

Xpert.press



Die Reihe **Xpert.press** vermittelt Professionals  
in den Bereichen Softwareentwicklung,  
Internettechnologie und IT-Management aktuell  
und kompetent relevantes Fachwissen über  
Technologien und Produkte zur Entwicklung  
und Anwendung moderner Informationstechnologien.

Marco Block

# JAVA Intensivkurs

In 14 Tagen lernen  
Projekte erfolgreich zu realisieren

Unter Mitarbeit von  
Ernesto Tapia und Felix Franke

Mit 90 Abbildungen



Springer

Marco Block

Freie Universität Berlin  
Fachbereich Mathematik und Informatik  
Takustraße 9  
14195 Berlin  
[block@inf.fu-berlin.de](mailto:block@inf.fu-berlin.de)  
<http://www.marco-block.de>

Bibliografische Information der Deutschen Bibliothek  
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

ISSN 1439-5428

ISBN 978-3-540-72271-7 Springer Berlin Heidelberg New York

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zu widerhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media

[springer.de](http://springer.de)

© Springer-Verlag Berlin Heidelberg 2007

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen. Text und Abbildungen wurden mit größter Sorgfalt erarbeitet. Verlag und Autor können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Satz und Herstellung: LE-T<sub>E</sub>X Jelonek, Schmidt & Vöckler GbR, Leipzig

Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg

Gedruckt auf säurefreiem Papier 33/3180 YL – 5 4 3 2 1 0

Für meine Katrin

---

## Vorwort

Es existieren viele gute Bücher über die Programmiersprache Java und warum sollte es sinnvoll sein, ein weiteres Buch zu schreiben? Aus meiner Sicht gibt es zwei wichtige Gründe dafür.

Während der Arbeit in den Kursen mit Studenten aus verschiedenen Fachrichtungen habe ich versucht, in Gesprächen und Diskussionen herauszufinden, welche Probleme es beim Verständnis und beim Erlernen der „ersten Programmiersprache“ gab. Ein Programmierer, dem bereits Konzepte verschiedener Programmiersprachen bekannt sind, erkennt schnell die Zusammenhänge und interessiert sich primär für die syntaktische Ausprägung der neu zu erlernenden Programmiersprache. An der *Freien Universität Berlin* habe ich einige Kurse betreut, bei denen ein Großteil der Studenten keinerlei oder kaum Erfahrung im Umgang mit Programmiersprachen besaßen. Eine Motivation für dieses Buch ist es, die Erkenntnisse und Schlüsselmethoden, die ich im Laufe der Zeit gesammelt habe, auch anderen zugänglich zu machen.

Ich bin der Ansicht, dass gerade Java als Einstiegssprache besonders gut geeignet ist. Sie ist typsicher (die Bedeutung dessen wird in Kapitel 2 klar) und verfügt im Kern über einen relativ kleinen Sprachumfang. Schon in kürzester Zeit und mit entsprechender Motivation lassen sich die ersten Softwareprojekte in Java erfolgreich und zielsicher realisieren.

Dieses Buch hat nicht den Anspruch auf Vollständigkeit, dem werden andere eher gerecht. Es wird vielmehr auf eine Ausbildung zum Selbststudium gesetzt. Ein roter Faden, an dem sich dieses Buch orientiert, versetzt den Leser in nur 14 Tagen in die Lage, vollkommen eigenständig Programme in Java zu entwickeln. Zu den sehr vielseitigen Themengebieten gibt es viele Beispiele und Aufgaben. Ich habe darauf verzichtet, „Standardbeispiele“ zu verwenden und versucht, auf frische neue Anwendungen und Sichtweisen zu setzen.

## Mitarbeiter dieses Buches

Motivation und Ausdauer, aus diesem über die Jahre zusammengestellten Manuskript, ein komplettes Buch zu erarbeiten, sind *Ernesto Tapia* und *Felix Franke* zu verdanken. Beide haben nicht nur mit ihrem Engagement bei dem Aufspüren von Fehlern und Ergänzungen ihren Teil beigetragen, sondern jeweils ein Kapitel beigesteuert und damit das Buch abwechslungsreicher gestaltet.

*Ernesto Tapia*, der auf dem Gebiet der Künstlichen Intelligenz promoviert hat, entwarf das Tetris-Projekt für Kapitel 14 „Entwicklung einer größeren Anwendung“ und hatte großen Einfluss auf Kapitel 13 „Methoden der Künstlichen Intelligenz“. Für die seit vielen Jahren in Forschung und Lehre währende gemeinsame Arbeit und Freundschaft bin ich ihm sehr dankbar.

*Felix Franke* war vor Jahren selbst einer meiner Studenten, dem ich den Umgang mit Java vermittelte. Durch seine Zielstrebigkeit konnte er sein Studium vorzeitig beenden und promoviert jetzt auf dem Gebiet „Neuronale Informationsverarbeitung“. Er hat neben dem Kapitel 12 „Bildverarbeitung“ viele neue Ideen beigesteuert.

## Zusatzmaterialien

Neben den Beispielprogrammen und Aufgaben aus diesem Buch steht eine ständig wachsende Sammlung an kommentierten Programmen und weiteren Aufgaben zum Download zur Verfügung.

<http://www.marco-block.de>

Es gibt auch ein Forum, in dem Fragen erörtert und Informationen ausgetauscht werden können.

## Übersicht der Kapitel

Da die Größe der einzelnen Kapitel etwas variiert, sei dem Leser angeraten auch mal zwei kleine Kapitel an einem Tag durchzuarbeiten, wenn der Stoff keine Schwierigkeiten bereitet. An kniffligen Stellen, wie z. B. der Einführung in die Objektorientierung in Kapitel 7, kann dann mehr Zeit investiert werden. Die Erfahrung zeigt, dass der Lehrstoff dieses Buches in 14 Tagen sicher aufgenommen und erfasst werden kann.

Kapitel 1 soll die Motivation zum Selbststudium wecken und unterstützt die Bereitstellung und Inbetriebnahme einer funktionsfähigen Java-Umgebung.

Kapitel 2 führt behutsam in die grundlegenden Prinzipien der Programmierung ein. Die kleinsten Java-Bausteine werden vorgestellt und es wird damit das Fundament für das Verständnis der Programmierung gelegt.

In Kapitel 3 werden die zuvor eingeführten Prinzipien in Java anhand konkreter Beispiele umgesetzt. Für die praktische Arbeit wird die Klasse zunächst nur als Programmumpf interpretiert.

Kapitel 4 behandelt das Ein- und Auslesen von Daten. Diese Daten können in Dateien vorliegen oder dem Programm auf der Konsole übergeben werden.

In Kapitel 5 wird der Umgang mit Arrays und Matrizen durch das erste Projekt *Conway's Game of Life* vermittelt.

Bevor mit der Objektorientierung begonnen wird, zeigt Kapitel 6 auf, welche Regeln bei der Erstellung von Programmen zu beachten sind und mit welchen Hilfsmitteln Fehler gefunden werden können.

In Kapitel 7 wird das Klassenkonzept vorgestellt. Mit Hilfe eines *Fußballmanagers* wird das Konzept der Vererbung vermittelt. Zusätzlich werden bisher ausgeklammerte Fragen aus den vorhergehenden Kapiteln zum Thema Objektorientierung an dieser Stelle aufgearbeitet.

Java verfügt im Kern über einen relativ kleinen Sprachumfang. Die Sprache lässt sich durch Bibliotheken beliebig erweitern. Kapitel 8 zeigt die Verwendung von Bibliotheken. Ein *Lottoprogramm* und das Projekt *BlackJack* werden mit den bisher kennengelernten Hilfsmitteln realisiert.

Kapitel 9 gibt Schritt für Schritt eine Einführung in die Erstellung von grafischen Oberflächen und die Behandlung von Fenster- und Mausereignissen.

In Kapitel 10 wird neben einer Kurzeinführung in HTML das Konzept von Applets vermittelt. Es genügen oft nur einfache Änderungen, um aus einer Applikation ein Applet zu machen.

Einen Einstieg in die Techniken der Programmierung gibt Kapitel 11. Es werden viele verschiedene Programmbeispiele zu den jeweiligen Entwurfstechniken vorgestellt.

Kapitel 12 macht einen Ausflug in die Bildverarbeitung. *Fraktale* werden gezeichnet und verschiedene Techniken der Bildverarbeitung aufgezeigt.

Kapitel 13 beschäftigt sich mit Aspekten der Künstlichen Intelligenz, wie z. B. der *Erkennung handgeschriebener Ziffern* oder der Funktionsweise eines perfekt spielenden *Tic Tac Toe*-Spiels.

Abschließend werden in Kapitel 14 alle Phasen einer Projektentwicklung für eine Variante des Spiels *Tetris*, vom Entwurf über die Implementierung bis hin zur Dokumentation, dargestellt.

Um den Leser für neue Projekte zu motivieren, werden in Kapitel 15 weitere Konzepte der Softwareentwicklung kurz erläutert.

## Danksagungen

Mein Dank gilt allen, die auf die eine oder andere Weise zur Entstehung dieses Buches beigetragen haben. Dazu zählen nicht nur die Studenten, die sichtlich motiviert waren und mir Freude beim Vermitteln des Lehrstoffs bereiteten, sondern auch diejenigen Studenten, die im Umgang mit Java große Schwierigkeiten hatten. Dadurch wurde ich angeregt auch unkonventionelle Wege auszuprobieren.

Die Vorlesungen an der Freien Universität Berlin von *Klaus Kriegel, Frank Hoffmann und Raúl Rojas* haben mein Verständnis für didaktische Methodik dabei am intensivsten geprägt.

An dieser Stelle möchte ich meinen Eltern, Großeltern und meinem Bruder danken, die mich immer bei allen meinen Projekten bedingungslos unterstützen.

Besonders möchte ich mich auch bei den vielen kritischen Lesern, die zahlreiche Korrekturen und Verbesserungen beigesteuert haben, und für die vielen bereichern den Diskussionen bedanken (in alphabetischer Reihenfolge): *Maro Bader, Anne, Katrin, Inge und Detlef Berlitz, Erik Cuevas, Christian Ehrlich, Dominic Freyberg, Ni klaas Görsch, Christine Gräfe, Ketill Gunnarson, Tobias Hannasky, Julia und Thors ten Hanssen, Frank Hoffmann, Nima Keshvari, André Knuth, Raul Kompaß, Falko Krause, Jan Kretzschmar, Klaus Kriegel, Tobias Losch, Günter Pehl, Marte Ramírez, Raúl Rojas, Michael Schreiber, Bettina Selig, Sonja und Thilo Rörig, Alexander Seibert, Manuel Siebeneicher, Mark Simon, Ole Schulz-Trieglaff, Tilman Walther, Da niel Werner und Daniel Zaldivar*.

Unterstützung erhielt ich von *Sonja Rörig*, die neben der Arbeit als Illustratorin und Designerin, Zeit für meine Abbildungen fand. Das Vererbungsbeispiel mit den witzigen Fußballspielern hat es mir besonders angetan. Vielen lieben Dank für Deine professionelle Hilfe! Auch *Tobias Losch* ist in diesem Zusammenhang noch einmal zu nennen, denn neben den zahlreichen Korrekturhilfen, hat auch er bei einigen Abbildungen geholfen und die Webseite zum Buch entworfen.

Die Zusammenarbeit mit dem Springer-Verlag, gerade bei diesem ersten Buchprojekt, war sehr angenehm und bereichernd. Für die geduldige und fachkundige Betreuung von *Hermann Engesser* und *Gabi Fischer* bin ich sehr dankbar.

Ich wünsche dem Leser genauso viel Spaß beim Lesen wie ich es beim Schreiben hatte und hoffe auf offene Kritik und weitere Anregungen.

