` aufgerufen. Während

```
sum(25.0, 69.0);
```

die zweite Version aufruft.

Der Bezeichner einer Methode zusammen mit der Anzahl, den Datentypen und der Ordnung der Parameter heisst *Signatur* einer Methode. Java verwendet immer die Signatur, um die korrekte Methode aufzurufen. Versucht man zwei Methoden mit gleicher Signatur in einer Klasse zu programmieren, wird der Kompilierer eine Fehlermeldung generieren – innerhalb der gleichen Klasse müssen Methodensignaturen eindeutig sein.

Beachten Sie, dass der Datentyp der Rückgabe *nicht* zur Signatur einer Methode gehört. Das heisst, zwei überladene Methoden können sich nicht nur im Rückgabetyp unterscheiden (der Rückgabetyp wird ja beim Aufruf einer Methode nicht mitgegeben und kann somit nicht zur Unterscheidung von überladenen Methoden berücksichtigt werden).

Die Methode `println` ist ein typisches Beispiel einer mehrfach überladenen Methode. Hier drei Beispiele von Signaturen:

- `println(String s)`
- `println(int i)`
- `println(boolean b)`

Das heisst, dass die folgenden zwei Anweisungen tatsächlich *zwei unterschiedliche* Methoden mit dem gleichen Bezeichner aufrufen:

```
System.out.println("Anzahl Fehler: ");
System.out.println(17);
```

Wäre Überladung nicht erlaubt, müssten für alle Versionen einer Methode jeweils eigene Bezeichner gewählt werden – dies könnte sehr schnell unübersichtlich und mühsam werden (z.B. gäbe es dann die Methoden `printString`, `printInt`, etc. statt mehrere überladene Methoden mit dem Bezeichner `print`).

Wir haben bereits in vorderen Kapiteln Überladung von Methoden verwendet, nämlich bei der Definition verschiedener Konstruktoren. Wir haben zum Teil mehrere Konstruktoren für eine Klasse programmiert, die sich jeweils in den formalen Parametern unterschieden haben. Überladene Konstruktoren erlauben einem, Objekte auf unterschiedliche Arten zu instanziieren.

Hinweis: Verwechseln Sie das *Überladen von Methoden* nicht mit dem *Überschreiben von Methoden*: Das Überladen befasst sich mit mehreren Methoden mit dem gleichen Namen, aber unterschiedlichen Signaturen in derselben Klasse. Überschreiben befasst sich mit zwei Methoden, eine in einer Super- und eine in einer Subklasse, die die gleiche Signatur besitzen.

10.2 Algorithmen in Methoden Umsetzen

Im Folgenden wollen wir für zwei klassische Probleme (*Sortieren* und *Suchen*) Algorithmen entwerfen und diese in Java mit Methoden umsetzen.

10.2.1 Mit Algorithmen Sortieren

Unter *Sortieren* versteht man das Anordnen einer Menge von Elementen in einer genau definierten Reihenfolge. Sie möchten beispielsweise Namen alphabetisch ordnen oder Umfrageergebnisse in absteigender numerischer Reihenfolge aufstellen. Im Laufe der Jahre sind viele Sortieralgorithmen entwickelt worden und Sortieren gilt als ein klassisches Studiengebiet der Informatik. In diesem Abschnitt wird ein einfacher Sortieralgorithmus vorgestellt: *Selection Sort*[1].

Selection Sort sortiert eine Liste von Werten, indem er bestimmte Werte nacheinander an die endgültigen Positionen platziert. Die allgemeine Strategie ist also: Wir durchsuchen die gesamte Liste, um den kleinsten Wert zu finden und tauschen diesen Wert mit dem Wert, der sich an der ersten Position befindet. Danach durchsuchen wir den Rest der Liste (alle bis auf den ersten Wert), um den kleinsten Wert zu finden, und tauschen ihn mit dem Wert an der zweiten Position aus. Wir setzen diesen Vorgang für alle Werte bis auf die letzte Position in der Liste fort (die am Ende den grössten Wert enthält). Diese Strategie ist in Alg. 2 formalisiert.

Algorithm 2 Selection-Sort(*list*)

```
1: for index = 0, ..., (list.length − 2) do
2:     min = index
3:     for scan = index + 1, ..., (list.length − 1) do
4:         if list[scan] < list[min] then
5:             min = scan
6:         end if
7:     end for
8:     Tausche list[index] mit list[min]
9: end for
```

Der Algorithmus verwendet zwei `for`-Schleifen zum Sortieren. Die äussere Schleife steuert die Position *index*, an der der nächste Wert platziert werden soll. Die innere Schleife findet den kleinsten Wert im noch nicht sortierten Rest, indem sie alle Werte rechts vom gegebenen *index* durchsucht. Wenn die Position *min* des kleinsten Wertes ermittelt ist, wird der kleinste Wert mit dem Wert an Position *index* getauscht.

[1] *Selection Sort* ist bei Weitem nicht die einzige Möglichkeit, wie man sortieren kann. Es existieren zahlreiche weitere – und zum Teil effizientere – Algorithmen. In diesem Abschnitt geht es exemplarisch darum, Algorithmen zu entwerfen und diese in Java zu übersetzen.

Die Klasse `Sorting` enthält eine statische Implementation des Algorithmus *Selection Sort* (→Video K-10.3). Grosse Teile der Java Implementation sind fast identisch mit dem Pseudo-Code – andere Teile hingegen (wie zum Beispiel das Tauschen oder das Vergleichen zweier Elemente) sind etwas technischer (und diese technischen Details lässt man üblicherweise in Pseudo-Code aus).

Beachten Sie die Typvariable `<T>` in spitzen Klammern im Methodenkopf vor dem Rückgabetyp. Die Methode `selectionSort` ist also generisch programmiert. Das heisst, sie ist nicht auf die Sortierung von Arrays einer speziellen Klasse von Objekten beschränkt. Die Methode `selectionSort` ist so eingerichtet, dass sie jedes beliebige Array von Objekten sortieren kann, solange diese Objekte vergleichbar sind. Das heisst, der formale Parameter `list` ist ein Array aus Objekten des Typs `T`, welche die Schnittstelle `Comparable` implementiert: `Comparable<T>[]`[2].

Wir können durch die Methode `selectionSort` also bspw. ein Array von `Double` Objekten sortieren lassen (da `Double` die Schnittstelle `Comparable` implementiert):

```
Double[] numbers = {4.1, 5.6, 2.9, 3.6, 4.2, 1.8};
Sorting.selectionSort(numbers);
```

```
public class Sorting {

        /**
         * Implementation des Selection Sort
         */
        public static <T> void selectionSort(Comparable<T>[] list) {
                int min;
                Comparable<T> temp;
                for (int index = 0; index < list.length - 1; index++) {
                        min = index;
                        for (int scan = index + 1; scan < list.length; scan++)
                                if (list[scan].compareTo((T) list[min]) < 0)
                                        min = scan;
                        // Swapping
                        temp = list[min];
                        list[min] = list[index];
                        list[index] = temp;
                }
        }
}
```

Beispiel 112 *Jedes Objekt vom Typ* `Contact` *repräsentiert eine Person mit einem*

[2]Zur Erinnerung: `Comparable` ist eine Schnittstelle, die nur eine Methode enthält, `compareTo`, die dazu dient, eine ganze Zahl zurückzugeben, die kleiner als Null, gleich Null oder grösser als Null ist, wenn das ausführende Objekt kleiner, gleich oder grösser als das Objekt ist, mit dem es verglichen wird.

Nachnamen, einem Vornamen und einer Telefonnummer.

```
public class Contact implements Comparable<Contact> {

        private String firstName, lastName, phone;

        /*
         * Konstruktor für Kontakt Objekte
         */
        public Contact(String firstName, String lastName, String phone) {
                this.firstName = firstName;
                this.lastName = lastName;
                this.phone = phone;
        }

        /*
         * Zeichenkettenrepräsentation für Kontaktobjekte
         */
        public String toString() {
                return this.lastName + ", " + this.firstName + "\t" + this.phone;
        }

        /*
         * Gleichheitstest auf Basis von Vor- und Nachnamen
         */
        public boolean equals(Object other) {
                Contact o = (Contact) other;
                return (this.lastName.equals(o.getLastName()) &&
                                this.firstName.equals(o.getFirstName()));
        }

        /*
         * Ordnungsvergleich auf Basis von Vor- und Nachnamen
         */
        public int compareTo(Contact other) {
                int result;
                if (this.lastName.equals(other.getLastName()))
                        result = this.firstName.compareTo(other.getFirstName());
                else
                        result = this.lastName.compareTo(other.getLastName());

                return result;
        }

        /*
         * Getter für Vornamen und Nachnamen
         */
        public String getFirstName() {
                return this.firstName;
        }

        public String getLastName() {
                return this.lastName;
        }
}
```

Die Klasse `Contact` *implementiert die Schnittstelle* `Comparable` *und bietet daher eine Definition der Methode* `compareTo`*. In diesem Fall werden die Kontakte nach dem Nachnamen sortiert; wenn zwei Kontakte den gleichen Nachnamen haben, werden ihre Vornamen verwendet.*

In der Klasse `PhoneList` *wird der Aufruf der Methode* `selectionSort` *auf Objekten vom Typ* `Contact` *demonstriert.*

```
public class PhoneList {

        /**
         * Demonstriert einen Aufruf der Methode selectionSort
         */
        public static void main(String[] args) {
                Contact[] friends = new Contact[8];

                friends[0] = new Contact("John", "Smith", "610-555-7384");
                friends[1] = new Contact("Sarah", "Barnes", "215-555-3827");
                friends[2] = new Contact("Mark", "Riley", "733-555-2969");
                friends[3] = new Contact("Laura", "Getz", "663-555-3984");
                friends[4] = new Contact("Larry", "Smith", "464-555-3489");
                friends[5] = new Contact("Frank", "Phelps", "322-555-2284");
                friends[6] = new Contact("Mario", "Guzman", "804-555-9066");
                friends[7] = new Contact("Marsha", "Grant", "243-555-2837");

                Sorting.selectionSort(friends);

                for (Contact friend : friends)
                        System.out.println(friend);
        }
}
```

10.2.2 Mit Algorithmen Suchen

Wie die Sortierung ist auch die *Suche* nach einem Element ein klassisches Problem der Informatik, das sich ebenfalls für eine generische Lösung eignet. Bei der Suche geht es darum, ein bestimmtes Zielelement innerhalb einer Sammlung von Elementen zu finden. Beim Suchen müssen wir immer die Möglichkeit in Betracht ziehen, dass das Zielelement gar nicht in der Sammlung vorhanden ist. In diesem Abschnitt untersuchen wir zwei Suchalgorithmen, die *lineare Suche* und die *binäre Suche*.

Lineare Suche

Eine einfache Suche besteht darin, die Suche am Anfang der Sammlung zu beginnen und jeden Wert der Reihe nach mit dem Zielelement zu vergleichen. Schliesslich wird entweder das Zielelement gefunden oder wir kommen zum Ende der Sammlung und

stellen fest, dass das Ziel nicht vorhanden ist.

Dieser Ansatz wird als *lineare Suche* bezeichnet, weil er an einem Ende beginnt und die Sammlung linear durchläuft. Die Klasse `Searching` enthält eine statische Methode `linearSearch`, welche diese Idee implementiert (→Video K-10.5). Die Methode `linearSearch` ist wiederum generisch implementiert, um in einem Array von Objekten vom Typ `T` ein bestimmtes Objekt – ebenfalls vom Typ `T` – zu suchen (die Typvariable `T` wird erst beim Aufruf der Methode durch den Parameter festgelegt).

In `linearSearch` durchläuft die `while`-Schleife die Elemente eines Arrays und bricht entweder ab, wenn das Ziel gefunden oder das Ende des Arrays erreicht ist. Wird das Zielelement gefunden, wird dieses Element zurückgegeben (beachten Sie den Rückgabetyp `T` der Methode).

```
public class Searching {

        /**
         * Durchsucht ein Array nach einem Zielelement mit einer linearen Suche
         */
        public static <T> T linearSearch(T[] list, T target) {
                int index = 0;
                boolean found = false;

                while (!found && index < list.length) {
                        if (list[index].equals(target))
                                found = true;
                        else
                                index++;
                }

                if (found)
                        return list[index];
                else
                        return null;
        }
}
```

Das folgende Programm definiert ein unsortiertes Array von Objekten vom Typ `Contact`, führt eine lineare Suche nach einem Kontakt durch und gibt das Ergebnis aus (beachten Sie auch nochmals die Klasse `Contact` und insbesondere die Methode `equals`):

```
Found: Phelps, Frank 322-555-2284
```

```java
public class PhoneList2 {

   /**
   * Demonstriert einen Aufruf der Methode linearSearch
   */
   public static void main(String[] args) {
      Contact test, found;
      Contact[] friends = new Contact[8];

      friends[0] = new Contact("John", "Smith", "610-555-7384");
      friends[1] = new Contact("Sarah", "Barnes", "215-555-3827");
      friends[2] = new Contact("Mark", "Riley", "733-555-2969");
      friends[3] = new Contact("Laura", "Getz", "663-555-3984");
      friends[4] = new Contact("Larry", "Smith", "464-555-3489");
      friends[5] = new Contact("Frank", "Phelps", "322-555-2284");
      friends[6] = new Contact("Mario", "Guzman", "804-555-9066");
      friends[7] = new Contact("Marsha", "Grant", "243-555-2837");

      test = new Contact("Frank", "Phelps", "");
      found = Searching.linearSearch(friends, test);
      if (found != null)
         System.out.println("Found: " + found);
      else
         System.out.println("The contact was not found.");
   }
}
```

Binäre Suche

Beachten Sie, dass wir bei einer linearen Suche jedes Element untersuchen müssen, bevor wir feststellen können, dass das Ziel nicht im Array vorhanden ist. Im Durchschnitt wird bei der linearen Suche die Hälfte der Daten durchsucht, bevor ein Ziel gefunden wird, das im Array vorhanden ist.

Wenn die Elemente in einem Array entweder auf- oder absteigend sortiert sind, kann eine *binäre Suche* wesentlich effizienter sein als der lineare Suchalgorithmus. Eine binäre Suche eliminiert bei jedem Vergleich grosse Teile des Suchraums.

Beispiel 113 *Betrachten Sie das Array in Abb. 10.2. Nehmen wir an, wir wollen herausfinden, ob die Zahl 67 in dieser sortierten Menge enthalten ist. Zu Beginn könnte das Ziel an einer beliebiger Stelle im Array stehen, oder auch gar nicht. Das heisst, dass zunächst alle Elemente im Suchraum in Frage kommen. Anstatt die Suche an dem einen oder anderen Ende zu beginnen, beginnt die binäre Suche in der Mitte der sortierten Menge. Wird das Zielelement nicht an dieser mittleren Position gefunden, wird die Suche fortgesetzt. Das mittlere Element dieser Liste ist 34, was nicht unser Ziel ist, also müssen wir weiter suchen. Da die Menge jedoch sortiert*

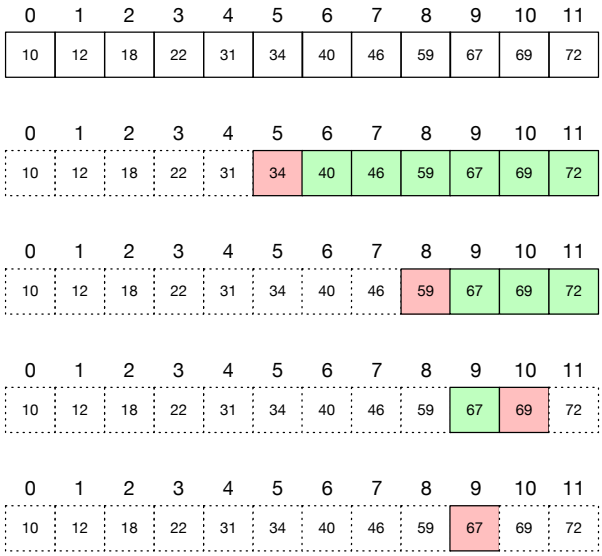

Abbildung 10.2: Die Arbeitsweise der binären Suche illustriert.

ist, wissen wir, dass 67, wenn es in der Sammlung vorkommt, in der zweiten Hälfte des Arrays zu finden ist. Alle Werte an den unteren Indizes sind kleiner als 34. Mit einem Vergleich haben wir also die Hälfte der Daten aus der Betrachtung genommen.

Um die verbleibenden Kandidaten zu suchen, untersuchen wir noch einmal das "mittlere" Element. Das mittlere Element der rechten Hälfte ist 59, und damit haben wir das Ziel immer noch nicht gefunden. Aber auch hier können wir die Hälfte der in Frage kommenden Kandidaten (diejenigen, die kleiner als 59 sind) ausschliessen.

Mit dem gleichen Ansatz wählen wir das mittlere Element, 69, und da das gesuchte Element kleiner ist als 69 suchen wir in der ersten Hälfte weiter. Im nächsten Schritt finden wir nun das gesuchte Element. Wäre es nicht unser Ziel gewesen, hätten wir diesen Prozess fortgesetzt, bis wir entweder den Wert gefunden oder alle möglichen Daten eliminiert hätten.

Die Methode `binarySearch` aus der Klasse `Searching` führt eine binäre Suche durch, indem sie eine Schleife durchläuft, bis das Zielelement gefunden wird oder bis die Anzahl der in Frage kommenden Kandidaten Null wird (→Video K-10.6). Zwei ganzzahlige Indizes, `min` und `max`, werden verwendet, um den Teil des Arrays zu definieren, der noch als brauchbar angesehen wird. Wenn `min` grösser als `max` wird, sind die in Frage kommenden Kandidaten erschöpft.

Bei jeder Iteration der Schleife wird die Mitte berechnet, indem die Summe von

`min` und `max` durch zwei geteilt wird. Gibt es derzeit eine gerade Anzahl brauchbarer Kandidaten und damit zwei "mittlere" Werte, wird bei dieser Berechnung der Bruchteil des Rests verworfen und der erste der beiden Werte ausgewählt. Wird das Zielelement nicht gefunden, wird der Wert von `min` oder `max` so geändert, dass die entsprechende Hälfte der in Frage kommenden Kandidaten eliminiert wird. Dann wird die Suche fortgesetzt.

```
public class Searching {

        /**
         * Durchsucht ein Array nach einem Zielelement mit einer binären Suche
         */
        public static <T> Comparable<T> binarySearch(Comparable<T>[] list,
                        Comparable<T> target) {
                int min = 0, max = list.length - 1, mid = 0;
                boolean found = false;

                while (!found && min <= max) {
                        mid = (min + max) / 2;
                        if (list[mid].equals(target))
                                found = true;
                        else
                                if (target.compareTo((T) list[mid]) < 0)
                                        max = mid - 1;
                                else
                                        min = mid + 1;
                }
                if (found)
                        return list[mid];
                else
                        return null;
        }
}
```

10.3 Testen von Algorithmen

Testen von Algorithmen oder von Software kann viele verschiedene Dinge bedeuten und umfasst jegliche Auswertung, die von Menschen oder Computern zur Qualitätsbeurteilung von Software durchgeführt wird. Testen umfasst mindestens das Ausführen eines vollendeten Programms mit verschiedenen Eingaben.