---

# Inhaltsverzeichnis

<b>1</b>	<b>Tag 1: Vorbereitungen und Motivation . . . . .</b>	<b>1</b>
1.1	Motivation: Warum gerade Java? . . . . .	1
1.1.1	Programme für Webseiten . . . . .	2
1.2	Vorteile des Selbststudiums . . . . .	2
1.3	Installation von Java . . . . .	3
1.4	Testen wir das Java-System . . . . .	4
<b>2</b>	<b>Tag 2: Grundlegende Prinzipien der Programmentwicklung . . . . .</b>	<b>7</b>
2.1	Primitive Datentypen und ihre Wertebereiche . . . . .	7
2.1.1	Primitive Datentypen in Java . . . . .	8
2.2	Variablen und Konstanten . . . . .	9
2.2.1	Deklaration von Variablen . . . . .	9
2.2.2	Variablen und Konstanten . . . . .	11
2.3	Primitive Datentypen und ihre Operationen . . . . .	11
2.3.1	boolean . . . . .	11
2.3.2	char . . . . .	14
2.3.3	int . . . . .	14
2.3.4	byte, short, long . . . . .	15
2.3.5	float, double . . . . .	16
2.4	Casting, Typumwandlungen . . . . .	16
2.4.1	Übersicht zu impliziten Typumwandlungen . . . . .	18
2.4.2	Die Datentypen sind für die Operation entscheidend . . . . .	19
2.5	Methoden der Programmerstellung . . . . .	20
2.5.1	Sequentieller Programmablauf . . . . .	21
2.5.2	Verzweigungen . . . . .	21
2.5.3	Sprünge . . . . .	22
2.5.4	Schleifen . . . . .	22
2.5.5	Mehrfachverzweigungen . . . . .	22
2.5.6	Mehrfachschleifen (verschachtelte Schleifen) . . . . .	22
2.5.7	Parallelität . . . . .	23
2.5.8	Kombination zu Programmen . . . . .	23

2.6	Programme in Java . . . . .	24
2.6.1	Erstellen eines Javaprogramms . . . . .	24
2.7	Zusammenfassung und Aufgaben . . . . .	25
<b>3</b>	<b>Tag 3: Programmieren mit einem einfachen Klassenkonzept . . . . .</b>	<b>29</b>
3.1	Sequentielle Anweisungen . . . . .	30
3.2	Verzweigungen . . . . .	31
3.2.1	if-Verzweigung . . . . .	31
3.2.2	switch-Verzweigung . . . . .	32
3.3	Verschiedene Schleifentypen . . . . .	33
3.3.1	for-Schleife . . . . .	34
3.3.2	while-Schleife . . . . .	35
3.3.3	do-while-Schleife . . . . .	36
3.4	Sprunganweisungen . . . . .	37
3.4.1	break . . . . .	37
3.4.2	continue . . . . .	39
3.5	Klassen . . . . .	41
3.5.1	Funktionen in Java . . . . .	41
3.6	Zusammenfassung und Aufgaben . . . . .	43
<b>4</b>	<b>Tag 4: Daten laden und speichern . . . . .</b>	<b>45</b>
4.1	Externe Programmeingaben . . . . .	46
4.2	Daten aus einer Datei einlesen . . . . .	47
4.3	Daten in eine Datei schreiben . . . . .	49
4.4	Daten von der Konsole einlesen . . . . .	49
4.5	Zusammenfassung und Aufgaben . . . . .	50
<b>5</b>	<b>Tag 5: Verwendung einfacher Datenstrukturen . . . . .</b>	<b>53</b>
5.1	Arrays und Matrizen . . . . .	53
5.1.1	Erzeugung eines Arrays . . . . .	53
5.1.2	Matrizen oder multidimensionale Arrays . . . . .	55
5.1.3	Conway's Game of Life . . . . .	56
5.1.3.1	Einfache Implementierung . . . . .	57
5.1.3.2	Eine kleine Auswahl besonderer Muster . . . . .	60
5.2	Zusammenfassung und Aufgaben . . . . .	61
<b>6</b>	<b>Tag 6: Debuggen und Fehlerbehandlungen . . . . .</b>	<b>63</b>
6.1	Das richtige Konzept . . . . .	63
6.2	Exceptions in Java . . . . .	65
6.2.1	Einfache try-catch-Behandlung . . . . .	66
6.2.2	Mehrfaeche try-catch-Behandlung . . . . .	67
6.3	Fehlerhafte Berechnungen aufspüren . . . . .	68
6.3.1	Berechnung der Zahl pi . . . . .	68
6.3.2	Zeilenweises Debuggen und Breakpoints . . . . .	71
6.4	Zusammenfassung und Aufgaben . . . . .	71

<b>7 Tag 7: Erweitertes Klassenkonzept . . . . .</b>	73
7.1 Entwicklung eines einfachen Fußballmanagers . . . . .	73
7.1.1 Spieler und Trainer . . . . .	73
7.1.1.1 Generalisierung und Spezialisierung . . . . .	74
7.1.1.2 Klassen und Vererbung . . . . .	75
7.1.1.3 Modifizierer public und private . . . . .	76
7.1.1.4 Objekte und Instanzen . . . . .	77
7.1.1.5 Konstruktoren in Java . . . . .	78
7.1.2 Die Mannschaft . . . . .	81
7.1.3 Turniere, Freundschaftsspiele . . . . .	82
7.1.3.1 Ein Interface festlegen . . . . .	82
7.1.3.2 Freundschaftsspiel Deutschland–Brasilien der WM-Mannschaften von 2006 . . . . .	85
7.1.3.3 Interface versus abstrakte Klasse . . . . .	88
7.2 Aufarbeitung der vorhergehenden Kapitel . . . . .	89
7.2.1 Referenzvariablen . . . . .	89
7.2.2 Zugriff auf Attribute und Methoden durch Punktnotation . . . . .	90
7.2.3 Die Referenzvariable this . . . . .	91
7.2.4 Prinzip des Überladens . . . . .	92
7.2.5 Überladung von Konstruktoren . . . . .	92
7.2.5.1 Der Copy-Konstruktor . . . . .	93
7.2.6 Garbage Collector . . . . .	94
7.2.7 Statische Attribute und Methoden . . . . .	94
7.2.8 Primitive Datentypen und ihre Wrapperklassen . . . . .	95
7.2.9 Die Klasse String . . . . .	96
7.2.9.1 Vergleich von Zeichenketten . . . . .	97
7.3 Zusammenfassung und Aufgaben . . . . .	98
<b>8 Tag 8: Verwendung von Bibliotheken . . . . .</b>	101
8.1 Standardbibliotheken . . . . .	101
8.2 Mathematik-Bibliothek . . . . .	103
8.3 Zufallszahlen in Java . . . . .	104
8.3.1 Ganzzahlige Zufallszahlen vom Typ int und long . . . . .	105
8.3.2 Zufallszahlen vom Typ float und double . . . . .	105
8.3.3 Weitere nützliche Funktionen der Klasse Random . . . . .	105
8.4 Das Spiel Black Jack . . . . .	106
8.4.1 Spielkarten . . . . .	106
8.4.2 Wertigkeiten der Karten . . . . .	106
8.4.3 Der Spielverlauf . . . . .	107
8.4.4 Spieler, Karten und Kartenspiel . . . . .	108
8.4.4.1 Verwendungsbeispiel für Datenstruktur Vector . . . . .	108
8.4.4.2 Implementierung der Klassen Spieler, Karten und Kartenspiel . . . . .	109
8.4.5 Die Spielklasse BlackJack . . . . .	112
8.5 JAMA - Lineare Algebra . . . . .	118

8.6 Eine eigene Bibliothek bauen . . . . .	120
8.7 Zusammenfassung und Aufgaben . . . . .	121
<b>9 Tag 9: Grafische Benutzeroberflächen . . . . .</b>	<b>123</b>
9.1 Fenstermanagement unter AWT . . . . .	123
9.1.1 Ein Fenster erzeugen . . . . .	123
9.1.2 Das Fenster zentrieren . . . . .	124
9.2 Zeichenfunktionen innerhalb des Fensters verwenden . . . . .	125
9.2.1 Textausgaben . . . . .	126
9.2.2 Zeichenelemente . . . . .	126
9.2.3 Die Klasse Color verwenden . . . . .	127
9.2.4 Bilder laden und anzeigen . . . . .	128
9.3 Auf Fensterereignisse reagieren und sie behandeln . . . . .	130
9.3.1 Fenster mit dem Interface WindowListener schließen . . . . .	130
9.3.2 GUI-Elemente und ihre Ereignisse . . . . .	132
9.3.2.1 Layout Manager . . . . .	133
9.3.2.2 Die Komponenten Label und Button . . . . .	133
9.3.2.3 Die Komponente TextField . . . . .	134
9.4 Auf Mausereignisse reagieren . . . . .	136
9.5 Zusammenfassung und Aufgaben . . . . .	137
<b>10 Tag 10: Appletprogrammierung . . . . .</b>	<b>139</b>
10.1 Kurzeinführung in HTML . . . . .	139
10.2 Applets im Internet . . . . .	140
10.3 Bauen eines kleinen Applets . . . . .	141
10.4 Verwendung des Appletviewers . . . . .	141
10.5 Eine Applikation zum Applet umbauen . . . . .	143
10.5.1 Konstruktor zu init . . . . .	143
10.5.2 paint-Methoden anpassen . . . . .	144
10.5.3 TextField-Beispiel zum Applet umbauen . . . . .	144
10.6 Flackernde Applets vermeiden . . . . .	146
10.6.1 Die Ghosttechnik anwenden . . . . .	147
10.6.2 Die paint-Methode überschreiben . . . . .	149
10.7 Ein Beispiel mit mouseDragged . . . . .	150
10.8 Zusammenfassung und Aufgaben . . . . .	151
<b>11 Tag 11: Techniken der Programmierung . . . . .</b>	<b>153</b>
11.1 Der Begriff Algorithmus . . . . .	153
11.2 Entwurfs-Techniken . . . . .	154
11.2.1 Prinzip der Rekursion . . . . .	154
11.2.2 Brute Force . . . . .	156
11.2.3 Greedy . . . . .	157
11.2.4 Dynamische Programmierung, Memoisierung . . . . .	158
11.2.5 Divide and Conquer . . . . .	160
11.3 Algorithmen miteinander vergleichen . . . . .	160

11.4	Kleine algorithmische Probleme . . . . .	161
11.4.1	Identifikation und Erzeugung von Primzahlen mit Brute Force . . . . .	161
11.4.2	Sortieralgorithmen . . . . .	162
11.4.2.1	InsertionSort . . . . .	162
11.4.2.2	BubbleSort . . . . .	163
11.4.2.3	QuickSort . . . . .	165
11.4.3	Needleman-Wunsch-Algorithmus . . . . .	166
11.5	Zusammenfassung und Aufgaben . . . . .	168
<b>12</b>	<b>Tag 12: Bildverarbeitung</b> . . . . .	171
12.1	Das RGB-Farbmodell . . . . .	171
12.2	Grafische Spielerei: Apfelmännchen . . . . .	173
12.2.1	Mathematischer Hintergrund . . . . .	174
12.2.2	Fraktale . . . . .	175
12.2.2.1	Implementierung eines einfachen Apfelmännchens . . . . .	176
12.2.3	Die Klasse BufferedImage . . . . .	177
12.2.4	Bilder laden und speichern . . . . .	179
12.2.5	Bilder bearbeiten . . . . .	184
12.2.5.1	Ein Bild invertieren . . . . .	184
12.2.5.2	Erstellung eines Grauwertbildes . . . . .	185
12.2.5.3	Binarisierung eines Grauwertbildes . . . . .	186
12.3	Zusammenfassung und Aufgaben . . . . .	187
<b>13</b>	<b>Tag 13: Methoden der Künstlichen Intelligenz</b> . . . . .	189
13.1	Mustererkennung . . . . .	189
13.1.1	$k$ -nn . . . . .	192
13.1.1.1	Visualisierung . . . . .	193
13.1.2	$k$ -means . . . . .	195
13.1.2.1	Expectation-Maximization als Optimierungsverfahren . . . . .	197
13.1.2.2	Allgemeine Formulierung des $k$ -means Algorithmus . . . . .	197
13.1.2.3	Implementierung des $k$ -means . . . . .	198
13.2	Spieltheorie . . . . .	202
13.2.1	MinMax-Algorithmus . . . . .	202
13.2.1.1	MinMax-Algorithmus mit unbegrenzter Suchtiefe .	204
13.2.1.2	MinMax-Algorithmus mit begrenzter Suchtiefe und Bewertungsfunktion . . . . .	204
13.2.1.3	Beispiel TicTacToe . . . . .	205
13.3	Zusammenfassung und Aufgaben . . . . .	208