Ziel des Testens ist es, Fehler zu finden. Durch das Auffinden und Beheben von Fehlern erhöht sich die Sicherheit, dass ein Programm wie vorgesehen funktioniert. Es ist i.a. nicht möglich, Testfälle für alle möglichen Eingaben und Benutzeraktionen zu erstellen. Das heisst, selbst durch sorgfältiges und ausführliches Testen können wir nie wirklich sicher sein, dass alle Fehler beseitigt wurden. Es stellt sich also die

Frage: Wann sollten wir mit dem Testen aufhören?

- Konzeptionelle Antwort: Niemals
- Zynische Antwort: Wenn uns die Zeit davonläuft
- Pragmatische Antwort: Wenn wir bereit sind, zu riskieren, dass noch ein unentdeckter Fehler existiert

Ein sogenannter *Review* (manchmal auch *Inspektion* oder *Walkthrough* genannt) ist ein Treffen, bei dem mehrere Personen ein Entwurfsdokument oder einen Abschnitt eines Quellcodes prüfen. Es ist eine übliche und effektive Form des menschbasierten Testens: Die Präsentation eines Entwurfs oder eines Codes vor anderen bringt uns dazu, genauer über unsere Entscheidungen und unser Design nachzudenken.

Ein sogenannter *Test Case* ist ein Satz von Eingaben und Benutzeraktionen, gekoppelt mit den erwarteten Ergebnissen. Oftmals werden mehrere *Test Cases* formal in einer *Test Suite* organisiert, die gespeichert und bei Bedarf wiederverwendet werden. Bei mittleren und grossen Systemen muss das Testen ein sorgfältig verwalteter Prozess sein (viele Organisationen verfügen über eine eigene Abteilung für Qualitätssicherung, die für die Durchführung von Tests zuständig ist).

Man kann zwischen *Defekt-* und *Regressionstests* unterscheiden. Defekttests sind die Durchführung von *Test Cases* zur Aufdeckung von Fehlern. Die Beseitigung eines Fehlers kann aber neue Fehler einführen und deshalb werden nach der Behebung eines Fehlers Regressionstests durchgeführt. Dies bedeutet die Ausführung von früheren *Test Suites*, um sicherzustellen, dass keine neuen Fehler eingeführt wurden.

Ferner unterscheidet man zwischen *Black-Box* und *White-Box Testing*. Beim Black-Box Testing werden die Testfälle ohne Berücksichtigung der internen Logik entwickelt. Diese Tests beruhen nur auf der Eingabe und der erwarteten Ausgabe. Die Eingaben können in Äquivalenzkategorien eingeteilt werden (zwei Eingabewerte in derselben Äquivalenzkategorie werden ähnliche Ergebnisse liefern). Eine gute Testmethode deckt alle Äquivalenzkategorien ab und konzentriert sich auf die Grenzen zwischen den Kategorien.

White-Box-Tests konzentrieren sich auf die interne Struktur des Codes. Ziel ist es, sicherzustellen, dass jeder Pfad durch den Code getestet wird (Pfade durch den Code werden durch Bedingungen oder Schleifen in einem Programm gesteuert).

Kapitel 11

Sammlungen von Daten – Das *Collection-Framework*

Zum Lösen von Problemen werden oft Techniken zum Organisieren und Verwalten von Sammlungen von Daten benötigt. Eine *Sammlung* (engl. *Collection*) ist eine Art Behälter, um (meist gleichartige) Elemente abzuspeichern und zu organisieren – ein solcher Behälter nennt man auch *Datenstruktur*. Die Klasse `ArrayList` stellt zum Beispiel eine Sammlung dar, welche mehrere Objekte verwalten kann. Eine Sammlung stellt typischerweise Methoden zur Verfügung, um den Zugriff auf die Elemente und die Modifikation der Elemente zu erleichtern (zum Beispiel das Hinzufügen oder das Entfernen eines Elementes).

Im Laufe der Jahre wurden viele verschiedene Sammlungen entwickelt. In Abb. 11.1 sind vier klassischen Sammlungen illustriert:

- *Menge*: *Ungeordnete* Sammlung von Elementen, von denen jeweils maximal ein Exemplar enthalten ist.

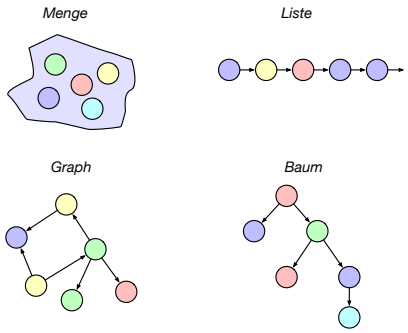

Abbildung 11.1: Illustration einer kleinen Auswahl an Sammlungen.

- *Liste*: Sammlung, in der die Elemente *linear* geordnet gespeichert sind.
- *Graph*: *Nichtlineare* Sammlung von Elementen, die jeweils über *Kanten* miteinander verbunden sein können (die Kanten können dabei gerichtet oder ungerichtet sein).
- *Baum*: Spezialfall eines Graphen: Jedes Element darf mit beliebig vielen (auch 0) Nachfolgern verbunden sein und mit Ausnahme der sogenannten *Wurzel* ist jedes Element durch eine Kante mit genau einem Vorgänger verbunden.

Keine Sammlung arbeitet für alle Zwecke gleich gut. Einige Sammlungen zeichnen sich dadurch aus, dass ihre Elemente immer sortiert bleiben – andere sind besonders gut darin, bestimmte Elemente schnell zu finden (z.B. den Vorgänger eines Elementes oder das kleinste Element der gesamten Sammlung). Bei der Entwicklung von Programmen ist es deshalb sehr wichtig, dass Sie die Vorteile und Einschränkungen verschiedener Typen von Sammlungen kennen.

11.1 Trennung von Schnittstelle und Implementierung

Ein entscheidender Aspekt von Sammlungen ist, dass das jeweilige Konzept der Sammlung auf verschiedene Arten implementiert werden kann. Das heisst, die zugrunde liegende *Datenstruktur*, in der die Elemente der Sammlung organisiert und gespeichert werden, kann mit verschiedenen Techniken implementiert werden.

Wir zeigen dieses Prinzip anhand zwei verschiedener Implementierungen des gleichen Konzeptes einer Sammlung (nämlich einer Liste).

11.1.1 Listen auf Basis von Arrays

Die Klasse `ArrayList` zum Beispiel implementiert mit der Hilfe eines Arrays die Idee einer Liste. Das heisst, die Klasse `ArrayList` speichert die einzelnen Elemente tatsächlich in einem Array fester Grösse als Instanzvariable (zu Beginn hat das zu Grunde liegende Array die Grösse `10`). Alle Operationen auf einer `ArrayList` werden durch den Aufruf von Methoden ausgeführt, die die entsprechenden Operationen auf dem zugrunde liegenden Array durchführen.

Beispiel 1 *In folgendem Beispiel ist die Funktionsweise einer Liste auf Basis eines Arrays in einer eigenen Klasse* `OwnArrayList` *illustriert (→ Video K-11.1). Diese Klasse entspricht einer starken Vereinfachung der Klasse* `ArrayList`*.*

```java
public class OwnArrayList {

	private int capacity = 10; // Startgrösse
	private int size; // Anzahl tatsächlich gespeicherter Elemente
	private Object[] listElements; // Elemente werden in Array verwaltet

	/**
	 * Konstruiert eine leere Liste mit einer Anfangskapazität von zehn.
	 */
	public OwnArrayList() {
		this.size = 0;
		this.listElements = new Object[capacity];
	}

	/**
	 * Fügt das übergebene Element an das Ende dieser Liste an.
	 */
	public boolean add(Object object) {
		if (this.size == this.capacity)
			this.increaseSize();
		this.listElements[this.size] = object;
		this.size++;
		return true;
	}

	/**
	 * Verdoppelt die Grösse des zu Grunde liegenden Arrays.
	 */
	private void increaseSize() {
		this.capacity *= 2;
		Object[] temp = new Object[this.capacity];
		for (int i = 0; i < this.listElements.length; i++)
			temp[i] = this.listElements[i];
		this.listElements = temp;
	}

	/**
	 * Liefert eine Zeichenkettenrepräsentation dieser Liste.
	 */
	public String toString() {
		String result = "";
		for (int i = 0; i < this.size; i++)
			result += this.listElements[i] + "\t";
		return result;
	}
}
```

Die Elemente, die in dieser Liste verwaltet werden sollen, werden intern in einem Array `listElements` gespeichert. Beachten Sie, dass wir der Einfachheit halber ein `Object[]` deklarieren – diese Liste kann also mit Objekten aus beliebigen Klassen gefüllt werden. Die Klasse `OwnArrayList` verwaltet intern zwei Grössen: zum einen die Anzahl der gespeicherten Elemente (`size`), zum anderen die Grösse des zu Grunde liegenden Arrays (`capacity`).

Der Einfachheit halber ist nur eine Funktionalität implementiert: Das Hinzufügen eines Objektes `object` (an das Ende der Liste) mit der Methode `add`. Ist die Kapazität des Arrays grösser als die Anzahl der Elemente, so können noch Elemente aufgenommen werden, ohne dass die Liste etwas unternehmen muss. Die Liste muss sich aber vergrössern (mit der Support Methode `increaseSize`), falls mehr Elemente aufgenommen werden, als ursprünglich vorgesehen waren. In unserer Implementierung erstellen wir hierzu ein Array `temp`, das doppelt so gross ist wie das aktuelle Array `listElements`. Danach werden alle Elemente in dieses grössere Array kopiert und schliesslich wird die Instanzvariable `listElements` mit `temp` ersetzt.

11.1.2 Listen auf Basis von Verkettung

Arrays besitzen während ihrer gesamten Existenz eine feste Grösse. Manchmal wissen wir aber nicht, wie gross wir ein Array deklarieren sollen, weil wir nicht wissen, wie viele Elemente wir speichern wollen. Die Klasse `ArrayList` handhabt dieses Problem, indem sie ein grösseres Array erstellt und die Elemente in dieses grössere Array kopiert, wann immer es nötig ist (was nicht unbedingt eine effiziente Implementierung ist).

Die Verwendung eines Arrays ist nur eine Möglichkeit, wie wir die Idee einer Liste umsetzen können – eine andere Implementierung verwendet *Verkettung mit Referenzen*. Zur Erinnerung: Eine Variable, die verwendet wird, um ein Objekt zu speichern, ist eigentlich eine Referenz auf das Objekt (was bedeutet, dass sie die Adresse des Objekts speichert). Eine Deklaration wie z.B.

```
House home = new House("602 Greenbriar Court");
```

erreicht eigentlich zwei Dinge: Sie deklariert `home` als Referenz auf ein Objekt vom Typ `House` und sie instanziiert ein Objekt der Klasse `House`. Betrachten wir nun ein Objekt, das eine Referenz auf ein anderes Objekt desselben Typs enthält (der Einfachheit halber deklarieren wir die Instanzvariablen `public`):

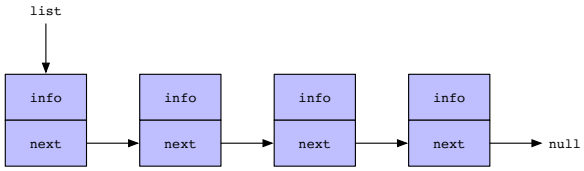

Abbildung 11.2: Eine verkettete Liste

```
public class Node {
        int info;
        Node next;
}
```

In diesem Beispiel ist die in jedem `Node`-Objekt gespeicherte Information `info` eine ganze Zahl, aber denken Sie daran, dass wir die Klasse `Node` auch so definieren könnten, dass sie eine beliebige Menge an Informationen verschiedener Typen enthält (als Instanzvariablen).

Zwei Objekte dieser Klasse können instanziiert und miteinander verkettet werden, indem die `next`-Referenz des einen `Node`-Objekts auf das andere `Node`-Objekt verweist:

```
Node first = new Node();
Node second = new Node();
first.next = second;
```

Die Referenz `next` des zweiten Objekts `second` kann auf ein drittes `Node`-Objekt verweisen, und so weiter, wodurch eine *verkettete Liste* entsteht. Der erste Knoten der Liste könnte über eine eigene Variable referenziert werden. Der letzte Knoten in der Liste würde unter `next` eine Referenz auf `null` speichern und so das Ende der Liste anzeigen (siehe Abb. 11.2).

Ein Einfügen eines Elementes in diese Liste besteht im Wesentlichen aus Umsetzen der Referenzen in den Objekten. Wird einer verketteten Liste ein Element `x` ohne Positionsangabe hinzugefügt, so kann `x` zum Beispiel "hinten" an das letzte Objekt der Liste angehängt werden. Das heisst, der Nachfolger `next` des letzten Elementes in der Liste wird auf `x` gesetzt. Wird ein Element an einer bestimmten Position in der Liste eingefügt, so müssen ebenfalls lediglich die entsprechenden Referenzen umgehängt werden (siehe Abb. 11.3).

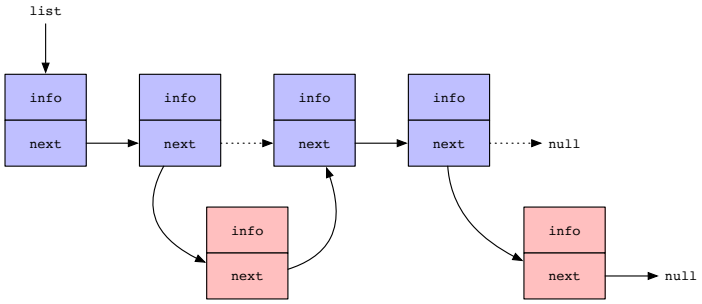

Abbildung 11.3: Einfügen in verketteten Listen entspricht "Umhängen" von Referenzen.

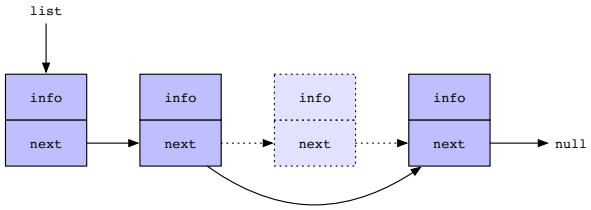

Abbildung 11.4: Ein Element wird aus der verketteten Liste entfernt.

Ein Entfernen eines Objektes `x` geschieht ebenfalls durch eine Aktualisierung der Referenzen. Im Wesentlichen muss das zu entfernende Element einfach übersprungen werden. Das heisst, im Vorgänger von `x` wird der Nachfolger auf den Nachfolger von `x` gesetzt (siehe Abb. 11.4)[1].

Nun setzen wir die Idee einer verketteten Liste in Java um (→Video K-11.2). Wir starten hierzu mit einer Klasse `Node`, welche einen einzelnen Knoten einer verketten Liste repräsentiert. Diese Klasse deklariert zwei Instanzvariablen – die Variable `element` vom Typ `E` und eine Referenz `next` vom Typ `Node`, welche auf den Nachfolgerknoten zeigt. Beachten Sie, dass `Node<E>` als generischer Typ deklariert wird (zur Erinnerung siehe Abschnitt 5.4.1). Der Typ `E` entspricht der Typvariablen – das heisst, dass der tatsächliche Datentyp `E` des Objektes `element` erst bei der Instanziierung bestimmt wird.

Wir wir wissen, kann die Variable `E` (in spitzen Klammer definiert im Klassenkopf) überall in der Klasse verwendet werden (quasi als "Platzhalter"). Bei der Instanziierung wird ein konkreter Datentyp angegeben (z.B. `Integer`):

[1] Sie wissen, dass ein Objekt durch das Entfernen aller Referenzen zu einem Kandidaten für die *Garbage-Collection* wird.

```
Node<Integer> node = new Node<Integer>(42);
```

Nun wird in der Klasse `Node` der Platzhalter `E` überall mit dem konkreten Datentypen `Integer` ersetzt. Mit anderen Worten: Objekte vom Typ `Node` können Objekte einer beliebigen andere Klasse als Instanzvariable `element` kapseln.

```
public class Node<E> {

        // Das zu speichernde Element im Knoten
        private E element;
        // Referenz auf den Nachfolgerknoten
        private Node<E> next;

        /**
         * Konstruiert einen neuen Knoten mit Element e.
         */
        public Node(E e) {
                this.element = e;
        }

        /**
         * Getter für den Nachfolgerknoten.
         */
        public Node<E> getNext() {
                return this.next;
        }

        /**
         * Setter für den Nachfolgerknoten.
         */
        public void setNext(Node<E> node) {
                this.next = node;
        }

        /**
         * Liefert eine Zeichenkettenrepräsentation dieses Knotens.
         */
        public String toString() {
                return element.toString();
        }
}
```

Der Konstruktor der Klasse `Node` nimmt ein Objekt vom Typ `E` entgegen und kapselt den Parameter als Instanzvariable `element`. Um den Nachfolgerknoten `next`

neu zu definieren bzw. diesen auszulesen, gibt es je einen *Getter* und einen *Setter* und zudem ist die Methode `toString` implementiert, welche die Zeichenkettenrepräsentation des gekapselten Objektes `element` zurückgibt.

Betrachten Sie die nachfolgende Klasse `OwnLinkedList`:

```
public class OwnLinkedList<E> {

        // Startknoten der Liste
        private Node<E> startNode;

        /**
         * Konstruiert eine leere Liste (Startknoten = null).
         */
        public OwnLinkedList(){
                this.startNode = null;
        }

        /**
         * Fügt das übergebene Element an das Ende dieser Liste an.
         */
        public void add(E element) {
                Node<E> node = new Node<E>(element);
                if (this.startNode == null)
                        this.startNode = node;
                else {
                        Node<E> current = this.startNode;
                        while (current.getNext() != null)
                                current = current.getNext();
                        current.setNext(node);
                }
        }

        /**
         * Liefert eine Zeichenkettenrepräsentation dieser Liste.
         */
        public String toString() {
                String result = "";
                Node<E> current = this.startNode;
                while (current != null) {
                        result += current + "\t";
                        current = current.getNext();
                }
                return result;
        }
}
```

Ein Objekt vom Typ `OwnLinkedList` speichert den Startknoten der Liste als Variable `startNode` vom Typ `Node`. Der Konstruktor `OwnLinkedList` setzt den Startknoten zunächst auf das leere Objekt `null`. Der Einfachheit halber ist wiederum nur die Methode `add` implementiert.