<b>14 Tag 14: Entwicklung einer größeren Anwendung . . . . .</b>	211
14.1 Entwurf eines Konzepts . . . . .	211
14.1.1 Klassendiagramm . . . . .	213
14.1.1.1 GUI Klassen . . . . .	213
14.1.1.2 Spiellogik . . . . .	214
14.1.1.3 Spieldatenverwaltung . . . . .	214
14.1.1.4 Komplettes Klassendiagramm . . . . .	215
14.2 Implementierung . . . . .	215
14.2.1 Klasse TeeTristBox . . . . .	215
14.2.2 Klasse TeeTristStein . . . . .	216
14.2.3 Klasse TeeTristSpielfeld . . . . .	219
14.2.4 Klasse SpielThread . . . . .	223
14.2.5 Klasse TeeTristPanel . . . . .	227
14.2.6 Klasse TeeTrist . . . . .	228
14.3 Spielen wir ein Spiel TeeTrist . . . . .	228
14.4 Dokumentation mit javadoc . . . . .	229
14.5 Zusammenfassung und Aufgaben . . . . .	230
<b>15 Java – Weiterführende Konzepte . . . . .</b>	233
15.1 Professionelle Entwicklungsumgebungen . . . . .	233
15.2 Das Klassendiagramm als Konzept einer Software . . . . .	234
15.2.1 Klassendiagramm mit UML . . . . .	234
15.2.1.1 Klasse . . . . .	234
15.2.1.2 Vererbung . . . . .	235
15.2.1.3 Beziehungen zwischen Klassen . . . . .	235
15.3 Verwendung externer Bibliotheken . . . . .	236
15.4 Zusammenarbeit in großen Projekten . . . . .	237
<b>Glossar . . . . .</b>	239
<b>Literatur . . . . .</b>	245
<b>Sachverzeichnis . . . . .</b>	249

# 1

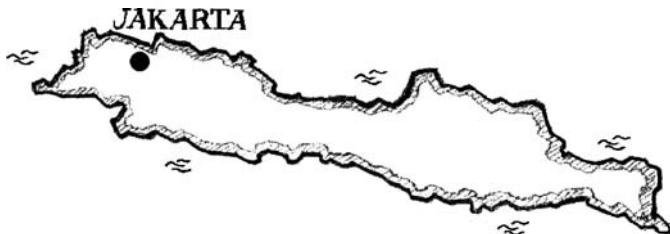
---

## Tag 1: Vorbereitungen und Motivation

Unser erster Abschnitt beschäftigt sich mit der Installation von Java, der Wahl einer Entwicklungsumgebung und der Motivation zum Selbststudium. Ein innerer Antrieb und viele praktische Übungen sind unerlässlich für das Erlernen einer Programmiersprache. Programmieren heißt auch meistens, mit anderen oder für andere zu programmieren. Daher werden wir lernen, durch einen verständlichen Programmaufbau, mit Diagrammen und ausreichender Kommentierung, die Kommunikation mit anderen zu verbessern, um allein oder gemeinsam komplexe Projekte meistern zu können.

### 1.1 Motivation: Warum gerade Java?

Es gibt viele Gründe mit dem Programmieren zu beginnen, aber warum sollte es gerade die Programmiersprache Java sein?



Java ist auch eine indonesische Insel mit der Hauptstadt Jakarta.

Die Vorteile von Java liegen in dem vergleichsweise zu anderen Programmiersprachen relativ kleinen Befehlsumfang und der strikten Typsicherheit. Java ist plattformunabhängig und kann daher auf verschiedenen Betriebssystemen eingesetzt werden. Es gibt eine sehr große Java-Community und die Sprache ist leicht zu erlernen.

Generell gilt: Die Chance in Java etwas falsch zu machen ist sehr viel kleiner als z. B. bei C++.

Es gibt aber auch Nachteile. Geschwindigkeitsrelevante Softwaresysteme sollten nicht mit Java, sondern eher mit Programmiersprachen wie C oder C++ geschrieben werden<sup>1</sup>. Der Speicher kann nicht direkt angesprochen werden, alles wird über eine virtuelle Maschine gesteuert. Damit ist auch sichergestellt, dass sicherheitsproblematische Speicherabschnitte nicht einfach so ausgelesen werden können (siehe dazu Abschnitt 1.3.9 in [37]).

### 1.1.1 Programme für Webseiten

Neben den üblichen **Applikationen**, das sind selbstständige Anwendungsprogramme (stand-alone Applications), können wir mit Java aber auch Programme schreiben, die in eine Webseite eingebunden werden können. Diese Programme tragen den Namen **Applets**.

In diesem Buch werden wir beide Programmtypen kennenlernen und mit ihnen arbeiten. Wir werden sehen, dass oft nur wenige Arbeitsschritte notwendig sind, um eine Applikation in ein Applet zu ändern.

## 1.2 Vorteile des Selbststudiums

Es gibt zwei Ansätze beim Erlernen einer Programmiersprache. Der erste ist ein detailliertes Studium aller Funktionen und Eigenschaften einer Sprache und das fleißige Erlernen aller Vokabeln. Dabei können zu jedem Teilaspekt Beispiele angebracht und Programmierübungen gelöst werden. Diese Strategie könnte man als **Bottom-Up-Lernen** bezeichnen, da erst die kleinen Bausteine gelernt und später zu großen Projekten zusammengefügt werden.

Beim zweiten Ansatz werden die zu erlernenden Bausteine auf ein Minimum beschränkt. Es werden eher allgemeine Konzepte als speziell in einer Sprache existierende Befehle und Funktionen vermittelt. Mit diesem Basiswissen und verschiedenen Wissensquellen, die zu Rate gezogen werden können, lassen sich nun ebenfalls erfolgreich größere Projekte realisieren. Die entsprechenden Bausteine werden während des Entwicklungsprozesses studiert und erarbeitet. Das zweite Verfahren, wir könnten es **Top-Down-Lernen** nennen, eignet sich für Personen, die sich nicht auf eine spezielle Programmiersprache beschränken und schnell mit dem Programmieren beginnen wollen. Der Lernprozess findet dabei durch die praktische Handhabung der Sprache statt.

Eine Analogie lässt sich zum Erlernen von Fremdsprachen ziehen. Man kann beginnen, indem man alle Vokabeln paukt und dann versucht, in entsprechenden Situationen die Sätze zusammenzubauen. Der Lernprozess findet also primär vor der

---

<sup>1</sup> Trotzdem hat sich Java gerade in der Mobilkommunikation durchgesetzt.

praktischen Ausübung statt. Der zweite Weg führt erst zu einer Situation und liefert dann die entsprechenden Vokabeln, die es nachzuschlagen gilt. Man kann früher mit dem Sprechen anfangen, da sich der Lernprozess in die praktische Ausübung verlagert hat.

Ich bevorzuge die Top-Down-Variante und habe mit Bedacht diesen didaktischen Weg für die Konzeption dieses Buches gewählt. Aus diesem Grund wird der Leser schnell mit dem Programmieren beginnen können, ohne erst viele Seiten Theorie studieren zu müssen. Da dieses Buch nicht den Anspruch auf Vollständigkeit hat, wird dem Leser im Laufe der ersten Tage vermittelt, wie und wo Informationsquellen zu finden sind und wie man sie verwendet, um Probleme zu lösen.

Es ist unmöglich, ein Buch zu schreiben, das auf alle erdenklichen Bedürfnisse eingeht und keine Fragen offen lässt. Deshalb ist es ratsam, zu entsprechenden Themenbereichen zusätzliche Quellen zu Rate zu ziehen. An vielen Stellen wird darauf verwiesen. Dieses Buch eignet sich nicht nur als Einstieg in die Programmierung mit Java, sondern versucht auch einen Blick auf den ganzen Prozess einer Softwareentwicklung zu geben.

Java ist nur ein Werkzeug zur Formulierung von Programmen. Zur erfolgreichen Realisierung von Projekten gehört aber noch viel mehr. Projekte müssen genau geplant und vor der eigentlichen Erstellung möglichst so formuliert werden, dass jeder Softwareentwickler weiß, was er zu tun hat.

Dieses Buch soll dem Leser einen umfassenden Einstieg in den gesamten Prozess der Softwareentwicklung ermöglichen.

## 1.3 Installation von Java

Wir beginnen zunächst damit, uns mit dem System vertraut zu machen. Dazu installieren wir auf dem vorhandenen Betriebssystem eine aktuelle Java-Version<sup>2</sup>. Es ist darauf zu achten, dass diese eine SDK-Version ist (SDK=Software Development Kit oder JDK=Java Development Kit). Zu finden ist sie zum Beispiel auf der Internetseite von Sun Microsystems [50]:

<http://java.sun.com/>

Eine wichtige Anmerkung an dieser Stelle: Bei vielen Betriebssystemen ist es notwendig, die **Umgebungsvariablen** richtig zu setzen. Bei Windows XP beispielsweise geht man dazu in die „Systemsteuerung“ und dort auf „Systemeigenschaften“, dann müssen unter der Rubrik „Erweitert“ die Umgebungsvariablen geändert werden. Es gibt eine Variable *PATH*, die um den Installationsordner + „/bin“ von Java

---

<sup>2</sup> Die Programme dieses Buches wurden mit Java 1.5 erstellt. Es wurde versucht, allgemein gültige Funktionen und Konzepte zu verwenden.

erweitert werden muss. In älteren Systemen muss noch eine neue Umgebungsvariable mit dem Namen *CLASSPATH* erzeugt und ihr ebenfalls der Javapfad (zusätzlich zum Javapfad muss „; .;“ angehängt werden) zugewiesen werden.

Nach der Installation von Java müssen wir uns für eine Entwicklungsumgebung entscheiden und diese ebenfalls installieren. Es gibt eine Reihe von Umgebungen, die es dem Programmierer vereinfachen, Programme in Java zu entwickeln. Hier sind ein paar kostenfrei erhältliche Programme aufgelistet:

- ✓ Jeder normale Texteditor kann verwendet werden
- ✓ Eclipse (online [52])
- ✓ JCreator (online [53])
- ✓ NetBeans (online [54])
- ✓ Borland JBuilder (online [55])

Es gibt viele weitere Editoren für Java. Da ich keinen bevorzugen, sondern den an gehenden Programmierer für die internen Prozesse sensibilisieren möchte, verzichte ich auf den Einsatz professioneller Entwicklungsumgebungen und schreibe alle Programme mit einem einfachen Texteditor. Dem Leser sei es selbst überlassen, eine Wahl zu treffen. Eine Übersicht findet sich beispielsweise auf der Internetseite vom Java-Tutor [51].

## 1.4 Testen wir das Java-System

Bevor es mit der Programmierung losgeht, wollen wir das System auf Herz und Nieren prüfen. Dazu starten wir eine **Konsole** (unter Windows heißt dieses Programm **Eingabeaufforderung**). Wenn die Installation von Java vollständig abgeschlossen ist und die Umgebungsvariablen richtig gesetzt sind, dann müsste die Eingabe der Anweisung *javac* zu folgender (hier gekürzter) Ausgabe führen:

```
C:\>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                     Generate no debugging info
  -g:{lines , vars , source}   Generate only some debugging info
  -nowarn                      Generate no warnings
  -verbose                     Output messages about what the compiler ...
  -deprecation                 Output source locations where deprecate ...
  ...
```

So oder so ähnlich sollte die Ausgabe aussehen. Falls eine Fehlermeldung an dieser Stelle auftaucht, z. B.

```
C:\>javac  
Der Befehl "javac" ist entweder falsch geschrieben oder  
konnte nicht gefunden werden.
```

bedeutet dies, dass das Programm *javac* nicht gefunden wurde. An dieser Stelle sollte vielleicht die *PATH*-Variable noch einmal überprüft (siehe dazu Abschnitt 1.3) oder ein Rechnerneustart vorgenommen werden.

Testen wir unser System weiter. Um herauszufinden, ob der *CLASSPATH* richtig gesetzt ist, schreiben wir ein kurzes Programm. Dazu öffnen wir einen Editor und schreiben folgende Zeile hinein:

```
public class Test {}
```

Diese Datei speichern wir mit dem Namen **Test.java** in einem beliebigen Ordner. Am besten wäre an dieser Stelle vielleicht, die Datei einfach auf „C:\“ zu speichern, denn wir werden das Programm anschließend sowieso wieder löschen. In meinem Fall hatte ich einen Ordner mit dem Namen „Java“ in „C:\“ angelegt und dort hinein die Datei gespeichert.

Anschließend navigieren wir in der Konsole zu der Datei und geben die folgende Anweisung ein:

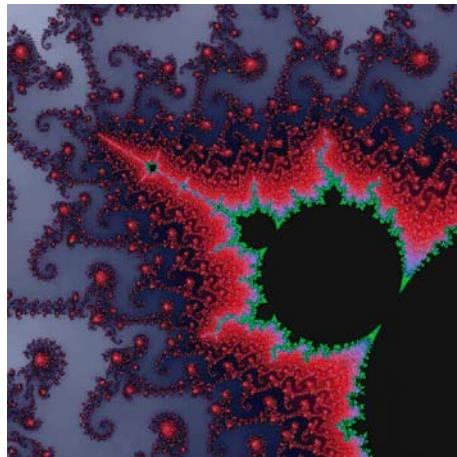
```
C:\>cd Java  
C:\Java>javac Test.java  
C:\Java>
```

Wenn keine Fehlermeldung erscheint, ist das System bereit und wir können endlich mit der Programmierung beginnen.

Dieses kleine Beispiel hat uns einen ersten Einblick in die Programmierstellung mit Java gegeben. Wir werden in einem Editor die Programme schreiben und sie anschließend in der Konsole starten. Mit der Anweisung *javac* wird das Programm in den sogenannten Byte-Code umgewandelt, dabei wird eine Datei mit dem gleichen Namen erzeugt, aber mit der Endung „.class“ versehen. Eine solche Datei wird auch für unser kleines Beispiel erzeugt.

```
C:\Java>javac Test.java  
C:\Java>dir  
Volume in Laufwerk C: hat keine Bezeichnung.  
Volume Seriennummer: A8DB-F3AF  
  
Verzeichnis von C:\Java  
  
14.03.2007 12:22 <DIR> .  
14.03.2007 12:22 <DIR> ..  
14.03.2007 12:22 182 Test.class  
14.03.2007 12:10 19 Test.java  
2 Datei(en) 201 Bytes  
2 Verzeichnis(se), 372.468.928.512 Bytes frei
```

Später werden wir noch erfahren, wie wir diesen Byte-Code starten und beispielsweise diese schicke Ausgabe erzeugen können:



Ich hoffe, dass Motivation und Kreativität für die Realisierung von Projekten in Java jetzt ausreichend vorhanden sind. Wir werden im folgenden Kapitel mit den kleinsten Bausteinen beginnen.

## Tag 2: Grundlegende Prinzipien der Programmierung

Um einen Einstieg in die Programmierung zu wagen, müssen wir uns zunächst mit den primitiven Datentypen und den Prinzipien der Programmierung vertraut machen. In Java wird im Gegensatz zu anderen Programmiersprachen, wie z. B. C oder C++, besonderer Wert auf Typsicherheit gelegt. Damit ist gemeint, dass das Programm bei der Verwaltung von Speicher immer genau Bescheid wissen möchte, um welchen Datentyp es sich handelt. Im Anschluss daran können wir mit den ersten Programmierübungen beginnen.

### 2.1 Primitive Datentypen und ihre Wertebereiche

Mit Programmen wollen wir den Computer dazu bringen, bestimmte Aufgaben und Probleme für uns zu lösen. Wir haben es dabei immer mit Daten zu tun, die entweder als Ein- oder Ausgabe für die Lösung einer Aufgabe oder zum Speichern von Zwischenlösungen verwendet werden.

Die vielen Jahre der Softwareentwicklung haben gezeigt, dass es drei wichtige Kategorien von Daten gibt, die wir als atomar bezeichnen können, da sie die kleinsten Bausteine eines Programms darstellen und wir sie später zu größeren Elementen zusammenfassen werden.

#### Wahrheitswerte

Wahrheitswerte repräsentieren die kleinste Einheit im Rechner. Auf 0 oder 1, *wahr* oder *falsch*, Strom *an* oder *aus* beruhen alle technischen Ereignisse eines Rechners. In jedem Programm werden sie verwendet, ob bewusst oder unbewusst.

#### Zeichen, Symbole

Die Ein- und Ausgaben von Text sind ebenfalls wichtige Hilfsmittel für einen Programmierer. Zeichen oder Symbole lassen sich später zu einem größeren Datentyp (Zeichenkette) zusammenfassen.

## Zahlen

Zahlen sind wichtig, um beispielsweise Ereignisse zu zählen. Je nach Verwendungszweck beanspruchen Zahlen mal mehr und mal weniger Speicher im Rechner. Eine Gleitkommazahl, die z. B. einen Börsenkurs wiedergibt, benötigt mehr Speicher als eine ganze Zahl, die die Stunden einer Uhr repräsentiert.

### 2.1.1 Primitive Datentypen in Java

Je nach Programmiersprache werden nun verschiedene Datentypen aus diesen drei Kategorien angeboten. Wir nennen sie, da sie die kleinsten Bausteine sind, primitive Datentypen. In Java gibt es:

**Wahrheitswerte:** boolean

**Zeichen, Symbole:** char

**Zahlen:** byte, short, int, long, float, double

Wie diese Übersicht zeigt, wird in der Programmierung dem Bereich der Zahlen eine besondere Aufmerksamkeit gewidmet. Das hat gute Gründe. Um gute, schnelle Programme zu schreiben, ist es notwendig, sich Gedanken über die verwendeten Datentypen zu machen. Sicherlich könnte der Leser der Meinung sein, dass ein Zahlenotyp ausreicht, beispielsweise ein double, der sowohl positive und negative als auch ganze und gebrochene Zahlen darzustellen vermag. Das ist korrekt.

Aber ein double benötigt im Rechner jede Menge Speicher und für Operationen, wie Addition, Subtraktion, Multiplikation und Division, einen größeren Zeitaufwand als z. B. ein short. Da wir in Programmen meistens sehr viele Datentypen und Operationen auf diesen verwenden wollen, um Aufgaben zu lösen, gilt die Devise, immer den Datentyp zu verwenden, der für die Aufgabe geeignet ist und den kleinsten Speicherbedarf hat.

Um nun entscheiden zu können, welche Datentypen in Frage kommen, bleibt es uns nicht erspart einmal einen Blick auf die Wertebereiche zu werfen:

Datentyp	Wertebereich	BIT
boolean	true, false	8
char	0 ... 65.535 (z. B. 'a','b',...,'A',...,'1',...)	16
byte	-128 bis 127	8
short	-32.768 bis 32.767	16
int	-2.147.483.648 bis 2.147.483.647	32
long	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	64
float	+/-1,4E-45 bis +/-3,4E+38	32
double	+/-4,9E-324 bis +/-1,7E+308	64

Übersicht der primitiven Datentypen in Java [38]. Die dritte Spalte zeigt den benötigten Speicher in BIT.

Diese Tabelle muss man jetzt nicht auswendig lernen, es genügt zu wissen, wo man sie findet. Von einer Programmiersprache zu einer anderen, selbst bei verschiedenen Versionen einer Programmiersprache, können sich diese Werte unterscheiden, oder es kommen neue primitive Datentypen hinzu.

Dem Leser mag aufgefallen sein, dass für einen boolean, obwohl er nur zwei Zustände besitzt, 8 BIT reserviert sind. Logischerweise benötigt ein boolean nur 1 BIT zur Speicherung der zwei möglichen Zustände true und false, aber aus technischen Gründen sind 8 BIT die kleinste Speichereinheit in der aktuellen Rechnergeneration.

## 2.2 Variablen und Konstanten

### 2.2.1 Deklaration von Variablen

Um die Datentypen aus dem vorhergehenden Abschnitt verwenden zu können, müssen wir, wie es z. B. in der Algebra üblich ist, **Variablen** verwenden. Diese Variablen dienen als Platzhalter für die Werte im Speicher und belegen je nach Datentyp mehr oder weniger Ressourcen. Um eine Variable eines bestimmten Typs im Speicher anzulegen, die für den Rest des Programms (oder bis sie gelöscht wird) bestehen bleibt, müssen wir sie **deklarieren**.

```
<Datentyp> <name>;
```

Beispielsweise wollen wir einen Wahrheitswert verwenden:

```
boolean a;
```

Die Variable a steht nach der **Deklaration** bereit und kann einen Wert zugewiesen bekommen. Das nennen wir eine **Zuweisung**.

Wir könnten nun nacheinander Variablen deklarieren und ihnen anschließend einen Wert zuweisen. Es lassen sich auch mehrere Variablen eines Datentyps gleichzeitig deklarieren oder, wie es die letzte Zeile demonstriert, Deklaration und Zuweisung in einer Anweisung ausführen.

```
boolean a;
a = true;
int b;
b = 7;
float c, d, e;
char f = 'b';
```

Die Bezeichnung einer Variable innerhalb des Programms wird uns überlassen. Es gibt zwar ein paar Beschränkungen für die Namensgebung, aber der wichtigste Grund für die Wahl ist die Aufgabe der Variable. Ein paar Beispiele:

```
boolean istFertig;
double kursWert;
int schrittZaehler;
```

Die Beschränkungen für die Bezeichnungen in Java lauten:

Variablennamen beginnen mit einem Buchstaben, Unterstrich oder Dollarzeichen (nicht erlaubt sind dabei Zahlen!). Nach dem ersten Zeichen dürfen aber sowohl Buchstaben als auch Zahlen folgen. **Operatoren** und **Schlüsselwörter** dürfen ebenfalls nicht als Variablennamen verwendet werden.

In diesem Buch werden bei allen Programmbeispielen die Schlüsselwörter fett gekennzeichnet. Diese Form der Kennzeichnung nennt man **Syntaxhervorhebung** und sie fördert die Übersicht und Lesbarkeit von Programmen.

### Reservierte Schlüsselwörter

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

Noch ein Hinweis an dieser Stelle: Java ist eine sogenannte **textsensitive Sprache**, was bedeutet, dass auf Groß- und Kleinschreibung geachtet werden muss. Folgendes würde also nicht funktionieren:



```
boolean istFertig;
istFERTIG = true;
```

Damit wären wir also in der Lage, Programme quasi wie einen Aufsatz hintereinander weg zu schreiben. Durch eine überlegte Variablenbezeichnung werden zwei Dinge gefördert. Zum einen benötigt der Programmator weniger Zeit, um seine eigenen Programme zu lesen, deren Erstellung vielleicht schon etwas länger zurück liegt, und zum anderen fördert es die Zusammenarbeit mit anderen Programmierern.