Wird mit `add` ein neues Objekt `element` vom Typ `E` zur Liste hinzugefügt, wird zuerst ein neuer Knoten `node` instanziiert, der das Objekt `element` speichern soll. Ist die Liste zu diesem Zeitpunkt noch leer (`startNode == null`), ersetzen wir den ersten Knoten der Liste mit dem neuen Knoten `node`. Andernfalls wird das Ende der Liste gesucht (mit einer `while`-Schleife). Der Variablen `next` des letzten Objektes der Liste wird dann der neue Knoten `node` zugewiesen (mit der Methode `setNext`).

Die Klasse `OwnLinkedList` kann nun instanziiert, mit Objekten gefüllt und ausgegeben werden:

```
public class MagazineRack {

        /**
         * Instanziiert eine leere Liste für Magazine Objekte,
         * füllt diese und gibt sie aus
         */
        public static void main(String[] args) {
                OwnLinkedList<Magazine> magazineList = new OwnLinkedList<Magazine>();

                magazineList.add(new Magazine("Time"));
                magazineList.add(new Magazine("Woodworking Today"));
                magazineList.add(new Magazine("Pattern Recognition"));
                magazineList.add(new Magazine("Mare"));

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

Beachten Sie insbesondere die parametrisierte Instanziierung

```
OwnLinkedList<Magazine> magazineList = new OwnLinkedList<Magazine>();
```

zur Erzeugung eines parametrisierten Typs `OwnLinkedList<Magazine>`. In diesem Beispiel füllen wir die Liste also mit Objekten vom Typ `Magazine`[2]:

[2]Die Typvariable `E` kann mit beliebigen Klassen ersetzt werden.

```java
public class Magazine {

        // Titel des Magazins
        private String title;

        /**
         * Konstruiert ein neues Magazin mit einem Tiel
         */
        public Magazine(String title) {
                this.title = title;
        }

        /**
         * Liefert eine Zeichenkettenrepräsentation dieses Magazins.
         */
        public String toString() {
                return this.title;
        }
}
```

Beachten Sie, dass die Klasse `Magazine` selbst keine Referenz auf das nächste `Magazine` in der Liste enthalten muss, da wir eine separate Klasse verwenden, um einen Knoten der Liste zu repräsentieren. Dadurch ist die Klasse `Magazine` frei von jeglichen Problemen bezüglich ihrer Einbindung in eine Liste.

Wir könnten nun fortfahren und weitere Methoden implementieren. Z.B. wäre wahrscheinlich eine zweite `add` Methode hilfreich, die nicht nur ein neues Objekt als Parameter entgegennimmt, sondern auch die Position des neuen Elementes in der Liste definiert. Weitere hilfreiche Methoden wären wahrscheinlich `set`, welche das Überschreiben eines Elementes an einer bestimmten Position ermöglicht, `delete`, um ein bestimmtes Element aus der Liste zu entfernen, oder `size`, welche die Grösse der Liste zurückgibt.

11.1.3 Abstrakte Datentypen (ADT)

Von ausserhalb der Klasse `OwnArrayList` oder `OwnLinkedList` betrachtet, interessiert uns nicht, wie die Funktionalitäten der Liste implementiert sind. Wir wissen z.B. nicht, ob die Liste die Elemente in einem Array oder durch Verkettung speichert und von aussen kümmern wir uns auch nicht darum, ob die Liste noch genügend Platz hat oder nicht. Beide Klassen stellen einfach eine Reihe von Methoden zur Verfügung (z.B. `add`), die es dem Benutzer ermöglichen, die Liste zu verwalten.

Implementationsdetails bleiben gegen aussen also verborgen – wir greifen nur über die Schnittstelle des Objektes auf die Funktionalität der Liste zu. Eine Liste ist somit ein sogenannt *abstrakter Datentyp* (*ADT*). Ein ADT besteht aus einem Namen und einer Reihe von Operationen, die ausgeführt werden können. Ein ADT trennt die Operationen von der zugrunde liegenden Implementierung. Das heisst, die Details, wie ein ADT seine Daten speichert und seine Methoden ausführt, sind von dem Konzept, das es verkörpert (z.B. das Konzept einer *Liste*), getrennt.

Objekte sind perfekt geeignet, um ADTs zu definieren. Ein Objekt hat per Definition eine wohldefinierte Schnittstelle, deren Implementierung in der Klasse verborgen ist. Die Art und Weise, wie die Daten dargestellt werden, und die Operationen, die die Daten verwalten, sind im Objekt gekapselt. Solange das Objekt die Versprechen der Schnittstelle erfüllt, ist es im Grunde egal, wie die Funktionalitäten implementiert sind. Dies hilft insbesondere dabei, Implementierungen zu ändern, ohne die Schnittstellen ändern zu müssen.

11.1.4 Weitere Implementierungsideen für Listen

Wir könnten unsere Listenimplementierungen auch spezifischeren Anforderungen anpassen. In manchen Situationen kann es beispielsweise die Verarbeitung erleichtern, eine *doppelt verkettete Liste* zu implementieren, in der jeder Knoten nicht nur eine Referenz auf den nächsten Knoten in der Liste, sondern auch eine Referenz auf den vorherigen Knoten in der Liste hat. Unsere einfache Klasse `Node` könnte wie folgt deklariert werden:

```
public class Node {
        int info;
        Node next, prev;
}
```

Abb. 11.5 zeigt eine doppelt verkettete Liste. Beachten Sie, dass, wie bei einer einfach verketteten Liste, die nächste Referenz des letzten Knotens `null` ist. In ähnlicher Weise ist der vorherige Knoten des ersten Knotens `null`, da es keinen Knoten gibt, der vor dem ersten Knoten steht. Diese Art von Struktur macht es einfach, sich zwischen den Knoten in der Liste hin und her zu bewegen, erfordert aber mehr Aufwand bei gewissen Operationen.

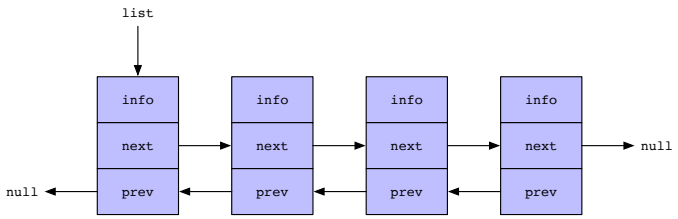

Abbildung 11.5: Eine doppelt verkette Liste.

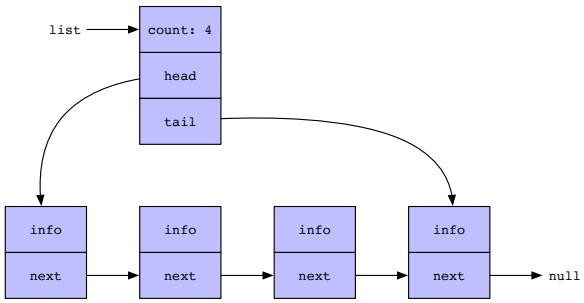

Abbildung 11.6: Eine Liste mit direkten Referenzen auf den Anfang und das Ende.

Eine andere Implementierung einer verketteten Liste könnte einen Kopfknoten für die Liste deklarieren, der eine Referenz auf den Anfang der Liste und eine weitere Referenz auf das Ende der Liste besitzt. Eine solche Referenz erleichtert insbesondere das Hinzufügen neuer Knoten am Ende der Liste. Ferner kann der Kopfknoten weitere Informationen enthalten, z.B. die Anzahl der aktuell in der Liste enthaltenen Elemente. Die Deklaration des Kopfknotens würde in etwa wie folgt aussehen:

```
public class ListHeader {
        int count;
        Node head, tail;
}
```

Beachten Sie, dass der Kopfknoten nicht von der gleichen Klasse ist wie die Klasse `Node`, auf die er verweist. Abb. 11.6 illustriert diese Idee.

Es können viele weitere Implementierungen von verketteten Listen erstellt werden. Die Implementierung sollte sich an der Art der Verarbeitung orientieren, die erforderlich ist. Ein gewisser Mehraufwand bei der Pflege einer komplexeren Datenstruktur kann sich lohnen, wenn dadurch häufige Operationen auf der Struktur effizienter werden.

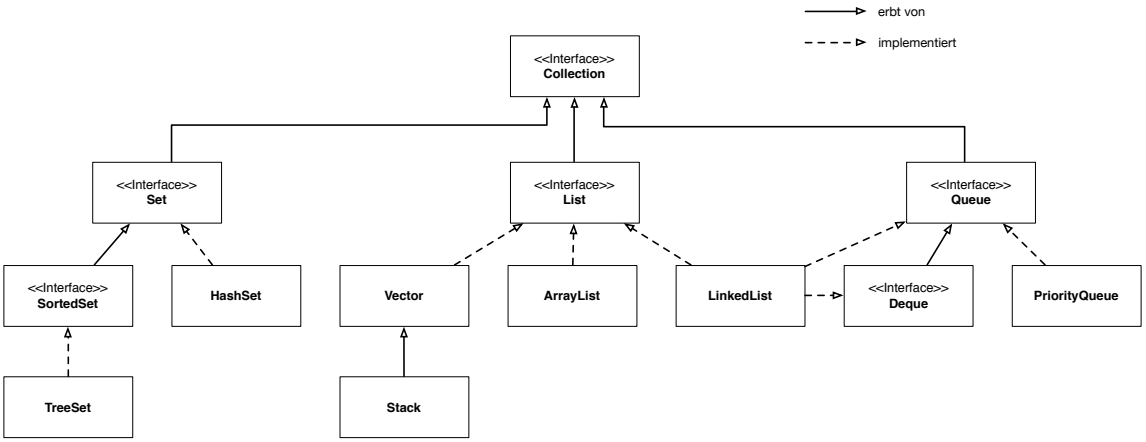

Abbildung 11.7: Die Klassenhierarchie der `Collection` API.

11.2 Das *Collection-Framework* von Java

Das Java API enthält mehrere Klassen, welche verschiedene Sammlungen repräsentieren. Diese werden oft als *Collection-Framework* bezeichnet. Das *Collection-Framework* von Java ist eine einheitliche Architektur zur Darstellung und Manipulation von *dynamischen Sammlungen*. Eine dynamische Sammlung passt ihre Grösse der tatsächlichen Anzahl Elemente an, die sie aktuell beinhaltet (das heisst, die Grösse der Sammlung muss nicht schon bei der Deklaration festgelegt werden).

Die meisten Namen der Klassen aus dem *Collection-Framework* geben Hinweise auf den Typ der dynamischen Sammlung und die tatsächliche Implementierung der Sammlung (z.B. entspricht die Klasse `ArrayList` einer Liste, welche mit einem Array implementiert ist).

Das *Collection-Framework* stellt lineare (z.B. Listen) und nicht-lineare Sammlungen (z.B. Bäume) zur Verfügung. In diesem Kapitel betrachten wir ausschliesslich lineare Datenstrukturen. Das *Collection-Framework* ist zudem in zwei Klassenhierarchien unterteilt (siehe auch Abb. 11.7 und 11.8):

- `java.util.Collection`
- `java.util.Map`

In beiden Hierarchien werden mehrere Schnittstellen verwendet. Zu diesen Schnittstellen gehören u.a. `List`, `Set`, `Queue` und `Map` – vier verschiedene Konzepte für Sammlungen.

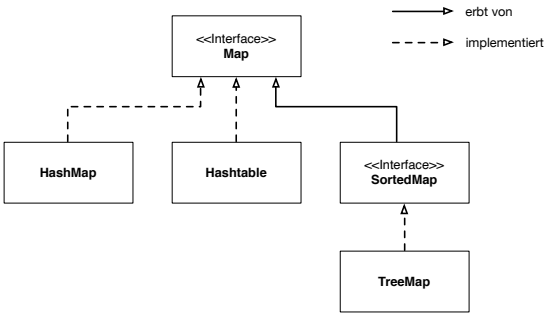

Abbildung 11.8: Die Klassenhierarchie der `Map` API.

- Eine `List` ist eine Sammlung von Elementen mit fester Ordnung.
- Ein `Set` entspricht einer Sammlung von Elementen ohne Ordnung und ohne Duplikate.
- Eine `Queue` funktioniert wie eine Liste mit Einschränkungen beim Hinzufügen und Entfernen von Elementen.
- Eine `Map` ist eine Sammlung von Elementen, die über Schlüsselwerte referenziert werden können.

Programme, welche Sammlungen verwenden, sollten nur von Schnittstellen (und nicht von Klassen) abhängig sein. Da verschiedenen Implementierungen einzelner Schnittstellen austauschbar sind, können Programme durch Austauschen der Implementierungen leicht geändert werden.

Beispielweise kann man folgende Deklaration

```
List<Integer> l = new ArrayList<Integer>();
```

ändern zu

```
List<Integer> l = new LinkedList<Integer>();
```

ohne dass der Rest des Programms geändert werden muss.

Wir betrachten im Folgenden nur Klassen, welche die Schnittstelle `List` bzw. `Queue` implementieren. Durch die gemeinsame Schnittstelle `Collection` erhalten alle implementierenden Klassen der Schnittstellen `List` und `Queue` einen gemeinsamen

Rahmen. Das heisst, alle implementierenden Klassen besitzen automatisch gemeinsame Methoden, um Daten aufzunehmen, zu manipulieren und auszulesen.

Alle Klassen des *Collection-Frameworks* sind generisch – das heisst, dass die Programmiererin bei der Instanziierung der Sammlung den Objekttypen `T` angibt, den diese verwalten soll. So kann man bspw. eine `ArrayList<String>` instanziieren, welche Zeichenketten verwaltet oder eine eine `ArrayList<Book>`, welche `Book` Objekte enthalten soll. Einige wichtige Methoden der Schnittstelle `Collection` sind bspw.:

- `boolean add(T t)`: Fügt der Sammlung das Element `t` vom Typ `T` hinzu und gibt `true` zurück, falls sich das Element einfügen lässt. Gibt `false` zurück, wenn schon ein Objekt gleichen Werts vorhanden ist und doppelte Werte in der verwendeten Sammlung nicht erlaubt sind.
- `boolean addAll(Collection c)`: Fügt alle Elemente der `Collection c` zur Sammlung hinzu.
- `void clear()`: Entfernt alle Elemente aus der Sammlung.
- `boolean contains(T t)`: Liefert `true`, falls die Sammlung ein Objekt `t` enthält (sonst `false`).
- `boolean isEmpty()`: Liefert `true`, falls die Sammlung keine Elemente enthält (sonst `false`).
- `Iterator<T> iterator()`: Liefert einen `Iterator`, mit dem über alle Elemente der Sammlung iteriert werden kann.
- `boolean remove(T t)`: Entfernt das Objekt `t` aus der Sammlung und gibt dann `true` zurück (falls das Objekt `t` nicht in der Sammlung vorhanden ist, wird `false` zurückgegeben).
- `int size()`: Gibt die Anzahl Elemente der Sammlung zurück.

11.2.1 Listen: Die Klassen `ArrayList` und `LinkedList`

Zwei implementierende Klassen der Schnittstelle `List` sind `LinkedList` (Liste durch verkettete Elemente) sowie `ArrayList` (Liste auf der Basis eines Arrays). Listen zeichnen sich durch eine feste Ordnung ihrer Elemente aus – jedes Element besitzt einen ganzzahligen Index. Zusätzlich zu den oben aufgezählten Methoden der Schnittstelle `Collection` besitzen beide Klassen deshalb weitere Methoden, die jeweils mit einem Index arbeiten:

- Zusätzlich zur Methode `add`, welche ein Objekt hinten an die Liste anhängt, gibt es noch eine Methode `add(int index, T t)`, welche das Objekt `t` an Position `index` in die Liste einfügt – verschiebt das aktuell an dieser Position befindliche Element (falls vorhanden) und alle nachfolgenden Elemente nach "rechts".
- Die Methode `set(int index, T t)` setzt das Element `t` an die Position `index` und *überschreibt* das ursprüngliche Element an dieser Position.
- Ein Element an einer speziellen Position `index` erfragen, ohne dieses zu entfernen, kann man mit der Methode `get(int index)`.
- Die Methoden `indexOf(T t)` und `lastIndexOf(T t)` liefern die erste bzw. letzte Fundposition des zu suchenden Objektes `t` (bzw. `-1` falls das Objekt `t` nicht in der Sammlung vorhanden ist).
- Zusätzlich zur Methode `remove(T t)`, der man das Objekt `t`, das entfernt werden soll, mitgeben kann, gibt es bei Listen noch die Methode `remove(int index)`, welche das Objekt an Position `index` aus der Liste entfernt und zurückgibt.

Eine `LinkedList` stellt noch weitere Hilfsmethoden zur Verfügung, um an beiden Enden der Liste Objekte hinzuzufügen, auszulesen oder zu entfernen: `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst` und `removeLast`.

Es ergeben sich Einsatzgebiete, die einmal für `LinkedList` und einmal für `ArrayList` sprechen:

- Da die Klasse `ArrayList` intern ein Array benutzt, ist der Zugriff auf ein Element an einer bestimmten Position über eine Indexangabe i.a. schneller als bei einer `LinkedList`. Bei einer `LinkedList` muss man hierfür nämlich zeitaufwändig den Referenzen folgen, bis man die entsprechende Position erreicht hat.
- Die Klasse `LinkedList` ist aber deutlich im Vorteil, wenn Elemente mitten in der Liste gelöscht oder eingefügt werden oder wenn die Liste sehr schnell wächst. In beiden Situationen müssen bei einer `ArrayList` Elemente zeitaufwändig kopiert und verschoben werden.

11.2.2 Stapel: Die Klasse `Stack`

Ein *Stapel* (engl. *Stack*) ist wie eine Liste eine lineare Datenstruktur. Im Gegensatz zu Listen ist bei Stapeln aber das Element, das als nächstes aus der Sammlung ent-

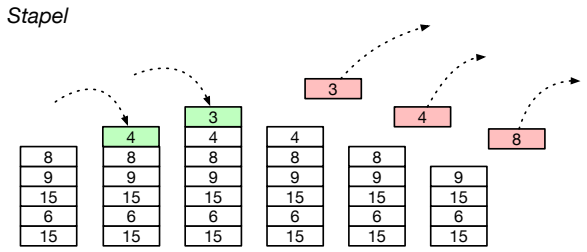

Abbildung 11.9: Die Arbeitsweise eines Stapels an einem Beispiel illustriert.

fernt werden soll, durch die Sammlung vorbestimmt. Man kann in einem Stapel also nicht via Index auf ein beliebiges Element der Sammlung zugreifen.

In einem Stapel können Elemente nur "oben" hinzugefügt werden und auch nur von dort wieder gelesen, bzw. entfernt werden. Das heisst, die hinzugefügten Elemente werden übereinander gestapelt und in umgekehrter Reihenfolge vom Stapel genommen[3]. In einem Stapel wird also zu jedem Zeitpunkt als nächstes das Element entfernt, das als letztes zur Sammlung hinzugefügt wurde (bekannt als Prinzip *last-in-first-out* (*LIFO*)).

Auf einem Stapel wird das Hinzufügen eines Elements häufig mit *Push* bezeichnet. Das Entfernen eines Objektes wird *Pop* genannt und das Nachsehen des obersten Objektes auf dem Stapel (ohne dieses zu entfernen) nennt man i.a. *Peek* (Weder *Pop* noch *Peek* benötigen ein Argument).

Beispiel 2 *In Abb. 11.9 ist die Arbeitsweise eines Stapels illustriert. Zuerst werden nacheinander Push(*4*) und Push(*3*) ausgeführt, danach drei mal Pop.*

Die Klasse `Stack` implementiert die Idee eines Stapels in Java. Auf einem Objekt des Typs `Stack` wird mit `push(T t)` das Element `t` auf den Stapel gebracht. Mit `pop()` holt man das "oberste" Element vom Stapel. Mit `peek()` wird das oberste Element nur vom Stapel gelesen, aber nicht wie bei `pop()` entfernt.

Beispiel 3 *In folgendem Programm wird die Funktionsweise eines Stapels in Java demonstriert (→Video K-11.3). Mit Hilfe eines Stapels wird überprüft, ob ein gegebener geklammerter Ausdruck gültig ist oder nicht. Die grundlegende Idee ist, jede öffnende Klammer auf den Stapel zu legen und wenn eine schliessende Klammer*

[3]Ein Stapelspeicher ist mit einem Stapel von Tellern vergleichbar. Es kann immer ein neues Teller oben auf den Stapel gepackt werden oder ein Teller von oben genommen werden.

gelesen wird, den obersten Eintrag vom Stapel zu entfernen – ist der Stapel am Ende leer bzw. wurde nie versucht auf dem leeren Stapel ein Pop auszuführen, ist die Klammerung korrekt (siehe auch Abb. 11.10).