Im Laufe der Zeit haben sich einige Konventionen bezüglich der Erstellung von Programmen etabliert. Man sollte sich an diese halten, um sich selbst und anderen das Leben nicht unnötig schwer zu machen. Frei nach dem Motto: Wir müssen nicht immer alles machen, was erlaubt ist.

Die gängigen Konventionen werden nach und nach in den folgenden Abschnitten beschrieben.

## 2.2.2 Variablen und Konstanten

Variablen sind eine schöne Sache und unverzichtbar. Manchmal ist es notwendig, Variablen zu verwenden, die die Eigenschaft besitzen, während des Programmablaufs unverändert zu bleiben. Beispielsweise könnte ein Programm eine Näherung der Zahl  $\pi$  (pi) verwenden. Der Näherungswert soll während des gesamten Programmablaufs Bestand haben und nicht geändert werden.

```
double pi = 3.14159;
```

Damit aber nun gewährleistet ist, dass man selbst oder auch ein anderer Softwareentwickler weiß, dass dieser Platzhalter im Speicher nicht variabel und demzufolge keine Änderung erlaubt ist, schreiben wir einfach ein **final** davor, verwenden (laut Konvention) für den Namen nur Großbuchstaben und machen so aus der Variable eine **Konstante**.

```
final <Datentyp> <NAME>;
```

```
final double PI = 3.14159;
```

Nun herrscht Klarheit. Wir haben anschließend keine Erlaubnis, in einem späteren Programmabschnitt diese Konstante zu ändern. Java kümmert sich darum und wird einen Fehler melden, falls wir es dennoch versuchen.

## 2.3 Primitive Datentypen und ihre Operationen

Die vorherige ausführliche Einführung diente dem Zweck, im Folgenden die **Operationen** der einzelnen Datentypen besser verstehen zu können. Die Syntax (das Aufschreiben des Programms in einem vorgegebenen Format) lenkt nun nicht mehr ab und wir können uns die wesentlichen Verknüpfungsmöglichkeiten der Datentypen anschauen.

### 2.3.1 boolean

Bezeichnet einen Wahrheitswert und kann demzufolge `true` (wahr) oder `false` (falsch) sein.

```
boolean b;  
b = false;
```

Zunächst wird `b` als boolean deklariert und ihm anschließend der Wert `false` zugewiesen (mit `=` wird ein Wert zugewiesen). Mit dem Datentyp boolean lassen sich alle logischen Operationen ausführen. Es gibt eine weit verbreitete Möglichkeit, auf die wir später noch genauer eingehen werden, einen boolean für eine Entscheidung zu verwenden:

```
if (<boolean>) <Anweisung>;
```

Wenn der boolean den Wert `true` hat, dann wird die nachfolgende Anweisung ausgeführt. Im anderen Fall, wenn der boolean den Wert `false` hat, überspringen wir die Anweisung.

Es lassen sich aber auch, wie schon erwähnt, logische Operationen mit einem boolean ausführen. Wir verwenden für die folgenden Abschnitte zwei boolean  $B_1$  und  $B_2$  mit ihren möglichen Belegungen 1 für `true` und 0 für `false`.

## Das logische UND

Nach folgender Wertetabelle wird die logische Verknüpfung **UND** angewendet:

$B_1$	$B_2$	$B_1 \text{ UND } B_2$
0	0	0
0	1	0
1	0	0
1	1	1

Nur wenn beide Operanden `true` sind, liefert die Operation **UND** auch ein `true`. In Java schreiben wir die Operation **UND** mit dem Symbol `&&`. Schauen wir uns ein Beispiel an:

```
boolean a, b, c;
a = true;
b = false;
c = a && b;
```

Wenn wir uns die Wertetabelle anschauen, dann sehen wir für den Fall  $B_1 = 1$  und  $B_2 = 0$ , dass  $B_1 \text{ UND } B_2$ , also `c`, den Wert 0, also `false` hat.

## Das logische ODER

Folgende Wertetabelle zeigt den **ODER**-Operator:

$B_1$	$B_2$	$B_1$ ODER $B_2$
0	0	0
0	1	1
1	0	1
1	1	1

Einer der beiden Operanden mit dem Wert `true` genügt, um ein `true` zu liefern. Wir verwenden das Symbol `||` für den **ODER**-Operator.

```
boolean a, b, c;
a = true;
b = a && a;
c = b || a;
```

Bei der Auswertung für `c` ermitteln wir zuerst den Wert `true` für `b`, da `true&&true` gleich `true` ist und erhalten schließlich für `c` den Wert `true`, da `true||true` ebenfalls `true` ist .

## Das logische NICHT

Eine einfache Umkehrung eines boolean wird durch die Negation erreicht.

$B_1$	NICHT $B_1$
0	1
1	0

Der **NICHT**-Operator wird in Java mit `!` symbolisiert.

Jetzt sind wir in der Lage, beliebige logische Funktionen zu bauen und zu verknüpfen, denn alle logischen Funktionen lassen sich durch die Bausteine UND, ODER und NICHT konstruieren.

```
boolean a = true, b = false, c, d;
c = a && b;
d = (a || b) && !c
```

In diesem Beispiel wurden die booleschen Operatoren `&&` (UND), `||` (ODER) und `!` (NICHT) verwendet. Es gibt auch noch das ENTWEDER-ODER, das durch `^` (XOR) symbolisiert wird und nur `true` liefert, wenn einer der beiden Operanden `true` und der andere `false` ist.

### 2.3.2 char

Zeichen oder Symbole werden durch den Datentyp `char` identifiziert. Wir schreiben ein Zeichen zwischen ' ', z. B. 'a'.

```
char d = '7';
char e = 'b';
```

Die Variable `d` trägt als Inhalt das Zeichen '7', sie wird aber nicht als die Zahl 7 interpretiert. Alle Datentypen lassen sich auch vergleichen. Anhand des Datentyps `char` schauen wir uns die relationalen Operatoren (Vergleichsoperatoren) an:

```
boolean d, e, f, g;
char a, b, c;
a = '1';
b = '1';
c = '5';

d = a == b;
e = a != b;
f = a < c;
g = c >= b;
```

Es existieren folgende Vergleichsoperatoren: `==` (gleich), `!=` (ungleich), `<`, `>`, `<=` und `>=`. Vergleichsoperatoren liefern einen `boolean`, wie z. B. in der Zeile `d = a == b;`, da `a==b` entweder `true` oder `false` ist. In unserem Beispiel ergeben sich `d=true`, `e=false`, `f=true` und `g=true`.

### 2.3.3 int

Ganzzahlen im Bereich von  $-2.147.483.648$  bis  $2.147.483.647$ . Der Datentyp `int` wird wahrscheinlich am häufigsten verwendet.

```
int a, b = 0;
a = 10;
```

Die Variablen `a` und `b` werden erzeugt und schon bei der Deklaration wird `b` der Wert 0 zugewiesen.

Das Schöne an Zahlen ist, dass wir sie addieren, subtrahieren, multiplizieren und dividieren können. Aber mit dem Dividieren meinen wir manchmal zwei verschiedene Operationen. Wenn zwei ganzzahlige Werte geteilt werden und wir wollen, dass ein ganzzahliger Wert als Ergebnis herauskommt, so können wir Teilen ohne Rest, z. B. 5 geteilt durch 2 ergibt eine 2 (DIV-Operation mit dem Symbol „/“). Oder wir können uns den Rest ausgeben lassen, indem wir fragen: *Was ist der Rest von 5 geteilt durch 2?* Antwort 1 (MOD-Operation mit dem Symbol „%“).

```
int a = 29, b, c;
b = a/10;           // DIV-Operator
c = a%10;          // MOD-Operator
```

Der Text hinter dem Doppelsymbol „//“ ist ein Kommentar und kann der Erläuterung halber, ohne Einfluss auf den Verlauf des Programms, eingefügt werden. Wir erhalten als Ergebnisse  $b=2$  und  $c=9$ . Mit dem ganzzahligen **int** können wir aber auch addieren und subtrahieren.

```
int a = 0, b = 1;
a = a + b;
b = b - 5;
a = a + 1;
```

In Java und anderen Programmiersprachen, wie z. B. C und C++, gibt es dafür eine Kurzschreibweise:

```
int a = 0, b = 1;
a += b;    // entspricht a = a + b
b -= 5;    // entspricht b = b - 5
a++;      // entspricht a = a + 1
```

### 2.3.4 byte, short, long

Die ebenfalls ganzzahligen Datentypen **byte**, **short** und **long** unterscheiden sich dem **int** gegenüber nur in der Größe der zu speichernden Werte. Alle bisherigen Operationen gelten auch für diese Datentypen. Daher schauen wir uns nur ein Beispielprogramm an, in dem die verschiedenen Datentypen verwendet werden.

```
byte b1=0, b2=100;
b1 = b2;
b2 = 100%15;

short s1=4, s2=12;
s1 -= s2;
s2 /= 2;

long a1=48659023, a2=110207203;
a1 *= a2;
```

Wir erhalten für die Variablen folgende Ergebnisse:  $b1=100$ ,  $b2=10$ ,  $s1=-8$ ,  $s2=6$ ,  $a1=5362574825542669$  und  $a2=110207203$ .

### 2.3.5 float, double

Die beiden primitiven Datentypen `float` und `double` repräsentieren gebrochene Zahlen oder Gleitkommazahlen. Die Operationen sind im Prinzip die gleichen, nur die beiden Teilungsoperatoren „/“ und „%“ (siehe 2.3.3) verhalten sich anders. Bei dem DIV-Operator ist das Ergebnis eine Gleitkommazahl, deren Genauigkeit dem entsprechend gewählten Datentyp `float` oder `double` entspricht. Das gleiche gilt für den MOD-Operator, der in den meisten Fällen aber Rundungsfehler liefert. Schauen wir uns dazu ein Beispiel an:

```
double a = 2;      // ganze Zahl wird double zugewiesen
double b = 2.2;    // 2.2 wird als double interpretiert

float c = 3;       // ganze Zahl wird float zugewiesen
float d = 3.3f;   // das f signalisiert den Float

float e, f;
e = d%c;

e = d/c;
f = c*(d/2.4f)+1;
f += 1.3f;
```

Jede ganze Zahl ist problemlos einem `float` oder `double` zuweisbar. Anders sieht es bei gebrochenen Zahlen aus. In Zeile 2 ist die Zuweisung des Wertes `2.2` an einen `double` möglich, da die `2.2` ebenfalls als `double` interpretiert wird. Anders ist es bei der Zuweisung zu einem `float`. Hier müssen wir dem Compiler<sup>1</sup> explizit sagen, dass es sich um einen `float`-Wert handelt, dazu hängen wir ein „`f`“ an die Zahl. Das müssen wir tun, da es möglicherweise zu einem Datenverlust kommen kann, da ein `double`-Wert, wie wir es aus Abschnitt 2.1 wissen, größere Zahlen repräsentieren kann.

Die Problematik des MOD-Operators wird in Zeile 8 deutlich. Erwarten würden wir als Ergebnis für `e=3.3%3` den Rest `e=0.3`. Java errechnet aber `e=0.29999995`. Es handelt sich um einen Rundungsfehler. Die übrigen Zeilen zeigen, wie die anderen Operatoren verwendet werden können.

## 2.4 Casting, Typumwandlungen

Es ist oft notwendig, den Wert eines primitiven Datentyps zu kopieren. Dabei können die Werte auch zwischen unterschiedlichen primitiven Datentypen ausgetauscht werden. Bei zwei gleichen Datentypen ist das kein Problem:

---

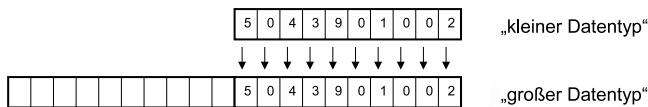
<sup>1</sup> Ein Compiler ist ein Programm, das den Programmcode in einen Maschinencode umwandelt. Bei Java sieht die Sache etwas anders aus. Der Programmcode wird in Bytecode konvertiert und eine virtuelle Maschine führt dann die Befehle auf dem entsprechenden Computer aus. Daher ist Java auch plattformunabhängig und langsamer als C und C++.

```
int a, b;
a = 5;
b = a;
```

Wenn wir aber den Wert an einen anderen Datentyp übergeben wollen, dann können zwei Fälle auftreten.