```
import java.util.Scanner;
import java.util.Stack;

public class StackExample {
        /*
         * Liest einen geklammerten Ausdruck ein und überprüft,
         * ob die Klammerung balanciert ist.
         */
        public static void main(String[] args) {
                Scanner scan = new Scanner(System.in);
                System.out.println("Geben Sie einen geklammerten Ausdruck ein:");
                String expression = scan.nextLine();
                System.out.println((balanced(expression) ? "Gültiger" : "Ungültiger") +
                                " Ausdruck");
        }

        /*
         * Gibt true zurück, wenn die Klammern im übergebenen
         * Ausdruck balanciert sind und sonst false.
         */
        private static boolean balanced(String expression) {
                Stack<Character> brackets = new Stack<Character>();
                int index = 0;
                while (index < expression.length()) {
                        if (expression.charAt(index) == '(')
                                brackets.push('(');
                        if (expression.charAt(index) == ')')
                                if (brackets.empty())
                                        return false; // zu viele schliessende Klammern
                                else
                                        brackets.pop();
                        index++;
                }
                return brackets.empty(); // ist der Stack leer, ist der Ausdruck ok
        }
}
```

11.2.3 Schlangen: Die Schnittstellen `Queue` und `Dequeue`

Warteschlangen, oder kurz *Schlangen*, arbeiten genau umgekehrt wie Stapel. In einer Schlange wird jeweils dasjenige Element entfernt, das als erstes zur Sammlung

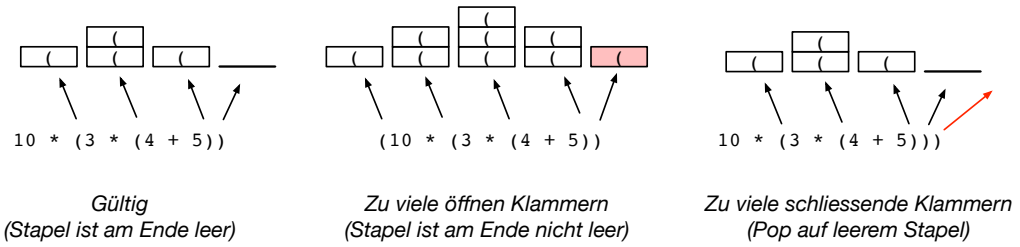

Abbildung 11.10: Verwendung eines Stapels für einen Klammertest.

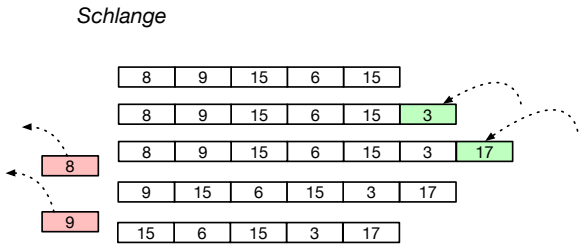

Abbildung 11.11: *Enqueue* und *Dequeue* in einer Schlange illustriert.

hinzugefügt wurde (bekannt als Prinzip *first-in-first-out* (*FIFO*)). Häufig bezeichnet man in Schlangen die Operation zum Hinzufügen eines Objektes als *Enqueue* und das Entfernen als *Dequeue*.

Die FIFO-Eigenschaft bewirkt, dass die Warteschlange wie eine Schlange von Kunden vor einem Schalter arbeitet. Die Schlange hat einen Kopf und ein Ende. Wenn ein Element eingereiht wird, dann nimmt es seinen Platz am Ende der Schlange ein. Das Element, das aus einer Warteschlange entfernt wird, ist immer dasjenige am Kopf der Schlange.

Beispiel 4 *In Abbildung 11.11 ist die Arbeitsweise von Enqueue und Dequeue auf einer Warteschlange illustriert. Zuerst werden nacheinander Enqueue(*3*) und Enqueue(*17*) ausgeführt, danach zwei Mal Dequeue.*

In Java ist das Konzept einer Warteschlange in der Schnittstelle `Queue` definiert. Zu den Klassen, die die Schnittstelle `Queue` implementieren, gehört unter anderem die Klasse `LinkedList`. Folgende Deklaration entspricht also der Definition einer Schlange, welche Objekte vom Typ `T` aufnehmen kann:

```
Queue<T> queue = new LinkedList<T>();
```

Gefüllt wird die Schlange mit der Methode `offer` (entspricht der Funktion *En-*

queue). Für das Erfragen des nächsten Elementes (ohne dieses zu entfernen) gibt es in einer `Queue` die Methode `peek`. Für die Entfernung des nächsten Elementes in der Schlange (Funktion *Dequeue*) ist die Methode `poll` vorgesehen[4].

Die Datenstruktur `Deque` (gesprochen "Deck") kann wie eine "normale" Schlange durch eine `LinkedList` instanziiert werden:

```
Deque<T> deque = new LinkedList<T>();
```

Eine `Deque` bietet Methoden für die FIFO-Verarbeitung an *beiden* Enden der Schlange an. Da es jeweils die Operationen an beiden Enden gibt, erweitert sich die Schnittstelle folgendermassen: Einfügen, Löschen und Entnehmen am Kopf/Ende der Schlange:

- `offerFirst(T t)` und `offerLast(T t)`
- `pollFirst(T t)` und `pollLast(T t)`
- `peekFirst(T t)` und `peekLast(T t)`

11.2.4 Algorithmen im *Collection-Framework*

Das *Collections-Framework* von Java enthält – neben den Sammlungen – mehrere Algorithmen zum Sortieren und Suchen, die einheitlich (polymorph) auf beliebigen Sammlungen arbeiten. Diese Algorithmen sind als statische Methoden in der Klasse `Collections` implementiert:

- `static int binarySearch(List l, Object o)`: Durchsucht die angegebene Liste `l` nach dem angegebenen Objekt (o) unter Verwendung des binären Suchalgorithmus. Die Liste muss vor diesem Aufruf entsprechend der natürlichen Reihenfolge ihrer Elemente aufsteigend sortiert sein.
- `static void copy(List source, List target)`: Kopiert alle Elemente aus der Liste `source` in die Liste `target`.
- `static Object max(Collection c)`: Gibt das maximale Element der gegebenen Sammlung zurück, entsprechend der natürlichen Reihenfolge ihrer Elemente. Alle Elemente in der Sammlung müssen die Schnittstelle `Comparable` implementieren.

[4] Beachten Sie, dass sowohl `peek` als auch `poll` auf leeren Schlangen `null` zurückgeben.

- `static Object min(Collection c)`: Gibt das minimale Element der gegebenen Sammlung zurück, entsprechend der natürlichen Reihenfolge ihrer Elemente. Alle Elemente in der Sammlung müssen die Schnittstelle `Comparable` implementieren.
- `static void reverse(List l)`: Kehrt die Reihenfolge der Elemente in der angegebenen Liste um.
- `static void shuffle(List l)`: Permutiert die angegebene Liste nach dem Zufallsprinzip.
- `static void sort(List l)`: Sortiert die angegebene Liste in aufsteigender Reihenfolge, entsprechend der natürlichen Reihenfolge ihrer Elemente. Alle Elemente in der Liste müssen die Schnittstelle `Comparable` implementieren.

Kapitel 12

Algorithmen und Methoden (Teil 2)

Rekursion ist eine leistungsstarke Programmiertechnik, die elegante Lösungen für bestimmte Probleme ermöglicht. Dieses Kapitel bietet eine Einführung in die rekursive Denkweise und in die Programmierung von rekursiven Algorithmen.

12.1 Rekursive Denkweise

Wir haben in früheren Beispielen schon oft gesehen, dass eine Methode eine andere Methode aufrufen kann, um ein Ziel zu erreichen. Was wir jedoch noch nicht gesehen haben, ist, dass eine Methode sich selbst aufrufen kann. Rekursion ist eine Programmiertechnik, bei der eine Methode sich selbst aufruft, um ihren Zweck zu erfüllen. Doch bevor wir uns mit den Einzelheiten der Verwendung von Rekursion in einem Programm befassen, müssen wir das allgemeine Konzept der Rekursion untersuchen. Die Fähigkeit, rekursiv zu denken, ist eine wesentliche Voraussetzung für die Verwendung der Rekursion als Programmiertechnik.

Im Allgemeinen ist Rekursion der Prozess, bei dem etwas durch sich selbst definiert wird. Nehmen wir zum Beispiel an, wir wollen eine Liste von einer oder mehreren Zahlen, die durch Kommas getrennt sind, formal definieren. Eine solche Liste kann rekursiv entweder als eine Zahl oder als eine Zahl, gefolgt von einem Komma, gefolgt von einer Liste definiert werden. Diese Definition kann wie folgt ausgedrückt werden:

- `Liste` := `Zahl`
  oder
- `Liste` := `Zahl Komma Liste`

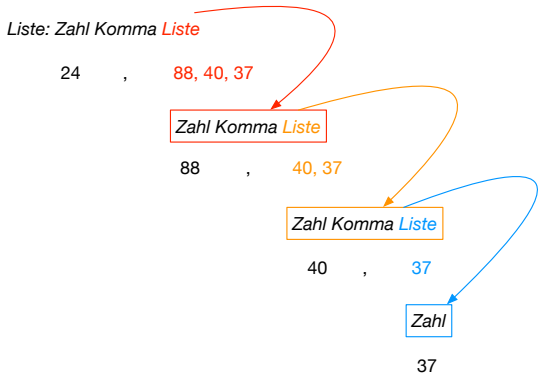

Abbildung 12.1: Rekursive Definition einer Liste illustriert

Diese rekursive Definition von `Liste` definiert jede der folgenden Listen von Zahlen:

- `24, 88, 40, 37`
- `96`
- `14, 64, 21, 69, 32, 93, 47, 81, 28, 45, 81, 52, 69`

Unabhängig davon, wie lang eine Liste ist, wird sie durch die rekursive Definition beschrieben. Eine Liste mit einem Element, wie im zweiten Beispiel, wird vollständig durch den ersten (nicht rekursiven) Teil der Definition beschrieben. Für jede Liste, die länger als ein Element ist, wird der rekursive Teil der Definition (der Teil, der sich auf sich selbst bezieht) so oft wie nötig verwendet, bis das letzte Element erreicht ist. Das letzte Element der Liste wird immer durch den nicht rekursiven Teil der Definition definiert (siehe auch Abb. 12.1).

12.1.1 Unendliche Rekursion

Beachten Sie, dass die Definition von `Liste` eine rekursive und eine nicht rekursive Option enthält. Der Teil der Definition, der nicht rekursiv ist, wird als *Basisfall* oder *Rekursionsanker* bezeichnet. Hätten alle Optionen eine rekursive Komponente, würde die Rekursion niemals enden. Wäre die Definition von `Liste` beispielsweise einfach

- `Liste := Zahl Komma Liste`

könnte eine Liste niemals enden. Dieses Problem wird als *unendliche Rekursion* bezeichnet und ähnelt einer Endlosschleife. Wie beim Problem der Endlosschleife muss eine Programmiererin darauf achten, Algorithmen so zu entwerfen, dass sie eine unendliche Rekursion vermeiden. Jede rekursive Definition muss einen Basisfall

haben, der nicht zu einer rekursiven Option führt. Der Basisfall der Definition von `Liste` ist eine einzelne Zahl, auf die nichts folgt. Mit anderen Worten, wenn die letzte Zahl in der Liste erreicht ist, beendet der Basisfall den rekursiven Pfad.

12.1.2 Rekursion in der Mathematik

Schauen wir uns ein Beispiel für Rekursion in der Mathematik an. Der Wert $N!$ (sprich: *N Fakultät*) ist für jede natürliche Zahl $N \geq 0$ definiert als das Produkt aller natürlichen Zahlen (ohne Null) kleiner oder gleich der Zahl N. Daher ist z.B. 3! definiert als:

$$3! = 3 * 2 * 1 = 6$$

und 5! ist definiert als:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Mathematische Formeln werden oft rekursiv ausgedrückt. Die Definition von $N!$ kann rekursiv ausgedrückt werden als:

- $0! = 1$
- $N! = N * (N - 1)!$ für $N > 0$

Der Basisfall dieser Definition ist 0!, der als 1 definiert ist. Alle anderen Werte von $N!$ (für $N > 0$) werden rekursiv als N mal den Wert $(N - 1)!$ definiert. Die Rekursion besteht darin, dass die Fakultät in Form der Fakultät definiert ist.

Nach dieser Definition ist 50! gleich $50 * 49!$ Und 49! ist gleich $49 * 48!$. Und 48! ist gleich $48*47!$ Dieser Prozess setzt sich fort, bis wir zum Basisfall 0! gelangen (nämlich mit $1! = 1 * 0! = 1 * 1 = 1$). Da die Fakultät nur für ganze Zahlen grösser-gleich 0 definiert ist, ist diese Definition vollständig und endet immer mit dem Basisfall. Der nächste Abschnitt beschreibt, wie die Rekursion in Programmen eingesetzt werden kann.

12.2 Rekursives Programmieren

Wir starten mit einer einfachen mathematischen Operation, um das Konzept der rekursiven Programmierung zu demonstrieren. Betrachten wir den Prozess der Summierung der Werte zwischen `1` und `N` inklusive, wobei `N` eine beliebige positive ganze

Zahl ist. Die Summe der Werte von `1` bis `N` kann rekursiv als `N` plus die Summe der Werte von `1` bis `N-1` ausgedrückt werden.

$$
\begin{aligned}
\sum_{i=1}^{1} i &= 1 \\
\sum_{i=1}^{N} i &= N + \sum_{i=1}^{N-1} i \\
&= N + N - 1 + \sum_{i=1}^{N-2} i \\
&= N + N - 1 + N - 2 + \sum_{i=1}^{N-3} i \\
&= \ldots \\
&= N + N - 1 + N - 2 + \ldots + 2 + \sum_{i=1}^{1} i \\
&= N + N - 1 + N - 2 + \ldots + 2 + 1
\end{aligned}
$$

Zum Beispiel ist die Summe der Werte zwischen `1` und `20` gleich `20` plus der Summe der Werte zwischen `1` und `19`. Wenn man diesen Ansatz weiterverfolgt, ist die Summe der Werte zwischen `1` und `19` gleich `19` plus der Summe der Werte zwischen `1` und `18`. Das klingt vielleicht nach einer seltsamen Art, über dieses Problem nachzudenken, aber es ist ein einfaches Beispiel, mit dem gezeigt werden kann, wie Rekursion programmiert wird.

Wie bereits erwähnt, kann sich in Java, wie in vielen anderen Programmiersprachen auch, eine Methode selbst aufrufen. Jeder Methodenaufruf schafft eine neue Umgebung, in der man arbeiten kann. Das heisst, alle lokalen Variablen und Parameter werden bei jedem Methodenaufruf neu definiert und erhalten ihren eigenen, eindeutigen Datenbereich. Jedes Mal, wenn eine Methode aufgerufen wird, erhält jeder Parameter einen Anfangswert, der auf dem neuen Aufruf basiert. Ist die Methode fertig, kehrt die Verarbeitung zu der Methode zurück, die sie aufgerufen hat (das kann ein früherer Aufruf derselben Methode sein). Diese Regeln unterscheiden sich nicht von denen, die für jeden "normalen" Methodenaufruf gelten.

Eine rekursive Lösung für das Summationsproblem wird durch die folgende rekursive Methode namens `sum` definiert (→Video K-12.1). Beachten Sie, dass diese Metho-

de im Wesentlichen unsere rekursive Definition von oben verkörpert. Die Methode `sum` ist rekursiv, denn `sum` ruft sich selbst auf. Der an `sum` übergebene Parameter wird bei jedem Aufruf von `sum` dekrementiert, bis er den Basisfall `1` erreicht. Rekursive Methoden enthalten immer eine `if-else`-Anweisung, wobei einer der Zweige, normalerweise der erste, den Basisfall darstellt (wie in diesem Beispiel).

```
//Gibt die Summe von 1 bis num zurück
public int sum(int num) {
        int result;
        if (num == 1)
                result = 1;
        else
                result = num + sum(num - 1); // rekursiver Aufruf
        return result;
}
```

Nehmen wir an, die Methode `main` ruft die Methode `sum` auf und übergibt dieser einen Anfangswert von `1`, der dem formalen Parameter `num` zugewiesen wird. Da `num` gleich `1` ist, wird das Ergebnis `1` an `main` zurückgegeben und es findet *keine* Rekursion statt.

Verfolgen wir nun die Ausführung der Methode `sum`, wenn ihr ein Anfangswert von `2` übergeben wird. Da `num` nicht gleich `1` ist, wird `sum` erneut mit einem Argument von `num - 1`, also `1`, aufgerufen. Dies ist ein neuer Aufruf der Methode `sum`, mit einem neuen Parameter `num` und einer neuen lokalen Variablen `result`. Da `num` bei diesem Aufruf gleich `1` ist, wird das Ergebnis von `1` ohne weitere rekursive Aufrufe zurückgegeben. Die Kontrolle geht an die erste Version von `sum` zurück, die aufgerufen wurde. Der Rückgabewert von `1` wird zu dem ursprünglichen Wert von `num` in diesem Aufruf von `sum` addiert, der `2` ist. Daher wird `result` der Wert `3` zugewiesen, der an die Hauptmethode zurückgegeben wird. Die von `main` aufgerufene Methode berechnet also die Summe der ganzen Zahlen von `1` bis `2` korrekt.

Der Basisfall im Summationsbeispiel ist, wenn `N` gleich `1` ist, dann werden keine weiteren rekursiven Aufrufe gemacht. Die Rekursion beginnt, in die früheren Versionen der Summenmethode zurückzufallen, und gibt jedes Mal den entsprechenden Wert zurück. Jeder Rückgabewert trägt zur Berechnung der Summe auf der höheren Ebene bei. Ohne den Basisfall würde es zu einer unendlichen Rekursion kommen. Jeder Aufruf einer Methode erfordert zusätzlichen Speicherplatz und daher führt eine unendliche Rekursion oft zu einem Laufzeitfehler, der anzeigt, dass der Speicher erschöpft ist.

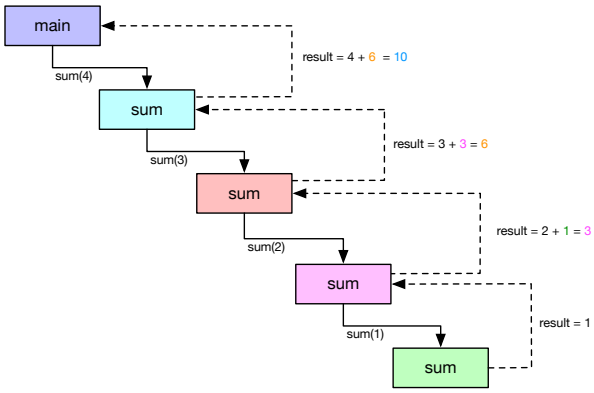

Abbildung 12.2: Rekursive Aufrufe einer Methode illustriert.

Beispiel 114 *Abb. 12.2 veranschaulicht die rekursiven Aufrufe, wenn* `main` *die Funktion* `sum` *aufruft, um die Summe der ganzen Zahlen von 1 bis 4 zu ermitteln. Jedes Kästchen stellt eine Kopie der Methode dar, während sie aufgerufen wird, und zeigt die Zuweisung von Speicherplatz zum Speichern der formalen Parameter und aller lokalen Variablen an. Die Aufrufe sind als durchgezogene Linien dargestellt, die Rückgaben als gepunktete Linien. Das Ergebnis des Rückgabewerts wird bei jedem Schritt angezeigt. Der rekursive Pfad wird vollständig verfolgt, bis der Basisfall erreicht ist; die Aufrufe beginnen dann, ihr Ergebnis über die Kette nach oben zurückzugeben.*

12.2.1 Rekursion vs. Iteration

Natürlich gibt es auch eine nicht-rekursive Lösung für das Summenproblem, das wir gerade untersucht haben. Eine Möglichkeit, die Summe der Zahlen von 1 bis `num` iterativ zu berechnen, ist die folgende:

```
//Gibt die Summe von 1 bis num zurück
public int sum(int num) {
        sum = 0;
        for (int number = 1; number <= num; number++)
                sum += number;
        return sum;
}
```

Diese Lösung ist sicherlich einfacher als die rekursive Version. Wir haben das Summenproblem zur Demonstration der Rekursion verwendet, weil es einfach ist, nicht weil man es unter normalen Bedingungen mit Rekursion lösen würde. Die rekursive Lösung hat den Nachteil mehrerer Methodenaufrufe und bietet in diesem Fall eine kompliziertere Lösung als ihr iteratives Gegenstück.

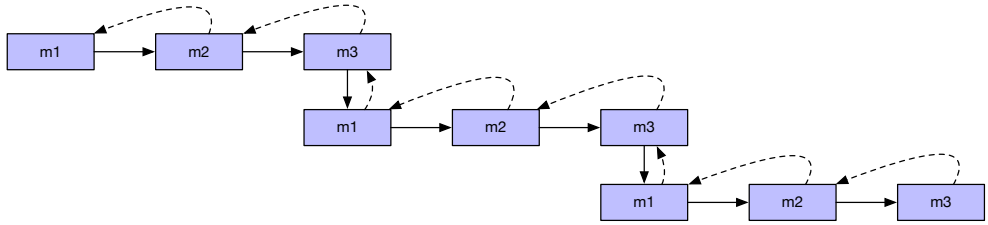

Abbildung 12.3: Indirekte Rekursion illustriert.

Eine Programmiererin muss lernen, wann sie eine Rekursion verwenden sollte und wann nicht. Welcher Ansatz am besten geeignet ist, hängt von dem zu lösenden Problem ab. Alle Probleme können iterativ gelöst werden, aber in manchen Fällen ist die iterative Lösung viel komplizierter. Die Rekursion ermöglicht es uns, für einige Probleme relativ kurze, elegante Programme zu erstellen.

12.2.2 Direkte vs. Indirekte Rekursion

Direkte Rekursion liegt vor, wenn eine Methode sich selbst aufruft, z.B. wenn `sum` `sum` aufruft. *Indirekte Rekursion* liegt vor, wenn eine Methode eine andere Methode aufruft, was schliesslich dazu führt, dass die ursprüngliche Methode erneut aufgerufen wird. Wenn beispielsweise die Methode `m1` die Methode `m2` aufruft und `m2` die Methode `m1` aufruft, kann man sagen, dass `m1` *indirekt rekursiv* ist.

Der Grad der Indirektion kann mehrere Ebenen tief sein, wenn z.B. `m1` `m2` aufruft, welche wiederum `m3` aufruft, die wiederum `m4` aufruft, die wiederum `m1` aufruft. Abb. 12.3 stellt eine Situation mit indirekter Rekursion dar. Die Methodenaufrufe sind mit durchgezogenen Linien dargestellt, die Rückgaben mit gepunkteten Linien. Der gesamte Aufrufpfad wird verfolgt, und dann wird die Rekursion entlang des Rückgabepfads abgewickelt.

Die indirekte Rekursion erfordert die gleiche Aufmerksamkeit für die Basisfälle wie die direkte Rekursion. Ausserdem kann die indirekte Rekursion aufgrund der dazwischen liegenden Methodenaufrufe schwieriger zu verfolgen sein. Daher ist bei der Entwicklung oder Auswertung von indirekt rekursiven Methoden besondere Sorgfalt geboten. Stellen Sie sicher, dass die Umleitung wirklich notwendig ist und in der Dokumentation klar erläutert wird.

12.3 Rekursive Algorithmen

In den folgenden zwei Abschnitten beschreiben wir je ein Problem und eine rekursive Lösung. Für beide Probleme wird untersucht, welche Rolle die Rekursion bei der Lösung spielt und wie ein Basisfall verwendet wird, um die Rekursion zu beenden. Bedenken Sie bei der Betrachtung dieser Beispiele, wie einfach oder kompliziert eine nicht rekursive Lösung für die Probleme wäre.

12.3.1 Rekursiv Sortieren

Wir haben bereits einen Sortieralgorithmus kennengelernt – *Selection-Sort*. Es gibt verschiedene Gründe, einen Sortieralgorithmus einem anderen vorzuziehen, z.B. die Einfachheit des Algorithmus oder seine Effizienz. Ein Algorithmus, der einfacher zu verstehen ist, ist auch einfacher zu implementieren und zu warten. Allerdings sind die einfachsten Algorithmen oft auch die ineffizientesten. Beim Vergleich von Sortieralgorithmen wird in der Regel die Effizienz als Hauptkriterium betrachtet. Im Allgemeinen ist ein Sortieralgorithmus weniger effizient als ein anderer, wenn er mehr Vergleiche durchführt als der andere.

Selection-Sort und viele weitere ähnliche Sortieralgorithmen sind im Wesentlichen gleich effizient. Viele dieser Algorithmen haben eine äussere Schleife und eine innere Schleife mit ähnlichen Eigenschaften (auch wenn die Zwecke der Schleifen unterschiedlich sind). Die äussere Schleife wird für jeden Wert in der Liste einmal ausgeführt, und die innere Schleife vergleicht den Wert in der äusseren Schleife mit den meisten, wenn nicht allen Werten, vom Rest der Liste. Daher führen beide Algorithmen ungefähr n^2 Vergleiche durch, wobei n die Anzahl der Werte in der Liste ist. Wir sagen, dass *Selection-Sort* ein *Algorithmus der Ordnung* n^2 ist.

Es gibt mehrere Algorithmen, die effizienter sind als *Selection-Sort* (und ähnliche Verfahren), diese sind aber auch schwieriger zu verstehen und arbeiten sehr oft rekursiv. Ein solches Verfahren ist *Quick-Sort*. Der Algorithmus *Quick-Sort* hat im schlechtesten Fall zwar auch eine Laufzeit der Ordnung n^2, ist aber im Mittel bemerkenswert effizient: Die erwartete Laufzeit hat die Ordnung $n \log n$.

Bei *Selection-Sort* haben wir den Algorithmus *inkrementell* entworfen: Nachdem wir einen Teil des Arrays sortiert haben, fügen wir das nächste Element an der richtigen Stelle ein. Dies ist nur eine mögliche Entwurfstechnik – Rekursion ist eine weitere Entwurfstechnik.

Rekursive Algorithmen folgen häufig der Methode *Teile-und-Beherrsche*: Sie teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähnlich, jedoch von kleinerer Grösse sind. Sie lösen die Teilprobleme rekursiv und kombinieren die Lösungen dann, um eine Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst drei Schritte auf jeder Rekursionsebene (*Teilen-Beherrschen-Vereinigen*):

1. *Teile* das Problem in Teilprobleme auf, die kleinere Instanzen des ursprünglichen Problems darstellen.
2. *Beherrsche* die Teilprobleme rekursiv (d.h. die Teilprobleme werden weiter aufgeteilt). Wenn die Teilprobleme klein genug sind, dann löse die Teilprobleme auf direktem Weg (die Rekursion stoppt mit dem Basisfall).
3. *Vereinige* die Lösungen der Teilprobleme zur Lösung des ursprünglichen Problems.

Das Sortierverfahren *Quick-Sort* folgt auch dem Paradigma von Teile-und-Beherrsche. Intuitiv arbeitet *Quick-Sort* wie folgt (wir gehen davon aus, dass ein Array A sortiert werden soll):

- Wir zerlegen das Array $A[p..r]$ so in zwei (ggf. leere) Teile $A[p..q-1]$ und $A[q+1..r]$, dass jedes Element von $A[p..q-1]$ kleiner oder gleich $A[q]$ ist, das wiederum kleiner jedem Element von $A[q+1..r]$ ist. Die Berechnung des Index q ist Teil dieses Schrittes[1].
- Wir sortieren die beiden Teilarrays $A[p..q-1]$ und $A[q+1..r]$ durch zwei rekursive Aufrufe von *Quick-Sort*.
- Hat ein Array die Grösse 1, ist dieses sortiert und es wird kein weiterer rekursiver Aufruf gemacht (dies ist der Basisfall).

In Algorithmus 3 ist *Quick-Sort* in Pseudo-Code beschrieben. Um ein Array A zu sortieren, muss der erste Aufruf von Aussen folgendermassen aussehen: *Quick-Sort(A,0, A.length-1)*

[1] Beachten Sie, dass das Element an Position q an der richtigen Stelle liegt. Links und rechts von diesem Element $A[q]$ befinden sich Elemente, die kleiner-gleich bzw. grösser sind als das Element $A[q]$. Die Teilbereiche links und rechts sind aber *nicht* zwingend sortiert.

Algorithm 3 Quick-Sort(A, p, r)

```
if p < r then
    q = Partition(A, p, r)
    Quick-Sort(A, p, q − 1)
    Quick-Sort(A, q + 1, r)
end if
```

Algorithm 4 Partition(A, p, r)

```
x = A[r]
i = p − 1
for j = p . . . r − 1 do
    if A[j] ≤ x then
        i = i + 1
        vertausche A[i] mit A[j]
    end if
end for
vertausche A[i + 1] mit A[r]
return i + 1
```

Der Schlüssel zum Algorithmus ist der Hilfsalgorithmus *Partition*, der das Teilfeld *A[p..r]* umordnet (siehe Algorithmus 4). Algorithmus *Partition* wählt immer das am weitesten rechts stehende Element $x = A[r]$ als sogenanntes *Pivotelement*. Um dieses Pivotelement x herum wird das Array *A[p..r]* in zwei Partitionen zerlegt.

In der ersten Partition befinden sich die Werte, die kleiner-gleich dem Pivotelement x sind. In der zweiten Partition befinden sich die Werte, die grösser sind als das Pivotelement. Index j zeigt auf das aktuell zu partitionierende Element *A[j]*. Der Index i zeigt auf das Ende der ersten Partition. Befindet sich also in *A[j]* ein Wert kleiner-gleich dem Pivotelement, wird i um 1 inkrementiert und das Element *A[j]* wird mit dem Element in *A[i]* vertauscht. Ist der Wert in *A[j]* hingegen grösser als das Pivotelement, passiert nichts – es wird nur j inkrementiert.

Beispiel 5 *Diese beiden Fälle sind in Abb. 12.4 illustriert.*

Am Schluss von *Partition* werden die Elemente an Position *i+1* und r getauscht.

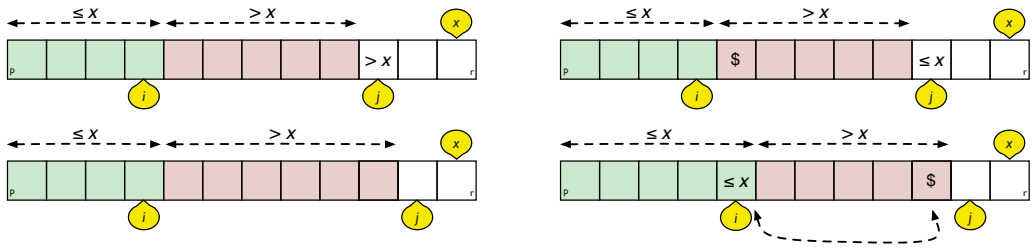

Abbildung 12.4: Die zwei Fälle von *Partition* illustriert: Links: Das Element an Position j ist auf der richtigen Seite – es passiert nichts, ausser dass j um 1 erhöht wird. Rechts: Das Element an Position j ist auf der falschen Seite. In diesem Fall wird i um 1 erhöht und die Elemente *A[i]* und *A[j]* werden getauscht und j um 1 erhöht.

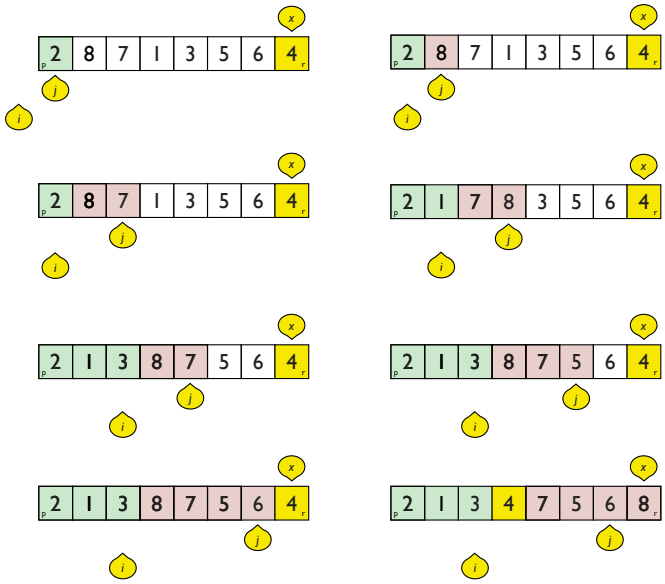

Abbildung 12.5: Die Arbeitsweise von Partition illustriert.

Dies bewirkt, dass das Pivotelement genau zwischen den beiden Partitionen zu liegen kommt. Danach geben wir die Position des Pivotelementes zurück.

Beispiel 6 *In Abb. 12.5 ist die Arbeitsweise von Partition illustriert. In der Abbildung ist das Pivotelement gelb markiert. Grün schattierte Elemente befinden sich alle in der ersten Partition, in der die Werte nicht grösser sind als das Pivotelement* $x = 4$*. Rot schattierte Elemente befinden sich in der zweiten Partition. Die weissen Felder sind noch nicht abgearbeitet.*

Beispiel 7 *In Abbildung 12.6 ist die Arbeitsweise von Quick-Sort basierend auf der Prozedur Partition illustriert.*

In folgender Klasse `Sorting` ist Quick-Sort mit einer statischen Methode `quickSort` implementiert (→Video K-12.2). Beachten Sie, dass wir `quickSort` wiederum generisch implementiert haben. Ein möglicher Aufruf von Aussen wäre bspw. (um eine Array von `Double` Werte zu sortieren):

```
Double[] numbers = {4.6, 3.1, 6.7, 8.9, 10.1, 1.2, 0.4, 4.6};
Sorting.quickSort(numbers, 0, numbers.length - 1);
```

Die beiden Teilaufgaben des Partitionierens und des Tauschens zweier Elemente sind zwei in Hilfsmethoden (`partition` und `swap`) ausgelagert.

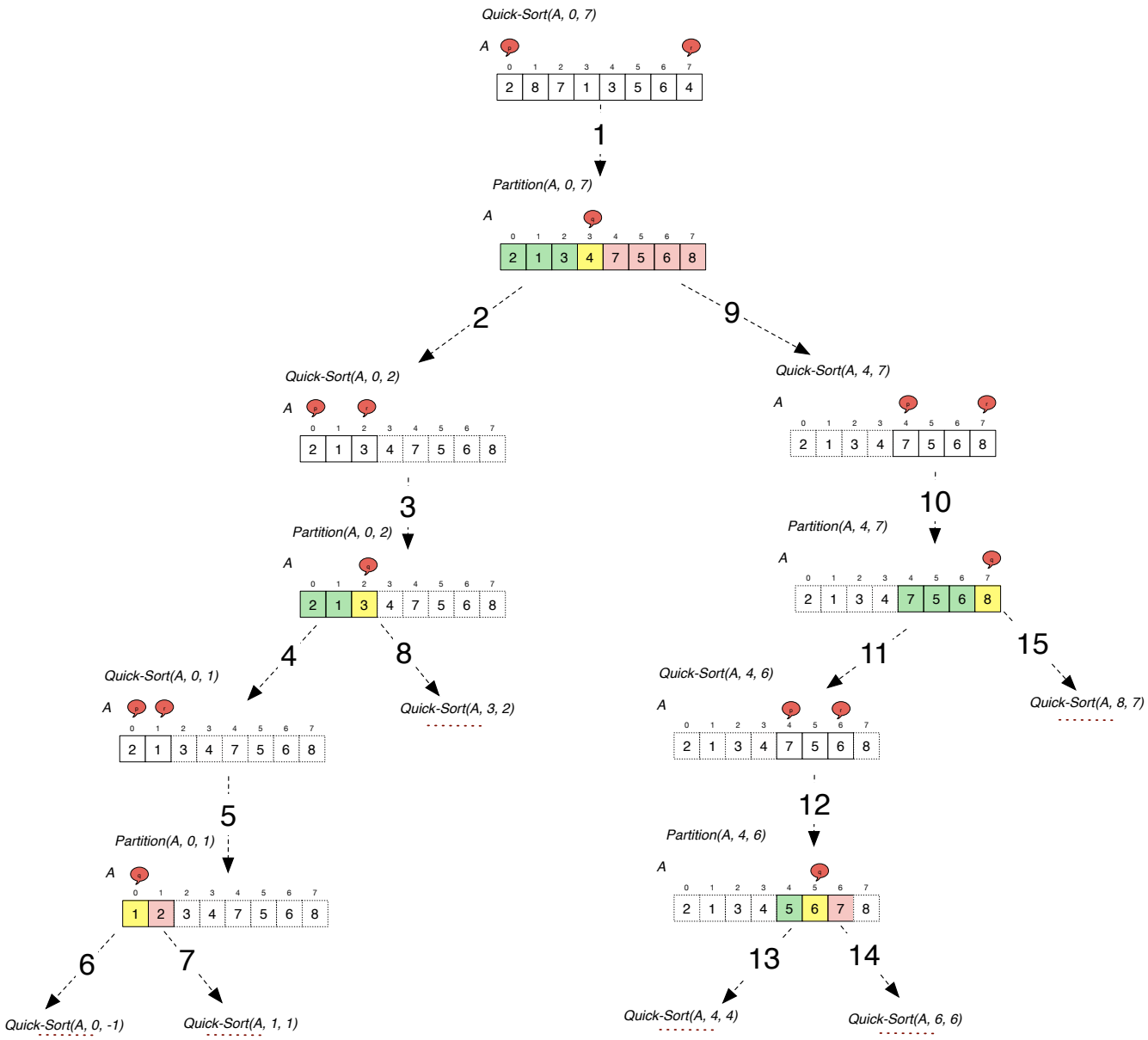

Abbildung 12.6: Die Arbeitsweise von *Quick-Sort* illustriert.