### Fall 1)

Wir haben einen kleinen Datentyp und wollen den Inhalt in einen größeren schreiben.



Hier ein Programmbeispiel dazu:

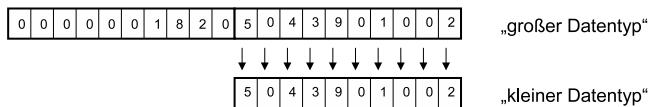
```
byte a = 28;
int b;
b = a;

float f = 2.3f;
double d;
d = f;
```

Das funktioniert ohne Probleme, denn wir haben in dem großen Datentyp mehr Speicher zur Verfügung. Daten können nicht verloren gehen.

### Fall 2)

Problematischer ist es, den Inhalt eines größeren Datentyps in einen kleineren zu kopieren.



Dabei kann es zu Datenverlust kommen!

Auch hier ein Programmbeispiel, das aber einen Fehler hervorruft:

```
float f;
double d = 2.3;
f = d;
```



Wir wissen, dass in unserem Beispiel der Wert 2.3 problemlos in ein float passt, aber der Compiler traut der Sache nicht und gibt einen Fehler aus.

Es gibt aber eine Möglichkeit, diesen Kopiervorgang zu erzwingen. Das nennen wir **Casting**. Dem Compiler wird damit gesagt, dass wir wissen, dass

- a) Daten verloren gehen können und es uns nicht wichtig ist, oder
- b) der Inhalt in dem neuen Datentyp untergebracht werden kann.

Dazu schreiben wir in Klammern vor die Zuweisung den neuen Datentyp (explizite Typumwandlung):

```
float f;
double d = 2.3;
f = (float)d;
```

Nun gibt sich der Compiler zufrieden. In den Übungsaufgaben werden wir sehen, welche Informationen beim Casten verloren gehen können.

#### 2.4.1 Übersicht zu impliziten Typumwandlungen

Es gelten für die Zahlentypen folgende Größenverhältnisse:

```
byte < short < int < long
float < double
```

Schauen wir uns dazu folgendes Beispiel an:

```
byte a = 1;
short b;
int c;
long d;
float e;
double f;

b = a; // Short <- Byte
c = a; // Int   <- Byte, Short
c = b;
d = a; // Long  <- Byte, Short, Int
d = b;
d = c;
e = a; // Float <- Byte, Short, Int, Long
e = b;
e = c;
e = d;
f = a; // Double <- Byte, Short, Int, Long, Float
f = b;
f = c;
f = d;
f = e;
```

Alle diese Typumwandlungen sind vom Compiler erlaubt. Da stellt sich aber die Frage, wie der Inhalt eines long (64 BIT) ohne Datenverlust in einen float (32 BIT) kopiert werden kann. In den Übungsaufgaben werden wir diese Frage noch einmal aufgreifen.

## 2.4.2 Die Datentypen sind für die Operation entscheidend

An dieser Stelle ist es wichtig, auf einen häufig gemachten Fehler hinzuweisen. Mit dem folgenden Beispiel wollen wir eine Division mit zwei int-Werten formulieren und das Ergebnis als double ausgeben.

```
int a=5, b=7;
double erg = a/b;
System.out.println("Ergebnis(double) von "+a+"/"+b+" = "+erg);
```



Wir erwarten eigentlich einen Wert um 0.71 Aber als Ausgabe erhalten wir:

```
C:\JavaCode>java Berechnung
Ergebnis(double) von 5/7 = 0.0
```

Der Grund für das Ergebnis 0.0 ist der Tatsache geschuldet, dass der DIV-Operator für zwei int-Werte nur den ganzzahligen Wert ohne Rest zurückgibt. Sollte einer der beiden Operatoren ein double sein, so wird die Funktion ausgeführt, die auf mehrere Stellen nach dem Komma den richtigen Wert für die Division berechnet. Der Datentyp des zweiten Operanden spielt dann keine Rolle, da er ebenfalls als double interpretiert wird. Uns genügt an dieser Stelle also lediglich, einen der beiden Operanden in einen double umzuwandeln.

Wenn wir die Programmzeilen wie folgt ändern, dann wird das erwartete Ergebnis geliefert.

```
int a=5, b=7;
double erg = (double)a/b;
System.out.println("Ergebnis(double) von "+a+"/"+b+" = "+erg);
```

Als Ausgabe erhalten wir:

```
C:\JavaCode>java Berechnung
Ergebnis(double) von 5/7 = 0.7142857142857143
```

## 2.5 Methoden der Programmerstellung

Um ein Programm zu erstellen, muss man verstehen, dass die Reihenfolge der Anweisungen, die man an den Rechner stellt, sehr entscheidend ist. Ein Programm ist wie ein Kochrezept zu verstehen. Nehmen wir als Beispiel folgende leckere Vorspeise:



Für die Zubereitung sind nun verschiedene Arbeiten auszuführen, bei den meisten ist die Reihenfolge entscheidend, schauen wir uns das Rezept mal an:

### *Rezept: „Birnen mit Käse gefüllt“*

Menge	Maß	Zutat
2	Stück	Birnen
$\frac{1}{2}$	Stück	Ausgepresste Zitrone
200	g	Schafskäse
3	EL	Sahne
1	Prise	Salz
1	Prise	Paprika
2	EL	Kresse

*Birnen waschen, halbieren, Kerngehäuse entfernen und in wenig Wasser mit Zitronensaft garen.*

*Inzwischen Ziegen- oder Schafskäse zerdrücken und mit Sahne verrühren, mit Salz und Pfeffer würzen. Käse in die Birnenhälften füllen und mit Kresse garnieren.*

*Variation: Roquefort mit Dosenmilch verrühren und mit etwas Kirschwasser abschmecken. In die Birnenhälften einfüllen. Mit Salatblättern, Kräutern oder Maraschino-kirschen garnieren.*

Dieses Rezept können wir nun als Programm verstehen. Versuchen wir es einmal in Pseudocode aufzuschreiben:

```

1 Wasche die Birnen
2   -> falls die Birnen noch nicht sauber sind, gehe zu 1
3 Halbiere die Birnen und entferne die Kerngehäuse
4 Gare die Birnen mit wenig Wasser und Zitronensaft in einem Topf
5 wenn Variante 1 gewünscht gehe zu 6 ansonsten zu 13
6 Variante 1:
7   Zerdrücke den Schafskäse und verrühre ihn mit Sahne
8   Würze die Käsesmasse mit Salz und Paprika
9   -> schmecke die Masse ab, falls Salz oder Paprika fehlen gehe zu 8
10  Fülle die Käsesmasse in die fertig gegarten Birnenhälften
11  Garniere die Birnenhälften mit Kresse
12  FERTIG
13 Variante 2:
14  Verrühre Roquefort mit Dosenmilch und etwas Kirschwasser
15  Fülle die Masse in die fertig gegarten Birnenhälften
16  Garniere alles mit Salatblättern, Kräutern oder Maraschinokirschen.
17  FERTIG

```

Alltägliche Abläufe von verschiedenen Prozessen kann man sich als ein Programm vorstellen. Es gibt immer wiederkehrende Vorgehensweisen. Bei genauerer Betrachtung gibt es sogar nur drei. Alle anderen Methoden sind leicht als Spezialfall dieser drei zu interpretieren. Im folgenden Abschnitt werden diese Konzepte vorgestellt, die ausreichend sind, um das Prinzip der Softwareerstellung zu verstehen. Alle üblichen Programmiersprachen lassen sich auf diese Methoden reduzieren. Anschließend werden wir in der Lage sein, die ersten Programme zu schreiben.

Das Verständnis dieser drei grundlegenden Konzepte scheint ein Problem vieler Programmieranfänger zu sein. Verinnerlicht man diese Konzepte und die Tatsache, dass die Reihenfolge entscheidend ist, dann werden die nachfolgenden Abschnitte viel besser verständlich sein.

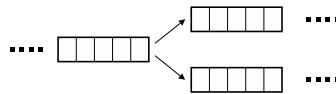
### 2.5.1 Sequentieller Programmablauf

Ein Programm kann prinzipiell Anweisung für Anweisung hintereinander weg geschrieben werden. Kein Abschnitt wird wiederholt. Das könnte z. B. eine maschinelle Qualitätsüberprüfung bei einem fertigen Produkt sein: prüfe die Größe, prüfe die Farbe, prüfe die Form, prüfe die Widerstandsfähigkeit, ... Ein weißes Kästchen steht für eine Anweisung, die das Programm auszuführen hat. Wir lesen das Programm von links nach rechts.



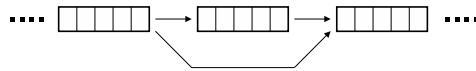
### 2.5.2 Verzweigungen

Oft kommt es aber vor, dass man eine Entscheidung treffen muss, die die Wahl der nachfolgenden Anweisungen beeinflusst. Es handelt sich dabei um eine Verzweigung. In unserem Rezeptbeispiel ist es die Wahl der Variante. Wir brauchen keinen Schafskäse, wenn Variante 2 gewählt ist.



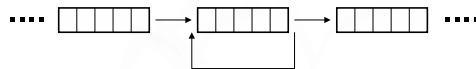
### 2.5.3 Sprünge

Sprünge sind ein Spezialfall einer Verzweigung. Wir könnten z. B. nach der Zugabe von Roquefort prüfen, ob dieser genug Salz mitgebracht hat. Wenn das nicht der Fall ist, dann geben wir noch etwas Salz hinzu, ansonsten lassen wir diesen Schritt aus.



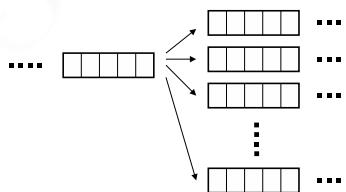
### 2.5.4 Schleifen

Schleifen sind ebenfalls ein Spezialfall der Verzweigung. Wir könnten einen bestimmten Prozess solange vom Computer wiederholen lassen, z. B. die Birnen zu putzen, bis die Sauberkeit einer bestimmten Qualitätsnorm entspricht. Oder wir möchten ganz einfach mehrmals einen bestimmten Programmteil ausführen und wollen nicht die Eingabe desselben Programmcodes wiederholen.



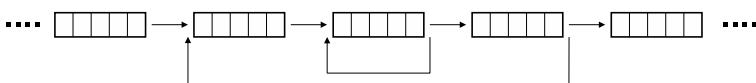
### 2.5.5 Mehrfachverzweigungen

Es gibt auch die Möglichkeit, eine Verzweigung in beliebig vielen Wegen fortzusetzen.



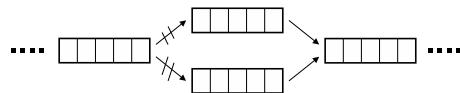
### 2.5.6 Mehrfachschleifen (verschachtelte Schleifen)

Schleifen kommen oft genug verschachtelt vor. Aber auch das sind nur Spezialfälle oder Kombinationen aus Verzweigung und sequentiellem Ablauf.



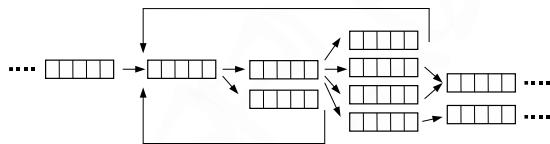
### 2.5.7 Parallelität

Das dritte wichtige Konzept, neben der sequentiellen Abfolge und dem Sprung, ist die Parallelität, das ich hier erwähnen möchte, aber das in den folgenden Abschnitten erstmal keinerlei Verwendung findet. Die Idee ist aber wichtig und bei großen Softwareprojekten kommt es vor, dass Programmteile parallel (also gleichzeitig) laufen müssen<sup>2</sup>. Zwei Striche auf den Verzweigungspfeilen sollen bedeuten, dass die beiden Wege gleichzeitig bearbeitet werden sollen und sich später wieder treffen können.