```java
public class Sorting {

	/**
	 * Implementation des Quick Sort
	 */
	public static <T> void quickSort(Comparable<T>[] list, int p, int r) {
		if (p < r) {
			int q = partition(list, p, r);
			quickSort(list, p, q - 1);
			quickSort(list, q + 1, r);
		}
	}

	/*
	 * Hilfsmethode Parition
	 */
	private static <T> int partition(Comparable<T>[] list, int p, int r) {
		Comparable<T> pivot = (Comparable<T>) list[r];
		int i = p - 1;
		for (int j = p; j < r; j++)
			if (list[j].compareTo((T) pivot) < 0){
				i++;
				swap(list, i, j);
			}
		swap(list, i + 1, r);
		return i + 1;
	}

	/*
	 * Hilfsmethode swap
	 */
	private static <T> void swap(Comparable<T>[] list, int i, int j) {
		Comparable<T> temp;
		// Swapping
		temp = list[j];
		list[j] = list[i];
		list[i] = temp;
	}
}
```

Der Vorteil bzgl. Effizienz von *Quick-Sort* gegenüber den einfacheren Sortieralgorithmen ist in Beispielimplementationen bereits in Arrays mit wenigen Tausenden Elementen deutlich bemerkbar.

Häufig wird *Quick-Sort* randomisiert implementiert. Anstatt immer $A[r]$ als Pivotelement zu verwenden, wählen wir hierbei das Pivotelement zufällig aus $A[p..r]$ aus. Das zufällig gewählte Element aus $A[p..r]$ wird dann mit dem Element $A[r]$ vertauscht. Danach kann der Aufruf von Partition(A, p, r) wie bei der obigen Version des Algorithmus erfolgen.

12.3.2 Ein Labyrinth Durchqueren

Das Finden eines Weges durch ein Labyrinth ist mit viel Ausprobieren verbunden: einem Pfad folgen, zurückgehen, wenn man nicht mehr weiterkommt, und andere, noch nicht ausprobierte Pfade versuchen. Solche Vorgänge lassen sich oft gut mit Rekursion lösen.

Das folgende Programm erstellt ein `Maze`-Objekt und versucht, dieses zu durchlaufen (→Video K-12.3). Hierbei wird zunächst das instanziierte Labyrinth ausgegeben und danach erfolgt ein erstes Mal ein Aufruf der rekursiven Methode `traverse` auf Position `(0,0)`. Die Methode gibt einen `boolean` zurück, der besagt, ob das Labyrinth erfolgreich durchquert werden konnte (oder nicht).

```
public class MazeSearch {

        /*
         * Definiert ein neues Labyrinth, gibt dieses aus,
         * löst es und gibt die Lösung aus.
         */
        public static void main(String[] args) {
                Maze labyrinth = new Maze();
                System.out.println(labyrinth);

                if (labyrinth.traverse(0, 0))
                        System.out.println("The maze was successfully traversed!");
                else
                        System.out.println("There is no possible path.");

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

Die nachfolgende Klasse `Maze` verwendet ein zweidimensionales Array `grid` von Ganzzahlen, um das Labyrinth darzustellen (in diesem Beispiel ist das Labyrinth `8` Zeilen mal `13` Spalten gross, obwohl der Code für ein Labyrinth jeder Grösse ausgelegt ist.). Das Ziel ist es, von der linken oberen Ecke (dem Eingangspunkt) zur rechten unteren Ecke (dem Ausgangspunkt) zu gelangen. Zu Beginn bedeutet eine `1`, dass der Weg frei ist, und eine `0`, dass der Weg versperrt ist.

Beim Lösen des Labyrinths werden die Elemente in `grid` in andere Werte umgewandelt, um versuchte Wege und schliesslich einen erfolgreichen Weg durch das Labyrinth anzuzeigen, falls es denn einen gibt (hierzu werden zwei Konstanten `TRIED = 3` und `PATH = 7` deklariert).

```java
public class Maze {
        private final int TRIED = 3;
        private final int PATH = 7;

        private int[][] grid =
                                { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                                  {1,0,1,1,1,0,1,1,1,1,0,0,1},
                                  {0,0,0,0,1,0,1,0,1,0,1,0,0},
                                  {1,1,1,0,1,1,1,0,1,0,1,1,1},
                                  {1,0,1,0,0,0,0,1,1,1,0,0,1},
                                  {1,0,1,1,1,1,1,1,0,1,1,1,1},
                                  {1,0,0,0,0,0,0,0,0,0,0,0,0},
                                  {1,1,1,1,1,1,1,1,1,1,1,1,1} };

        /*
         * Versucht, das Labyrinth rekursiv zu durchqueren. Fügt spezielle
         * Zeichen ein, die auf Orte hinweisen, die ausprobiert wurden und die
         * schliesslich Teil der Lösung werden.
         */
        public boolean traverse(int row, int column) {
                boolean done = false;

                if (this.valid(row, column)) {
                        this.grid[row][column] = this.TRIED; // Zelle wurde besucht
                        if (row == this.grid.length - 1 &&
                                        column == this.grid[0].length - 1)
                                done = true; // Labyrinth ist gelöst
                        else {
                                done = this.traverse(row + 1, column);  // unten
                                if (!done)
                                        done = this.traverse(row, column + 1); // rechts
                                if (!done)
                                        done = this.traverse(row - 1, column); // oben
                                if (!done)
                                        done = this.traverse(row, column - 1); // links
                        }
                        if (done)  // Zelle ist Teil der finalen Lösung
                                this.grid[row][column] = this.PATH;
                }
                return done;
        }

        /**
         * Überprüft, ob die Position gültig ist
         */
        private boolean valid(int row, int column) {
                boolean result = false;
                // Ist Zelle innerhalb der Grenzen?
                if (row >= 0 && row < this.grid.length &&
                                column >= 0 && column < this.grid[row].length)
                        //  Ist Zelle nicht blockiert und noch nicht ausprobiert?
                        if (grid[row][column] == 1)
                                result = true;
                return result;
        }
```

```java
        /**
         * Zeichenkettenrepräsentation des Labyrinths
         */
        public String toString() {
                String result = "\n";

                for (int row = 0; row < this.grid.length; row++) {
                        for (int column=0; column < this.grid[row].length; column++)
                                result += this.grid[row][column] + "";
                        result += "\n";
                }
                return result;
        }
}
```

Die einzigen gültigen Bewegungen durch das Labyrinth sind in den vier Hauptrichtungen: nach unten, nach rechts, nach oben und nach links. Diagonale Bewegungen sind nicht erlaubt.

Lassen Sie uns das Ganze rekursiv durchdenken. Das Labyrinth kann erfolgreich durchlaufen werden, wenn es von der Position `(0, 0)` aus erfolgreich durchlaufen werden kann. Das Labyrinth kann also erfolgreich durchquert werden, wenn es von einer beliebigen Position neben `(0, 0)` erfolgreich durchquert werden kann, nämlich von Position `(1, 0)`, Position `(0, 1)`, Position `(-1, 0)` oder Position `(0, -1)`.

P_{start} beschreibt die Startposition und P_{ziel} die Zielposition in einem Labyrinth. Eine rekursive Lösung für das Problem ein Labyrinth zu durchqueren wäre demnach:

- $\text{Traverse}(P_{\text{start}} \rightarrow P_{\text{ziel}}) :=$ `true`, wenn P_{start} `==` P_{ziel} (Basisfall).
- $\text{Traverse}(P_{\text{start}} \rightarrow P_{\text{ziel}}) := \text{Traverse}(P_{\text{next}} \rightarrow P_{\text{ziel}})$, wobei P_{next} eine der vier Positionen direkt neben P_{start} ist.

Wenn wir einen möglichen nächsten Schritt wählen, sagen wir `(1, 0)`, befinden wir uns in der gleichen Situation wie zuvor. Um das Labyrinth von der neuen aktuellen Position aus erfolgreich zu durchqueren, müssen wir es von einer benachbarten Position aus erfolgreich durchqueren. Zu jedem Zeitpunkt können einige der benachbarten Positionen ungültig sein, blockiert sein oder einen erfolgreichen Weg darstellen. Wir setzen diesen Prozess rekursiv fort. Wenn der Basisfall, die Position `(7, 12)`, erreicht ist, wurde das Labyrinth erfolgreich durchquert.

Die rekursive Methode in der Klasse `Maze` heisst `traverse`. Der erste Aufruf von `traverse` erfolgt an der oberen linken Position `(0, 0)`. Zunächst wird ermittelt, ob

die aktuelle Position durch die angegebene Zeile und Spalte gültig ist (Aufruf der Hilfsmethode `valid`). Eine Position wird als gültig angesehen, wenn sie innerhalb der Grenzen des Labyrinthes bleibt und wenn das Gitter an dieser Stelle eine `1` enthält, was bedeutet, dass eine Bewegung in diese Richtung nicht blockiert ist.

Wenn eine Bewegung auf die aktuelle Position gültig ist, wird der Gittereintrag von einer `1` in eine `3` geändert, wodurch dieser Ort als besucht markiert wird, so dass wir später unsere Schritte nicht zurückverfolgen müssen. Die Methode `traverse` stellt dann fest, ob das Labyrinth durch das Erreichen der Position unten rechts bereits durchquert ist. Es gibt also drei Möglichkeiten für den Basisfall dieses Problems, die einen bestimmten rekursiven Pfad beenden:

- ein ungültiger Zug, weil der Zug ausserhalb der Grenzen liegt
- ein ungültiger Zug, weil der Zug schon einmal versucht wurde
- ein Zug, der an der Endposition ankommt

Wenn die aktuelle Position nicht die untere rechte Ecke ist, suchen wir nach einer Lösung in jeder der Hauptrichtungen, falls erforderlich. Zunächst suchen wir nach unten, indem wir rekursiv die Methode `traverse` aufrufen und die neue Position übergeben. Die Logik der Methode `traverse` beginnt mit dieser neuen Position von vorne. Entweder wird eine Lösung gefunden, indem zuerst versucht wird, von der aktuellen Position aus nach unten zu gehen, oder sie wird nicht gefunden. Wenn sie nicht gefunden wird, versuchen wir, nach rechts zu gehen. Schlägt das fehl, versuchen wir es mit einer Bewegung nach oben. Wenn keine andere Richtung einen korrekten Weg ergeben hat, versuchen wir es schliesslich nach links.

Wenn keine Richtung von der aktuellen Position aus eine korrekte Lösung ergibt, dann gibt es keinen Pfad von dieser Position aus, und die Methode `traverse` gibt `false` zurück.

Wenn von der aktuellen Position aus eine Lösung gefunden wird, wird der Gittereintrag in eine `7` geändert. Die erste `7` wird in der rechten unteren Ecke platziert. Die nächste `7` wird an der Stelle platziert, die zur rechten unteren Ecke geführt hat, und so weiter, bis die letzte `7` in der linken oberen Ecke platziert ist. Wenn also das endgültige Labyrinth ausgedruckt wird, zeigen die Nullen immer noch einen blockierten Weg an, eine `1` zeigt einen offenen Weg an, der nie ausprobiert wurde, eine `3` zeigt einen Weg an, der ausprobiert wurde, aber nicht zur richtigen Lösung führte,

und eine 7 zeigt einen Teil der endgültigen Lösung des Labyrinths an.

Beachten Sie, dass es bei jedem Aufruf der Methode mehrere Möglichkeiten der Rekursion gibt. Je nach der Konfiguration des Labyrinths kann eine oder alle davon befolgt werden. Obwohl es viele Wege durch das Labyrinth geben kann, wird die Rekursion beendet, wenn ein Weg gefunden wurde.

Beispiel 115 *In Abb. 12.7 ist die Arbeitsweise der rekursiven Methode* `traverse` *illustriert. In diesem Beispiel verwenden wir ein kleines Labyrinth der Grösse 5 × 3:*

```
1,1,1,0,1
1,0,1,1,1
0,0,0,0,1
```

Der erste Aufruf von `traverse` *erfolgt mit den Parametern* `(0, 0)`*. Da diese Position gültig ist, wird diese mit dem Wert* `3` *codiert. Da diese Position aber nicht der Endposition entspricht, erfolgt der erste rekursive Aufruf mit* `traverse(1, 0)`*. Diese Position ist ebenfalls gültig und entspricht ebenfalls nicht der Endposition, weshalb wieder ein rekursiver Aufruf erfolgt (dieses mal mit* `traverse(2, 0)`*). Die Position* `(2, 0)` *ist nicht gültig und deshalb gibt der Aufruf von* `traverse(2, 0)` *den Wert* `false` *zurück. Als nächstes wird also* `traverse(1, 1)` *ausprobiert – auch eine ungültige Position. Danach werden noch die Positionen* `(0, 0)` *und* `(1, -1)` *ausprobiert – beides ebenfalls ungültige Positionen – mit denen die Methode* `traverse` *jeweils* `false` *zurückgibt.*

Insgesamt haben wir nun alle Möglichkeiten ausgehend von Position `(1, 0)` *ausprobiert und keine Lösung finden können. Also suchen wir nun Rechts von* `(0, 0)` *bei Position* `(0, 1)` *weiter (eine gültige Position, aber nicht die Endposition). Von hier aus gehen wir nach unten zu Position* `(1, 1)` *– einer ungültigen Position. Also versuchen wir von* `(0, 1)` *ausgehend weiter nach rechts zu gehen und landen bei* `(0, 2)`*. Da wir wieder eine gültige Position gefunden haben, gehen wir nach unten zu Position* `(1, 2)`*, usw.*

Verfolgen Sie sorgfältig den weiteren Ausführung des Codes, während Sie dem Labyrinthfeld folgen, um zu sehen, wie die Rekursion das Problem löst. Überlegen Sie dann, wie schwierig es wäre, eine nicht rekursive Lösung für dieses Problem zu programmieren.

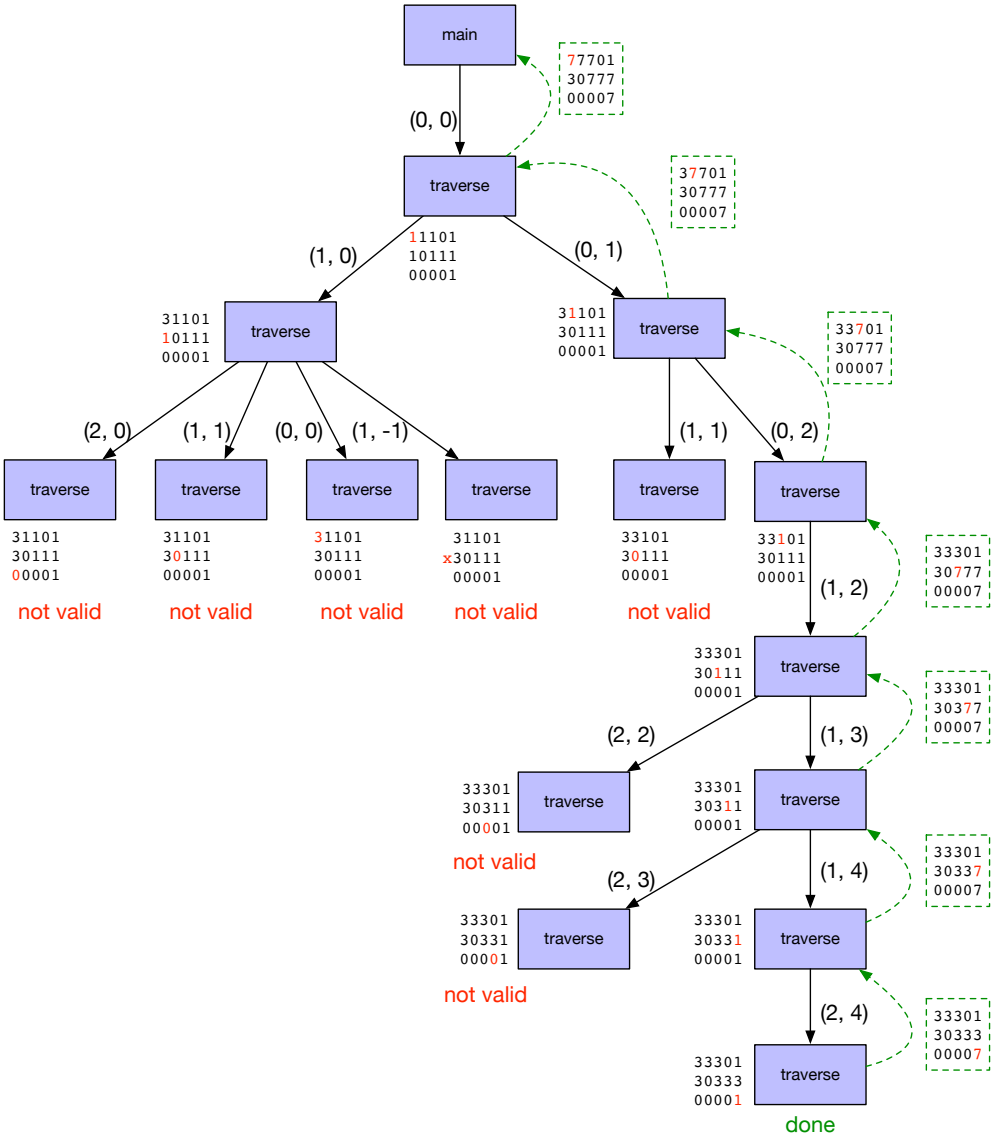

Abbildung 12.7: Die Methode `traverse` illustriert.

Kapitel 13

Laufzeitfehler

In diesem Skript haben wir an einigen Stellen von sogenannten *Laufzeitfehlern* gesprochen. Typische Situationen, welche einen Laufzeitfehler auslösen können, sind beispielsweise:

- Division durch `0`
- Zugriff auf ein Element in einem Array mit ungültigem Index
- "Falsche" Eingabe des Benutzers (z.B. Zeichen statt Zahl)
- Lesen einer Datei, die nicht gefunden werden kann
- Aufruf einer Methode auf einem noch nicht instanziierten Objekt

In Java werden *unerwünschte Situationen* oder *Ausnahmefälle* mit Objekten der Klasse `Exception` (bzw. Subklassen von `Exception`) modelliert. Tritt eine dieser Situationen auf, sagt man, dass ein Objekt vom Typ `Exception` "geworfen" wird (engl. *throw an exception*).

Ein *Fehler* wird in Java ebenfalls als Objekt dargestellt (der Klasse `Error`). Ein solches Objekt repräsentiert aber normalerweise eine nicht behebbare Situation und wird typischerweise nicht abgefangen.

Wir haben grundsätzlich drei Möglichkeiten, mit einem geworfenen Objekt vom Typ `Exception` umzugehen: Wir behandeln die geworfene `Exception` ...

1. ... nicht (wir ignorieren diese also)
2. ... genau da, wo diese aufgetreten ist

3. ... an einer anderen Stelle (werfen die `Exception` also weiter an die aufrufende Methode)

In den nächsten drei Abschnitten besprechen wir diese drei Möglichkeiten.

13.1 Nicht Behandeln einer `Exception`

Wird eine geworfene `Exception` nicht behandelt, wird das Programm normalerweise terminieren und es wird eine Fehlermeldung produziert (in der Standardausgabe – der Konsole). Diese Fehlermeldung beinhaltet den Namen der `Exception`, möglicherweise eine Nachricht, welche den Fehler umschreibt, sowie eine genaue Angabe, wo im Quellcode die `Exception` geworfen wurde.

Beispiel 116 *Die Anweisung* ***`numerator / denominator`*** *in folgendem Code-Fragment wirft eine sogenannte* ***`ArithmeticException`*** *(eine Subklasse von* ***`Exception`****). Diese Klasse repräsentiert die Situation, dass eine ungültige arithmetische Operation durchgeführt wird (in diesem Fall eine Division durch `0`).*

```
int numerator 10;
int denominator = 0;
System.out.println(numerator / denominator);
System.out.println("OK?");
```

Beachten Sie, dass die zweite `println` Anweisung nicht ausgeführt wird, da die geworfene `Exception` nicht behandelt wird. Das heisst, das Programm terminiert zu diesem Zeitpunkt und gibt eine Beschreibung und den Ursprung des Problems aus. In diesem Beispiel wäre dies die folgende Ausgabe:

```
exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionDemo.main(ExceptionDemo.java:11)
```

Die erste Zeile der Fehlerausgabe bei einer geworfenen `Exception` gibt an, welche Subklasse von `Exception` tatsächlich geworfen wurde. Oftmals haben diese Subklassen selbsterklärende Bezeichner, so dass der Grund für das Problem bereits dadurch klar werden kann.

Beispiel 117 *Die drei folgenden Code-Fragmente werfen zum Beispiel jeweils eine*

- ***`ArrayIndexOutOfBoundsException`***
- ***`FileNotFoundException`***

- *NullPointerException*

```
int[] numbers = new int[7];
System.out.println(numbers[7]);
System.out.println("OK?");
```

```
Scanner scan = new Scanner(new File("/diese/Datei/existiert/nicht.txt");
System.out.println("OK?");
```

```
Random rand = null;
System.out.println(rand.nextBoolean());
System.out.println("OK?");
```

Beachten Sie, dass auch in diesen drei Beispielen die zweite `println` Anweisung jeweils nicht ausgeführt wird, da das Programm vorher mit einer Fehlermeldung terminiert.

Möglicherweise wird der Bezeichner der tatsächlich geworfenen `Exception` in der Fehlerausgabe mit einer Nachricht ergänzt, welche das Problem noch etwas genauer beschreibt (in obigem Beispiel der Fehlerausgabe entspricht bspw. `/ by zero` dieser Nachricht).

Die restlichen Zeilen der Fehlermeldung entsprechen der sogenannten *Stapelverfolgung von Aufrufen* (engl. *call stack trace*). Diese gibt an, *wo* im Quellcode die `Exception` geworfen wurde. In unserem einfachen Beispiel umfasst die Stapelverfolgung nur gerade eine Zeile:

```
at ExceptionDemo.main(ExceptionDemo.java:11)
```

Das bedeutet, dass der Fehler auf Zeile `11` in der Methode `main` in der Klasse `ExceptionDemo` geworfen wurde. Im Allgemeinen kann die Stapelverfolgung mehrere Zeilen umfassen. Diese Zeilen geben an, welche Methoden in welcher Reihenfolge aufgerufen wurden, bis schliesslich der Methodenaufruf erfolgte, der zur `Exception` führte. In obigem Beispiel wurde offensichtlich nur eine Methode aufgerufen (nämlich `main` und diese hat auch gleich die `Exception` geworfen).

Um herauszufinden, wo der Ursprung des Problems liegt, kann man die Stapelverfolgung entweder "von oben nach unten" oder "von unten nach oben" lesen:

Beispiel 118 *Folgende Stapelverfolgung gibt an, dass in der Methode* `main` *(in der Klasse* `ExceptionDemo`*) auf Zeile* `17` *die Methode* `fooMethod` *(aus der Klasse* `Foo`*)*

aufgerufen wurde. In dieser Methode wiederum wurde auf Zeile `8` `barMethod` *(aus der Klasse* `Bar`*) aufgerufen – und dort wurde die* `Exception` *geworfen (und zwar auf Zeile* `10`*).*

```
at Bar.barMethod(Bar.java:10)
at Foo.fooMethod(Foo.java:8)
at ExceptionDemo.main(ExceptionDemo.java:17)
```

13.2 Die `try`-`catch` Anweisung

Im Folgenden wollen wir eine geworfene `Exception` mit einer `try`-`catch` Anweisung auffangen und behandeln – und zwar genau dort, wo diese auch geworfen wird.

Die `try`-`catch` Anweisung besteht aus drei Blöcken: Dem `try` Block, dem `catch` Block und dem optionalen `finally` Block (alles reservierte Wörter). Im `try` Block programmieren wir die kritischen Anweisungen. Das sind also Anweisungen, von denen die Programmiererin weiss, dass diese möglicherweise eine `Exception` werfen werden. Im `catch` Block werden dann die Anweisungen programmiert, die ausgeführt werden sollen, sobald eine bestimmte `Exception` tatsächlich geworfen wird.

Wird eine `try`-`catch` Anweisung ausgeführt, so werden zuerst die Anweisungen im `try` Block ausgeführt. Wird dabei *keine* `Exception` geworfen, fährt der Kontrollfluss unterhalb der gesamten `try`-`catch` Anweisung weiter (d.h. der `catch`-Block wird einfach übersprungen). Diese Situation entspricht dem normalen Programmablauf.

Wird aber während der Abarbeitung der Anweisungen im `try` Block eine `Exception` geworfen, wird der Kontrollfluss direkt an den entsprechenden `catch` Block weitergeleitet (wenn dieser vorhanden ist). Das heisst, der Kontrollfluss geht an genau den `catch` Block über, dessen `Exception` Klasse mit der geworfenen `Exception` übereinstimmt. Nach der Ausführung der Anweisungen im `catch` Block fährt der Kontrollfluss unterhalb der gesamten `try`-`catch` Anweisung weiter – das heisst also, das Programm terminiert nicht, sondern läuft nach dem `catch` normal weiter.

Der optionale `finally` Block in einer `try`-`catch` Anweisung wird ausgeführt, ungeachtet wie der `try` Block verlassen wird. Der `finally` Block wird verwendet, um sicherzustellen, dass bestimmte Teile einer Methode in *jedem* Fall ausgeführt werden (d.h. im Normalfall, in erwarteten und unerwarteten Fehlerfällen).

Beispiel 119 *In folgendem Programm ist in der Methode* `main` *eine* `try-catch` *Anweisung gezeigt (inklusive* `finally` *Block) (→Video K-13.1).*

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class IntegerDivision {

        /**
         * Demonstriert eine try-catch Anweisung (inkl. einem finally Block)
         */
        public static void main(String[] args) {

                Scanner scan = new Scanner(System.in);

                try {
                        System.out.println("Geben Sie einen Zähler ein");
                        int numerator = scan.nextInt();
                        System.out.println("Geben Sie einen Nenner ein");
                        int denominator = scan.nextInt();
                        System.out.println("Resultat der ganzzahligen Division in Java: "
                                        + numerator / denominator );
                }
                catch (InputMismatchException exception){
                        System.out.println("Das war keine gültige Eingabe!");
                }
                finally {
                        System.out.println("Das wird in *jedem* Fall ausgeführt...");
                }
                System.out.println("Das Programm läuft hier weiter...");
        }
}
```

Wird dieses Programm mit einer korrekten Eingabe ausgeführt, wird folgendes ausgegeben:

```
Geben Sie einen Zähler ein
7
Geben Sie einen Nenner ein
2
Resultat der ganzzahligen Division in Java: 3
Das wird in *jedem* Fall ausgeführt...
Das Programm läuft hier weiter...
```

Beachten Sie, dass der `catch` *Block übersprungen wird, da im* `try` *Block keine* `Exception` *geworfen wird. Der* `finally` *Block wird aber durchgeführt und danach läuft das Programm normal weiter.*

Gibt der Benutzer zur Laufzeit des Programmes statt einer ganzen Zahl eine andere Zeichenfolge ein (z.B. `abc`), generiert das Programm folgende Ausgabe:

```
Geben Sie einen Zähler ein
abc
Das war keine gültige Eingabe!
Das wird in *jedem* Fall ausgeführt...
Das Programm läuft hier weiter...
```

Es wird die `Exception` geworfen – um genau zu sein, eine `InputMismatchException`. Für diese `Exception` haben wir einen entsprechenden `catch` Block programmiert:

```
catch (InputMismatchException exception)
```

Die geworfene `InputMismatchException` wird also in diesem Block aufgefangen. In diesem Beispiel wird hier einfach eine `println` Anweisung durchgeführt (`Das war keine gültige Eingabe!`). Der `finally` Block wird auch in diesem Fall durchgeführt und das Programm läuft danach normal weiter.

Um den Nutzen des `finally` Blockes zu verdeutlichen, geben wir im nächsten Durchlauf des Programmes `IntegerDivision` eine `0` als Nenner ein. Dies führt zu einer `ArithmeticException` (Division durch `0`). Für diese `Exception` haben wir keinen `catch` Block programmiert – dies entspricht also einem von uns nicht erwarteten Fehler und die geworfene `Exception` wird dementsprechend nicht aufgefangen.

Bevor das Programm nun mit einer Fehlermeldung terminiert, werden in diesem Fall aber noch sämtliche Anweisungen im `finally` Block ausgeführt:

```
Geben Sie einen Zähler ein
7
Geben Sie einen Nenner ein
0
Das wird in *jedem* Fall ausgeführt...
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ch13.IntegerDivision.main(IntegerDivision.java:19)
```

Ein `try` Block kann mehrere `catch` Blöcke besitzen. Das heisst, im obigen Beispiel könnten wir einen weiteren Block erstellen:

```
catch (ArithmeticException exception){
        System.out.println("Division durch 0!");
}
```

In diesem Block behandeln wir die Situation, dass der Benutzer für den Nenner eine `0` eingegeben hat.

Beachten Sie, dass der Kontrollfluss nach dem Auffangen einer `Exception` *unterhalb* der `try-catch` Anweisung fortfährt – für unser Beispiel von oben heisst das, dass die Benutzerinteraktion im `try` Block nach dem Werfen der `Exception` *nicht* wiederholt wird. Indem wir die `try-catch` Anweisung innerhalb einer Schleife platzieren, könnten wir dieses Ziel erreichen.

Beispiel 120 *In folgendem Programm ist dieses Prinzip veranschaulicht (→ Video K-13.2). Die Variable* `inputOK` *wird nur dann* `true`*, wenn die letzte Zeile des* `try` *Blockes tatsächlich erreicht wird – in diesem Fall wurde also keine* `Exception` *geworfen und die Schleife kann verlassen werden. Im Fall einer geworfenen* `Exception` *wird die gesamte* `try-catch` *Anweisung wiederholt.*

```
import java.util.Scanner;

public class IntegerInput {

        /**
         * Demonstriert das Erwzingen einer gültigen Eingabe
         */
        public static void main(String[] args) {

                Scanner scan = new Scanner(System.in);
                boolean inputOK = false;
                int number = 0;
                do {
                        try {
                                System.out.println("Geben Sie eine ganze Zahl ein:");
                                number = Integer.parseInt(scan.nextLine());
                                inputOK = true;
                        }
                        catch (NumberFormatException exception){
                                System.out.println("Das war keine gültige Eingabe!");
                        }

                } while(!inputOK);
                System.out.println("Ihre Eingabe: " + number);
        }
}
```

Beispiel 121 *Nehmen wir an, dass wir ein Programm zum Einlesen von Codes programmieren wollen (→Video K-13.3). Ein gültiger Code bestehe dabei aus vier Ziffern. Das folgende Programm liest Eingaben vom Benutzer ein, zählt die gültigen Eingaben und gibt am Schluss alle gültigen Codes aus.*

```
import java.util.ArrayList;
import java.util.Scanner;

public class CodeReader {

        /**
         * Liest Codes ein und zählt die Anzahl gültigen Eingaben
         */
        public static void main(String[] args) {

                System.out.println("Code-Eingabe");
                System.out.println("Aus den ersten vier Ziffern Ihrer "
                                + "Eingabe wird jeweils der Code erstellt\n");

                // Liste mit gültigen Codes
                ArrayList<Integer> validCodes = new ArrayList<Integer>();

                Scanner scan = new Scanner(System.in);
                System.out.print("Mindestens vier Ziffern eingeben - q  zum Beenden: ");
                String code = scan.nextLine();

                while (!code.equalsIgnoreCase("Q")) {
                        try {
                                int number = Integer.parseInt(code.substring(0, 4));
                                validCodes.add(number);
                        }
                        catch(StringIndexOutOfBoundsException exception) {
                                System.out.println("Code ist zu kurz!");
                        }
                        catch (NumberFormatException exception) {
                                System.out.println("Code darf nur Ziffern enthalten!");
                        }
                        System.out.print("Mindestens vier Ziffern eingeben "
                                        + "- q  zum Beenden: ");
                        code = scan.nextLine();
                }

                System.out.println();
                System.out.println("Anzahl gültige Codes: " + validCodes.size());
                System.out.println();
                System.out.println("Gültige Eingaben: " + validCodes);

        }
}
```

Die Anweisungen im `try` *Block versuchen die Zeichen an Position* `0` *bis* `3` *aus der Benutzereingabe* `code` *zu lesen (mit Hilfe der Methode* `substring`*). Danach wird aus dieser Zeichenkette eine ganze Zahl ausgelesen (mit Hilfe der Methode* `parseInt`*).*

Gibt es bei diesen Anweisungen Probleme, wird eine `Exception` *geworfen.*

Zum Beispiel wird eine `StringIndexOutOfBoundsException` *durch die Methode* `substring` *geworfen, wenn der eingegebene Code zu kurz ist. Die Methode* `parseInt` *wird eine* `NumberFormatException` *werfen, falls sich an einer der Positionen* `0` *bis* `3` *keine Ziffer befinden sollte. Für beide Fälle ist ein entsprechender* `catch` *Block programmiert und in beiden Fällen wird eine entsprechende Nachricht ausgegeben. Das Programm läuft danach aber normal weiter.*

Beachten Sie, dass die Zahl `number` *nur dann zur Liste* `validCodes` *hinzugefügt wird, wenn keine* `Exception` *geworfen wird. Im Fall, dass eine der beiden* `Exception` *geworfen wird, springt der Kontrollfluss unmittelbar in den entsprechenden* `catch` *Block und die restlichen Anweisungen im* `try` *Block werden übersprungen. Eine mögliche Ein- und Ausgabe wäre z.B.:*

```
Code-Eingabe
Aus den ersten vier Ziffern Ihrer Eingabe wird jeweils der Code erstellt

Mindestens vier Ziffern eingeben - q zum Beenden: 1979
Mindestens vier Ziffern eingeben - q zum Beenden: 123
Code ist zu kurz!
Mindestens vier Ziffern eingeben - q zum Beenden: 12345
Mindestens vier Ziffern eingeben - q zum Beenden: A123
Code darf nur Ziffern enthalten!
Mindestens vier Ziffern eingeben - q zum Beenden: 2021
Mindestens vier Ziffern eingeben - q zum Beenden: q

Anzahl gültige Codes: 3

Gültige Eingaben: [1979, 1234, 2021]
```

Jede Subklasse von `Exception` stellt (via Vererbung) zwei Methoden zur Verfügung:

- `String getMessage()`: Gibt die gespeicherte Fehlermeldung im entsprechenden `Exception` Objekt zurück.
- `void printStackTrace()`: Gibt die Stapelverfolgung als Fehlermeldung auf der Konsole aus.

Beispiel 122 *In folgendem Code-Fragment rufen wir die beiden Methoden* `getMessage` *und* `printStackTrace` *im* `catch` *Block auf (auf dem aufgefangenen Objekt* `exception`*):*