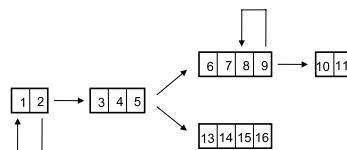
### 2.5.8 Kombination zu Programmen

Die vorgestellten Methoden können nun beliebig kompliziert verknüpft werden und ergeben in ihrer Kombination schließlich ein Programm.



Ein Programm sollte nach der Lösung einer Aufgabe **terminieren** (beenden). Das heißt, dass wir immer dafür Sorge tragen müssen, dass das Programm nicht in eine Sackgasse oder Endlosschleife gerät.

Als Beispiel für ein Programm wollen wir nun nochmal das Birnen-Rezept nehmen. Jede Zeile ist nummeriert und stellt eine Anweisung dar. In der folgenden Darstellung sind die Anweisungen durch ihre Zeilenummern repräsentiert.



Ich hoffe der Leser entwickelt ein Gefühl für diese Denkweise, dann ist das Programmieren keine Kunst mehr.

---

<sup>2</sup> Das Stichwort hier ist **Thread** [49].

## 2.6 Programme in Java

Nach den bisherigen Trockenübungen geht es nun mit den ersten Javaprogrammen los. Es gibt zahlreiche Entwicklungsumgebungen für Java. Um keine zu bevorzugen und den Leser dafür zu sensibilisieren, was „hinter den Kulissen“ geschieht, verwenden wir keine bestimmte, sondern erarbeiten unsere Programme mit einem Texteditor.

Wir könnten sogar schon die Erstellung eines Javaprogramms mit Hilfe eines Programms ausdrücken. Versuchen wir es mal in Pseudocode<sup>3</sup>:

```

1 Öffne einen Texteditor
2 Lege ein Dokument mit gewünschtem Programmnamen an
3 Schreibe Dein Programm
4 Speichere es mit der Endung ".java" in Ordner <X>
5 Gehe außerhalb des Editors in einer Konsole zu Ort <X>
6 Schreibe "javac <Programmname>.java"
7 Falls java einen Fehler ausgibt
   -> gehe zu Editor
      repariere Fehler
      springe zu Zeile 4
8 Falls java keinen Fehler ausgibt
   -> schreibe "java <Programmname>"
      falls das Programm nicht das gewünschte tut
      -> springe zu Zeile 8
      falls alles ok ist
      -> fertig und Freude
16

```

### 2.6.1 Erstellen eines Javaprogramms

Gehen wir einmal Schritt für Schritt durch, wie sich ein Java-Programm mit einem Texteditor unter Windows schreiben lässt. Wir öffnen zunächst unseren Texteditor und legen eine Datei mit dem Namen **ProgrammEins.java** an. Der Name der Datei MUSS mit dem Namen des Programms, das wir schreiben, übereinstimmen.



```

1 public class ProgrammEins{
2     public static void main(String[] args){
3         System.out.println("Endlich ist es soweit! Mein erstes Programm
4                         läuft... ");
5     }
6 }

```

Wir speichern in diesem Beispiel das Programm in das Verzeichnis „C:\Java“. Uns ist im Programm ein kleiner Fehler unterlaufen, den wir gleich sehen werden. Jetzt wechseln wir in ein Konsolenfenster und gehen in den Ordner, in dem unser Programm gespeichert ist. Anschließend kompilieren wir es mit **javac ProgrammEins.java** und hoffen, dass der Java-Compiler im Programm keinen Fehler findet.

---

<sup>3</sup> Mit Pseudocode bezeichnen wir die Notation eines Programms in keiner speziellen Programmiersprache, sondern einer allgemeineren Form. Es können ganze Sätze verwendet werden, um Anweisungen zu beschreiben.

```
C:\Java>javac ProgrammEins.java
ProgrammEins.java:1: class ProgrammEins is public, should be declared
in a file named ProgramEins.java
public class ProgrammEins{
          ^
1 error
```

Leider gibt es einen. Die Fehlermeldung sagt uns, dass wir der Klasse **ProgramEins** auch den entsprechenden Dateinamen geben müssen. Wir haben ein 'm' vergessen. Wir wechseln wieder in den Editor und ändern den Namen der Klasse. Nun entspricht er dem Dateinamen und wir dürfen nicht vergessen zu speichern. Ansonsten hätte unsere Veränderung keinen Einfluss.

```
1 public class ProgrammEins{
2     public static void main(String[] args){
3         System.out.println("Endlich ist es soweit! Mein erstes Programm
4                         läuft...");
```

```
5     }
6 }
```

Ein zweiter Versuch liefert keine Fehlermeldung. Demzufolge scheint das Programm **syntaktisch**, also von der äußerlichen Form her, korrekt zu sein und es kann gestartet werden:

```
C:\Java>javac ProgrammEins.java
C:\Java>java ProgrammEins
Endlich ist es soweit! Mein erstes Programm läuft ...
```

Ein erster Erfolg. Wir sehen noch einen kleinen Makel. Bei der Ausgabe wurde das Zeichen 'ä' durch ein anderes ersetzt. Der Grund dafür ist, dass Java mit einem internationalen Zeichensatz arbeitet und 'ä' zu den deutschen Sonderzeichen gehört.

## 2.7 Zusammenfassung und Aufgaben

### Zusammenfassung

Wir haben die kleinsten Bausteine eines Programms kennengelernt. Java bietet boolean, char, byte, short, int, long, float und double. Die arithmetischen Operationen auf den Zahlen +, -, \*, /, % und booleschen Operationen mit Wahrheitswerten &&, ||, ! wurden erläutert. Größere Datentypen können wir in kleinere konvertieren und kennen die abstrakten Methoden der Programmerstellung: sequentieller Programmablauf, Verzweigung, Sprung, Schleife, Mehrfachverzweigung, Mehrfachschleife und Parallelität. Im letzten Abschnitt haben wir zum ersten Mal ein Javaprogramm kompiliert und gestartet.

## Aufgaben

Übung 1) Überlegen Sie sich Fälle, bei denen ein Programm nicht terminiert. Verwenden Sie als Beschreibung die Konzepte aus Abschnitt 2.5.

Übung 2) Welche der folgenden Variablenamen sind ungültig?

Norbert, \$eins, \_abc123, #hallihallo, erne\$to, const, int, 1a, gRoSS, k\_1-e\_i-n, %nummer, Class, klasse, !wahr, final, blablubs

Übung 3) Was ist der Unterschied zwischen: `a=b` und `a==b`?

Übung 4) Welchen Datentyp und welche Bezeichnung würden Sie für die folgenden Informationen vergeben:

- i) das Alter Ihrer Familienmitglieder
- ii) den Geldbetrag Ihres Kontos
- iii) die Fläche eines Fußballfeldes in  $cm^2$
- iv) Anzahl der unterschiedlichen Positionen im Schach
- v) die durchschnittliche Anzahl der Kinder in Deutschland pro Paar

Übung 5) Deklarieren Sie Variablen und weisen ihnen Werte zu. Geben Sie eine Variable `c` an, die die Funktion aus folgender Wertetabelle berechnet:

$B_1$	$B_2$	$(B_1 \text{ AND } B_2) \text{ OR } (\text{NICHT } B_2)$
0	0	1
0	1	0
1	0	1
1	1	1

Übung 6) Werten Sie die folgenden Programmzeilen aus und geben Sie die Werte von `c`, `d`, `e`, `f` und `g` an:

```
boolean a=true, b=false, c, d, e, f, g;
c = a ^ b;
d = !a || b;
e = (d && !c) || !a;
f = ((d == e) || (d != e)) == true;
g = 5==7;
```

Übung 7) Gehen Sie die einzelnen Schritte aus Abschnitt 2.6.1 durch und bringen Sie das Programm **ProgrammEins.java** zum Laufen.

Übung 8) Geben Sie ein Javaprogramm an, das die folgenden kleinen Berechnungen für Sie ausführt:

- i) Gegeben sei ein Rechteck mit den Seitenlängen  $a=5$  und  $b=4$ . Geben Sie die Variable `c` an, die die Fläche des Rechtecks berechnet.
- ii) Wenn 12 Äpfel 22% entsprechen, wie viele Äpfel gibt es dann insgesamt?

Übung 9) Gegeben sei folgender Programmabschnitt, welchen Wert hat b am Ende?

```
boolean b;  
int a=7, c=22, d;  
d=(c/a)*2;  
b=((c%a)<=(c/a))&&(d==6)
```

# 3

---

## Tag 3: Programmieren mit einem einfachen Klassenkonzept

Wir wollen mit der Programmierung in Java beginnen, ohne vorher viel Theorie zum Thema Klassen und Vererbung zu studieren. Jedes von uns geschriebene Programm sehen wir erstmal als eine Klasse an. Später in Kapitel 7 werden wir dann verstehen, warum wir von **Klassen** sprechen und lernen verschiedene Methoden kennen, die uns helfen werden, systematische und wiederverwendbare Programme zu schreiben.

Das folgende Programm tut erstmal herzlich wenig, soll aber zeigen, welche Zeilen wir zum Schreiben eines Java-Programms benötigen. Das Einrücken der Zeilen innerhalb eines Blocks, so wird der Abschnitt zwischen „{“ und „}“ genannt, dient der Lesbarkeit eines Programms.

```
1 public class MeinErstesProgramm{
2     public static void main(String [] args){
3         //HIER KOMMEN DIE ANWEISUNGEN DES PROGRAMMS HIN
4     }
5 }
```

In *Zeile 1* geben wir unserem Programm einen Namen, z. B. `MeinErstesProgramm`. Der Name ist ein Wort, bestehend aus Groß- und Kleinbuchstaben. Begonnen wird laut Konvention aber immer mit einem Großbuchstaben. Dann speichern wir dieses Programm mit dem Namen „`MeinErstesProgramm.java`“. Das ist sehr wichtig, da Java das Programm sonst nicht finden kann.

In *Zeile 2* sehen wir eine Funktion mit dem Namen `main`. Was eine Funktion ist, warum in Klammern dahinter `String[] args` steht und was die anderen Wörter bedeuten, werden wir später verstehen. Im Moment ist es wichtig, dass wir mit der Programmierung beginnen.

In *Zeile 3* wird nun das eigentliche Programm, also die Anweisungen, geschrieben. Wenn das Programm gestartet wird, werden diese Anweisungen ausgeführt und nach Ausführung aller Anweisungen beendet sich das Programm. Die dritte Zeile beginnt mit „//“. Das bedeutet, dass ein Kommentar folgt und Java den Rest dieser Zeile bitte ignorieren soll. Sie dient lediglich dem Programmierer, der sich so Notizen machen

kann. Es ist nicht immer einfach, ein langes Programm ohne hilfreiche Kommentare zu verstehen. Dies trifft sogar auf selbst geschriebene Programme zu, erst recht auf die von anderen. Daher sind gerade in Projekten, bei denen mehrere Programmierer zusammenarbeiten, Kommentare unverzichtbar.