```
try {
        // macht etwas
} catch (Exception exception) {
        // gibt die in exception gespeicherte Fehlermeldung aus
        System.out.println(exception.getMessage());
        // gibt die Stapelverfolgung aus
        exception.printStackTrace();
}
```

13.3 Weitergeben einer `Exception`

Wird eine `Exception` in einer Methode nicht dort aufgefangen und behandelt, wo diese auch geworfen wird, so wird der Kontrollfluss unmittelbar an die aufrufende Methode zurückgegeben, sobald die `Exception` geworfen wird. Wir können unseren Quellcode so konzipieren, dass die `Exception` dann dort aufgefangen und behandelt wird (innerhalb einer `try-catch` Anweisung). Wird die geworfene `Exception` aber auch dort nicht aufgefangen, so wird diese erneut weitergegeben (wiederum an die aufrufende Methode).

Diese Weitergabe einer geworfenen `Exception` wird solange fortgesetzt, bis die geworfene `Exception` entweder aufgefangen und behandelt wird oder letztlich die `main` Methode erreicht wird. Falls in `main` die `Exception` ebenfalls nicht behandelt wird (d.h. auch `main` gibt die `Exception` weiter), terminiert das Programm nicht ordnungsgemäss und es wird eine entsprechende Fehlermeldung ausgegeben.

Damit eine Methode eine geworfene `Exception` weitergeben kann, muss deren Methodenkopf mit einem `throws` (ein reserviertes Wort) und dem Bezeichner der genauen `Exception`, die möglicherweise geworfen wird, erweitert werden[1]. Folgender Methodenkopf gibt zum Beispiel an, dass die Methode `method` möglicherweise eine `NumberFormatException` werfen wird:

```
public void method() throws NumberFormatException
```

[1]Dies ist technisch gesehen nicht 100% korrekt: Man unterscheidet zwischen sogenannt *überprüften* und *unüberprüften* `Exception`. Eine unüberprüfte `Exception` erfordert kein `throws` im Methodenkopf. Die Klasse `NumberFormatException` ist zum Beispiel eine unüberprüfte `Exception`. Es ist aber auch nicht verboten, im Methodenkopf auch für unüberprüfte `Exception` jeweils ein `throws` zu definieren.

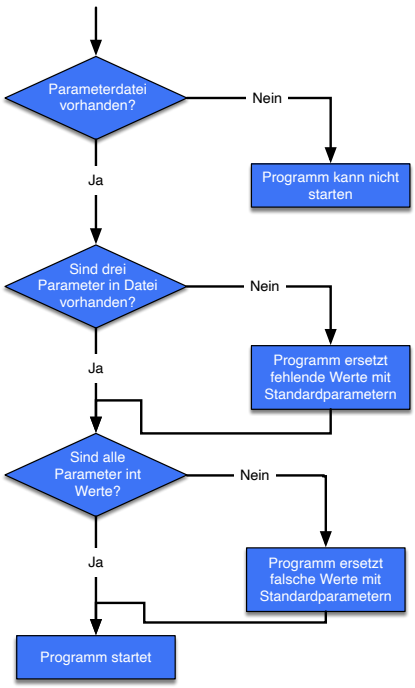

Abbildung 13.1: Je nach Fehler folgen wir einem anderen Pfad.

Eine von der Methode `method` geworfene `NumberFormatException` kann nun ausserhalb von `method` aufgefangen werden. Hierzu wird der Aufruf der Methode `method` innerhalb einer `try-catch` Anweisung platziert:

```
try {
        this.method();
} catch(NumberFormatException exception) {
        // Behandeln der Exception
}
```

Alternativ kann der Methodenkopf der aufrufenden Methode ebenfalls mit einem `throws` erweitert werden, so dass diese die `Exception` auch weitergeben kann. Mit dieser Technik können wir für unterschiedliche `Exception` Typen unterschiedliche Stellen in unserem Quellcode definieren, wo diese aufgefangen und behandelt werden sollen.

Beispiel 123 *Wir programmieren ein Gerüst eines Programmes mit der Logik aus Abb. 13.1 (→ Video K-13.4).*

In der Klasse ***`FileReader`*** *wirft die Methode* ***`readFile`*** *möglicherweise eine sogenannte* ***`FileNotFoundException`*** *während die Methode* ***`extractParameters`*** *möglicher-*

weise eine **`NoSuchElementException`** *werfen wird (beides ist in den Methodenköpfen hinter dem* **`throws`** *ersichtlich):*

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class FileReader {

        /**
         * Liest Parameter aus einer Datei
         */
        public static void readFile(String url) throws FileNotFoundException {
                Scanner fileScanner = new Scanner(new File(url));
                String line = fileScanner.nextLine();
                try {
                        extractParameters(line);
                }
                catch (NoSuchElementException exception) {
                        System.out.println("WARNUNG: Zu wenige Parameter in der Datei.");
                        System.out.println("Es werden z.T. Standardparameter verwendet!");
                }
        }

        /**
         * Extrahiert drei int Werte aus der Zeichenkette line
         */
        private static void extractParameters(String line)
                        throws NoSuchElementException {
                Scanner lineScanner = new Scanner(line);
                for (int i = 0; i < Main.parameters.length; i++) {
                        String next = lineScanner.next();
                        try {
                                Main.parameters[i] = Integer.parseInt(next);
                        }
                        catch (NumberFormatException exception) {
                                System.out.println("WARNUNG: Parameter " + next +
                                        " an Position " + i + " ist keine ganze Zahl");
                                System.out.println("Es werden z.T. Standardparameter " +
                                                "verwendet!");
                        }
                }
        }
}
```

Die statische Methode **`readFile`** *instanziiert zunächst ein Objekt vom Typ* **`Scanner`**, *das aus einer Datei lesen kann:*

```
Scanner fileScanner = new Scanner(new File(url));
```

Dieser Aufruf kann eine **`FileNotFoundException`** *werfen, da die Datei möglicherweise*

nicht gefunden werden kann. In diesem Fall wird die geworfene `FileNotFoundException` *an die aufrufende Methode weitergegeben.*

Die Methode `main` *in der Klasse* `Main` *ruft die Methode* `readFile` *auf. Der Aufruf von* `readFile` *ist innerhalb eines* `try` *Blockes platziert und für die mögliche* `FileNotFoundException` *ist ein* `catch` *Block vorhanden. Falls die Datei* `parameters.txt` *nicht im Ordner* `src/resources/` *gefunden werden kann, wird die von* `readFile` *erhaltene* `Exception` *also hier aufgefangen (mit einer Ausgabe, dass das Programm nicht starten kann). Beachten Sie, dass die restlichen Anweisungen im* `try` *Block nicht ausgeführt werden.*

```
public class Main {

        public static final String url = "src/resources/parameters.txt";
        public static int[] parameters = {0, 0, 0};

        /**
         * Demonstriert das Weitergeben einer Exception
         * @throws FileNotFoundException
         */
        public static void main(String[] args) {

                try {
                        FileReader.readFile(url);
                        System.out.println("Das Programm startet mit den Parametern:");
                        for (int p : parameters)
                                System.out.println(p);
                } catch (FileNotFoundException exception) {
                        System.out.println("Parameterdatei kann nicht geladen werden");
                        System.out.println("Das Programm kann nicht starten");
                }
                System.out.println("Das Programm wird beendet...");
        }
}
```

Dies führt zu folgender Ausgabe:

```
Parameterdatei kann nicht geladen werden
Das Programm kann nicht starten
Das Programm wird beendet...
```

Nachdem der `catch` *Block ausgeführt ist, wird das Programm beendet. Das bedeutet insbesondere, dass keine weiteren Anweisungen der Methode* `readFile` *ausgeführt werden – das Lesen der Datei ging schief und der Kontrollfluss hat mit der geworfenen* `Exception` *die Methode* `readFile` *unmittelbar verlassen.*

Angenommen die Datei `parameters` *existiert und enthält die drei Zahlen* `17 3 5` *auf einer Zeile. In diesem Fall wird diese Zeile in der Methode* `readFile` *eingelesen und der Variablen* `line` *zugewiesen. Danach erfolgt innerhalb eines* `try` *Blockes der Aufruf der Methode* `extractParameters`*.*

In dieser Methode werden mit Hilfe eines weiteren `Scanner` *Objektes und einer* `for`*-Schleife die drei Zahlen aus* `line` *ausgelesen und im statischen* `int[]` `parameters` *gespeichert. Da alles geklappt hat, springt der Kontrollfluss zurück in die Methode* `readFile`*, in der der* `catch` *Block übersprungen wird.*

Zurück in `main` *werden die geladenen Parameter ausgegeben:*

```
Das Programm startet mit den Parametern:
17
3
5
Das Programm wird beendet...
```

In der Methode `extractParameters` *können unterschiedliche Dinge schief gehen. Befinden sich zum Beispiel weniger als* `3` *Elemente in* `line`*, wird die Methode* `next` *eine* `NoSuchElementException` *werfen. In diesem Fall wird das Einlesen der Parameter direkt unterbrochen und die* `Exception` *an die aufrufende Methode – also* `readFile` *– weitergegeben. Die Methode* `readFile` *fängt die geworfene* `NoSuchElementException` *in einem* `catch` *Block auf und generiert eine entsprechende Warnung.*

Danach kehrt der Kontrollfluss zurück in die Methode `main` *und dort werden die bis zum Werfen der* `Exception` *geladenen Parameter ausgegeben. Enthält die Datei* `parameters` *zum Beispiel die zwei Zahlen* `17 3`*, lautet die Ausgabe:*

```
WARNUNG: Zu wenige Parameter in der Datei.
Es werden z.T. Standardparameter verwendet!
Das Programm startet mit den Parametern:
17
3
0
```

```
Das Programm wird beendet...
```

Ist eines der drei Elemente in `line` *nicht eine ganze Zahl (z.B.* `17 3.1 5`*), wird die Methode* `parseInt` *eine* `NumberFormatException` *werfen. Diese* `Exception` *wird von* `extractParameters` *nicht weitergegeben, sondern direkt aufgefangen (im entsprechenden* `catch` *wird eine Warnung ausgegeben). Die* `for`*-Schleife fährt danach aber normal weiter, da wir die* `Exception` *ja nicht weitergegeben haben. Die Ausgabe lautet also zum Beispiel:*

```
WARNUNG: Parameter 3.1 an Position 1 ist keine ganze Zahl
Es werden z.T. Standardparameter verwendet!
Das Programm startet mit den Parametern:
17
0
5
Das Programm wird beendet...
```

13.4 Eine Eigene `Exception` Programmieren und Werfen

Die `Exception` Klassen in der Java-API sind durch Vererbung miteinander verwandt und bilden eine Klassenhierarchie (siehe Abb. 13.2). All diese Klassen sind direkte oder indirekte Subklassen der Klasse `Throwable`. Wir können eigene `Exception` Klassen definieren, indem wir von der Klasse `Exception` (oder einer Subklasse davon) erben. Welche Elternklasse verwendet wird, hängt davon ab, wie die neue `Exception` verwendet werden soll.

Eine `Exception` wird entweder geprüft oder nicht geprüft (engl. *checked* vs. *unchecked*). Eine geprüfte `Exception` muss entweder innerhalb einer `try-catch` Klausel abgefangen werden oder muss in der `throws`-Klausel jeder Methode aufgeführt werden, die sie auslösen und weitergeben können (der Kompilierer wird einen Fehler generieren, wenn dies für eine geprüfte `Exception` nicht geschieht).

Eine ungeprüfte `Exception` erfordert keine explizite Behandlung, obwohl sie auch auf diese Weise verarbeitet werden kann. In Java sind die einzigen ungeprüften `Exception` Objekte vom Typ `RuntimeException` oder einer ihrer Subklassen.

Beispiel 124 *In folgender Klasse haben wir die Klasse* `EmptyInputException` *aus der Klasse* `Exception` *abgeleitet. Sehr oft sind eigene* `Exception` *Klassen nicht*

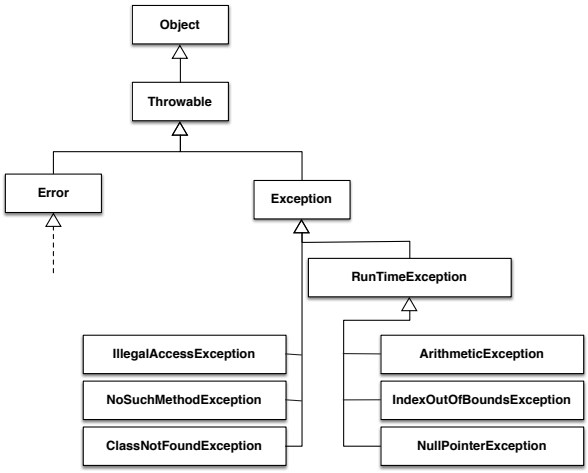

Abbildung 13.2: Die Klassenhierarchie für `Exception`.

komplexer, als die in diesem Beispiel gezeigte: Eine direkte Erweiterung der existierenden Klasse ***`Exception`****, welche eine bestimmte Fehlermeldung speichert (z.B. eine Beschreibung der fehlerhaften Situation, auf die wir dann später mit der Methode* ***`getMessage`*** *zugreifen können).*

```
public class EmptyInputException extends Exception {

        /*
         * Konstruktor für die eigene Exception gibt eine
         * Fehlermeldung an den Konstruktor der Superklasse weiter.
         */
        public EmptyInputException(String missing) {
                super("Keine Eingabe für Feld " + missing + " gemacht.");
        }

}
```

In diesem Fall deklarieren wir den Konstruktor ***`EmptyInputException`*** *so, dass dieser eine Zeichenkette* ***`missing`*** *als Parameter erwartet. Diese Zeichenkette wird via* ***`super`*** *an den Konstruktor der Superklasse weitergegeben. Die Superklasse* ***`Exception`*** *speichert dann diese Fehlermeldung.*

Haben wir eine eigene `Exception` programmiert, können wir diese wenn nötig instanziieren und werfen. Hierzu verwenden wir das Schlüsselwort `throw`:

```
throw new EmptyInputException("Feldname");
```

Beispiel 125 *In folgendem Programm wird in der Methode* ***`readInputFromUser`*** *der Name und das Alter des Benutzers eingelesen (→ Video K-13.5). Sollte eine der*

beiden Eingaben nicht gemacht werden, wird eine **`EmptyInputException`** *geworfen (beachten Sie, dass wir dem Konstruktor den Namen des fehlenden Feldes mitgeben.*

```
import java.util.Scanner;

public class OwnException {

        /**
         * Liest Namen und Alter des Benutzers ein.
         */
        public static void main(String[] args) {
                try {
                        readInputFromUser();
                } catch (EmptyInputException e) {
                        System.out.println("Eingabe unvollständig!");
                        System.out.println(e.getMessage());
                }
        }

        /**
         * Liest Namen und Alter des Benutzers ein. Wirft eine
         * Exception, falls eine der beiden Angaben fehlt.
         */
        private static void readInputFromUser() throws EmptyInputException {
                Scanner scan = new Scanner(System.in);
                System.out.print("Geben Sie Ihren Namen ein: ");
                String name = scan.nextLine();
                System.out.print("Geben Sie Ihr Alter ein: ");
                String age = scan.nextLine();
                if (name.length() == 0)
                        throw new EmptyInputException("Name");
                if (age.length() == 0)
                        throw new EmptyInputException("Alter");
        }
}
```

Eine mögliche Ein- und Ausgabe wäre also:

```
Geben Sie Ihren Namen ein: Kaspar
Geben Sie Ihr Alter ein:
Eingabe unvollständig!
Keine Eingabe für Feld Alter gemacht.
```

13.5 Ein- und Ausgabefehler

Zum Abschluss dieser Vorlesung behandeln wir Probleme im Zusammenhang mit Ein- und Ausgaben. Ein sogenannter *Datenstrom* (engl. *data stream*) ist eine Folge von Bytes, die von einer Quelle zu einem Ziel fliessen. In einem Programm *lesen* wir

Informationen aus einem *Eingabestrom* und *schreiben* Informationen in einen *Ausgabestrom*. Dabei kann ein Programm gleichzeitig mehrere Datenströme verwalten. Es gibt drei Standard-Ströme für die Ein- und Ausgaben:

- Standardausgabe: definiert durch `System.out` (verwendet für `println` Anweisungen)
- Standardeingabe: definiert durch `System.in` (verwendet als Parameter für `Scanner` Objekte)
- Standardfehler: definiert durch `System.err` (verwendet für `println` Anweisungen)

Einige der Ein- und Ausgabe-Operationen können eine `IOException` (oder eine Subklasse hiervon) auslösen:[2]

- Eine Datei existiert möglicherweise nicht
- Selbst wenn die Datei existiert, kann ein Programm sie möglicherweise nicht finden
- Die Datei enthält möglicherweise nicht die Art von Daten, die wir erwarten

Wir haben die Klasse `Scanner` bereits verwendet, um Eingaben aus Dateien entgegenzunehmen. Wird die Datei hierbei nicht gefunden, wird eine `FileNotFoundException` geworfen (die Klasse `FileNotFoundException` ist eine direkte Subklasse der Klasse `IOException`)[3]:

```
public void readFile(String fileName) throws FileNotFoundException {
        // Einlesen der Datei
        Scanner fileReader = new Scanner(new File(fileName));
        // macht etwas mit dem Scanner
}
```

Mit Hilfe der Klasse `PrintWriter` können wir Daten in eine Textdatei *schreiben*. Die beiden Methoden `print` und `println` können hierzu verwendet werden, wie wir es von `System.out` kennen. Ausgabeströme sollten explizit mit der Methode `close` geschlossen werden.

[2] `IOException` ist eine geprüfte `Exception` – wir müssen diese also behandeln.

[3] Hinweis: Dies haben wir in vorherigen Kapiteln schon so programmiert ohne besprochen zu haben, was eine `Exception` tatsächlich ist.

Beispiel 126 *Folgendes Programm erstellt eine Datei* ***output.txt*** *mit Daten, die aus zehn Zeilen besteht, die jeweils zehn ganzzahlige Werte im Bereich von 10 bis 99 enthalten (→ Video K-13.6). Sollte bei der Ausgabe der Daten etwas schief gehen, wird die Methode* ***main*** *eine* ***IOException*** *werfen.*

```
import java.util.Random;
import java.io.*;

public class Output {

        /*
         * Erstellt eine Datei mit Daten, die aus zehn Zeilen besteht,
         * die jeweils zehn ganzzahlige Werte im Bereich von 10 bis 99 enthält.
         */
        public static void main(String[] args) throws IOException {
                final int MAX = 10;
                int value;
                String fileName = "output.txt";

                PrintWriter outFile = new PrintWriter(fileName);
                Random rand = new Random();

                for (int line = 1; line <= MAX; line++) {
                        for (int num = 1; num <= MAX; num++) {
                                value = rand.nextInt(90) + 10;
                                outFile.print(value + "   ");
                        }
                        outFile.println();
                }

                outFile.close();
                System.out.println("Output wurde erstellt: " + fileName);
        }
}
```