Es gibt zwei Möglichkeiten in Java Kommentare zu schreiben:

```

1 // Programm: Kommentierung.java
2
3 // Ich bin ein hilfreicher Kommentar in einer Zeile
4
5 /* Falls ich mal Kommentare über mehrere Zeilen
6    hinweg schreiben möchte, so kann ich das
7    mit diesen Kommentarsymbolen tun
8 */
9
10 public class Kommentierung{      // ich kann auch hier stehen
11     public static void main(String[] args){
12
13         /* An diese Stelle schreiben wir die
14            Programmanweisungen */
15
16     }
17 }
```

### 3.1 Sequentielle Anweisungen

Wir betrachten jetzt den eigentlichen Programmabschnitt. Eine Anweisung ist ein Befehl und Anweisungen werden sequentiell, also hintereinander ausgeführt. Nach einer Anweisung steht das Symbol „;“. Damit schließen wir immer eine Anweisung ab.

```

1 // Programm: Sequentiell.java
2 public class Sequentiell{
3     public static void main(String[] args){
4         int a=5;      // Anweisung 1
5         a=a*2;      // Anweisung 2
6         a=a+10;     // Anweisung 3
7         a=a-5;      // Anweisung 4
8     }
9 }
```

Unser Programm **Sequentiell** deklariert eine neue Variable **a** vom Typ **int** und initialisiert sie mit dem Wert 5. Das ist unsere erste Anweisung an das Programm. Anschließend soll **a** mit 2 multipliziert und das Ergebnis wieder in **a** gespeichert, **a** also überschrieben werden. Nach der vierten Anweisung sollte **a** den Wert 15 haben. Um das zu überprüfen, geben wir folgende Zeile als fünfte Anweisung dazu:

```
System.out.println("a hat den Wert: "+a);
```

`System.out.println()` ist eine Anweisung, die in dem Konsolenfenster eine Textzeile ausgibt. Der Inhalt der Variable `a` wird an den String „`a` hat den Wert: “ gehängt und ausgegeben. In diesem Beispiel haben wir die erste Zeichenkette verwendet. Später werden wir noch einmal darauf zu sprechen kommen und Methoden zur Manipulation von Zeichenketten kennen lernen. Testen wir das mal auf der Kommandozeile:

```
C:\>javac Sequentiell.java
C:\>java Sequentiell
a hat den Wert: 15
```

Wunderbar. Dann haben wir uns also nicht verrechnet, `a` hat den Wert 15.

## 3.2 Verzweigungen

In Abschnitt 2.5 haben wir gesehen, dass es Verzweigungen gibt. Die Syntax für eine Verzweigung kann verschieden formuliert werden. Zunächst schauen wir uns die am häufigsten verwendete Verzweigung `if` an. In den meisten Fällen geht es darum, ob eine Bedingung erfüllt ist oder nicht. Wir haben demnach eine Straßenkreuzung mit zwei Wegen vor uns und entscheiden, welchem wir folgen.

Die zweite Verzweigungsmöglichkeit `switch` kann als mehrspurige Autobahn angesehen werden und wir entscheiden, auf welcher der vielen Spuren wir nun weiterfahren möchten.

### 3.2.1 if-Verzweigung

Wenn eine Bedingung erfüllt ist, führt eine einzelne Anweisung aus:

```
if (<Bedingung>) <Anweisung>;
```

Es können aber auch mehrere Anweisungen, also ein ganzer Block von Anweisungen, ausgeführt werden. Dazu verwenden wir die geschweiften Klammern. Das ist für die Folgebeispiele ebenfalls immer eine Alternative. Anstatt eine Anweisung auszuführen, kann ein ganzer Anweisungsblock ausgeführt werden.

```
if (<Bedingung>){
    <Anweisungen>;
}
```

Eine Erweiterung dazu ist die `if-else`-Verzweigung. Wenn eine Bedingung erfüllt ist, führe eine Anweisung 1 aus. Wenn diese Bedingung nicht erfüllt ist, dann führe Anweisung 2 aus. In jedem Fall wird also eine der beiden Anweisungen ausgeführt, denn die Bedingung ist entweder erfüllt (`true`) oder nicht (`false`).

```
if (<Bedingung>) <Anweisung1>;
else <Anweisung2>;
```

Jetzt können wir auch verschiedene Bedingungen verknüpfen. Wenn Bedingung 1 erfüllt ist, führe Anweisung 1 aus. Falls nicht, dann prüfe, ob Bedingung 2 erfüllt ist. Wenn ja, dann führe Anweisung 2 aus, wenn nicht, dann mache weder Anweisung 1 noch Anweisung 2.

```
if (<Bedingung1>) <Anweisung1>;
else if (<Bedingung2>) <Anweisung2>;
```

Mit der `if`-Verzweigung können wir immer nur eine Bedingung überprüfen und anschließend in zwei Wegen weiter gehen. Diese können wir aber beliebig verschachteln.

### 3.2.2 switch-Verzweigung

Um nun Mehrfachverzweigungen zu realisieren und nicht zu einer unübersichtlichen Flut von `if`-Verzweigungen greifen zu müssen, steht uns die `switch`-Verzweigung zur Verfügung. Leider gibt es für die Verwendung ein paar Einschränkungen, so können wir nur Bedingungen der Form: *Ist int a == 4?*, usw. überprüfen. Die Syntax sieht wie folgt aus.

```
switch (<Ausdruck>) {
    case <Konstante1>:
        <Anweisung1>;
        break;
    case <Konstante2>:
        <Anweisung2>;
        break;
    default:
        <Anweisung3>;
}
```

Als Ausdruck lassen sich verschiedene primitive Datentypen auf Ihren Inhalt untersuchen. Zu den erlaubten gehören: `char`, `byte`, `short` und `int`. Dieser Ausdruck, also eine Variable dieses Typs, wird mit den Konstanten verglichen.

<Ausdruck> könnte z. B. ein int a sein und wir möchten den Fall 1 anwenden, wenn a=3 erfüllt ist. Die <Konstante1> wäre dann eine 3.

Ein Beispiel wäre jetzt angebracht<sup>1</sup>:

```
for (int i=0; i<5; i++){
    switch(i){
        case 0:
            System.out.println("0");
            break;
        case 1:
        case 2:
            System.out.println("1 oder 2");
            break;
        case 3:
            System.out.println("3");
            break;
        default:
            System.out.println("hier landen alle anderen... ");
            break;
    }
}
```

Die switch-Verzweigung wird fünfmal mit den Zahlen  $i=0, 1, 2, 3, 4$  für eine Verzweigung verwendet. Bei der Ausführung dieser Zeilen erzeugen wir folgende Ausgaben:

```
C:\Java>java Verzweigung
0
1 oder 2
1 oder 2
3
hier landen alle anderen...
```

Für  $i=0$  wird der erste Fall **case 0:** ausgeführt und mit dem **break** signalisiert, dass die switch-Verzweigung beendet wird. **case 1:** und **case 2:** führen für  $i=1, 2$  zu derselben Ausgabe, denn nur ein **break** beendet ein **case**.  $i=3$  verhält sich analog zu  $i=0$ . Für alle anderen Fälle, also Werte für  $i$ , die nicht durch ein **case i:** abgedeckt sind, tritt der **default**-Block in Kraft.

### 3.3 Verschiedene Schleifentypen

Wir haben folgendes Programm geschrieben, das eine Zahl quadriert und das Ergebnis ausgibt:

---

<sup>1</sup> In der *ersten Zeile* dieses Beispiels haben wir bereits den Schleifentyp **for** verwendet, der im nächsten Abschnitt erläutert wird. Ich wollte aber die Reihenfolge **if**, **switch** und Schleifen aus Abschnitt 2.5 beibehalten.

```
System.out.println("1 zum Quadrat ist " +(1*1));
System.out.println("2 zum Quadrat ist " +(2*2));
System.out.println("3 zum Quadrat ist " +(3*3));
System.out.println("4 zum Quadrat ist " +(4*4));
System.out.println("5 zum Quadrat ist " +(5*5));
System.out.println("6 zum Quadrat ist " +(6*6));
```

Wir sehen schon, dass es mühsam ist, jedes Mal die gleiche Anweisung aufzuschreiben. Es wäre sicherlich auch eine unangenehme Aufgabe, alle ganzen Zahlen zwischen 1 und 1000 auf diese Weise quadrieren und ausgeben zu lassen. Daher ist das Konzept der Schleifen unverzichtbar.

Angenommen, wir möchten die Aufgabe lösen, alle Quadrate der Zahlen zwischen 1 und 1000 auszugeben. Dann könnten wir das in Pseudocode in etwa so ausdrücken:

```
1 Beginne bei i=1
2 Quadriere i und gib das Ergebnis aus
3 Falls i <=1000
4   -> Erhöhe i um 1 und springe zu Zeile 2
```

Oder in Worten: Starte bei  $i=1$  und solange  $i \leq 1000$  ist, mache folgendes: quadriere  $i$  und gib den Wert aus. Erhöhe  $i$  um eins und mache weiter.

### 3.3.1 for-Schleife

Eine for-Schleife benötigt eine Variable, die zu Beginn mit einem Startwert initialisiert wird und führt den nachfolgenden Anweisungsblock solange aus, erhöht oder verringert dabei die Variable um eine konstante Schrittgröße, bis die Bedingung nicht mehr erfüllt ist:

```
for (<Startwert>; <Bedingung>; <Schrittweite>) {
    <Anweisung>;
}
```

Im Programm könnten wir das zu Beginn geschilderte Problem mit einer for-Schleife so lösen:

```
for (int i=1; i<=1000; i=i+1){
    System.out.println(i+" zum Quadrat ist "+(i*i));
}
```

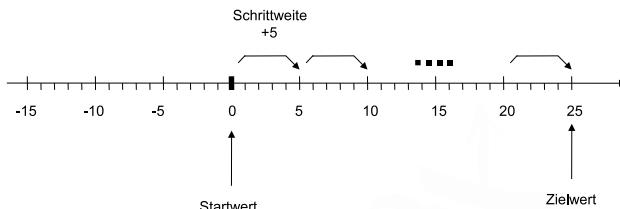
Mit `int i=1` geben wir einen Startwert vor. Die Schleife führt die Anweisungen innerhalb der geschweiften Klammern solange aus und erhöht jedes Mal  $i$  um 1 bis die Bedingung  $i \leq 1000$  nicht mehr erfüllt ist. Es sei dem Leser überlassen, diesen Programmabschnitt einmal auszuprobieren.

Wir können **for**-Schleifen auch für andere Probleme verwenden, es gilt immer:

```

1 for (Startwert;
2     Bedingung die erfüllt sein muss, damit es weiter geht;
3     Schrittweite)
4 {
5     Anweisungen
6 }
```

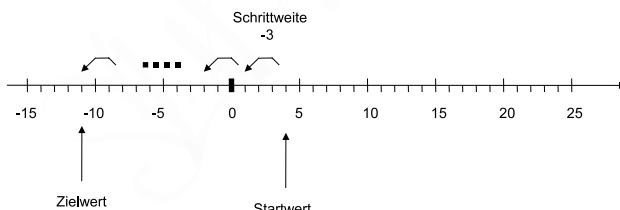
Beispielsweise könnten wir bei 0 anfangen, in Fünferschritten weiterlaufen und alle Werte bis einschließlich 25 erzeugen:



```

for (int i=0; i<=25; i=i+5){
    System.out.println("Aktueller Wert für i ist "+i);
}
```

Oder bei 4 beginnend immer 3 abziehen, bis wir bei -11 angelangt sind:



```

for (int i=4; i>=-11; i=i-3){
    System.out.println("Aktueller Wert für i ist "+i);
}
```

### 3.3.2 while-Schleife

Es gibt neben der **for**-Schleife noch weitere Möglichkeiten, Schleifen zu konstruieren. Manchmal ist es nicht klar, wie viele Schleifendurchläufe benötigt werden, um ein Ergebnis zu erhalten. Da wäre es schön, eine Möglichkeit zu haben, wie: Wiederhole Anweisungen solange, bis eine Bedingung erfüllt ist. Genau diesen Schleifentyp repräsentiert die **while**-Schleife.

```
while (<Bedingung>) {
    <Anweisung>;
}
```

Wir werden das Beispiel aus der `for`-Schleife wieder aufnehmen und sehen, wie das mit `while` gelöst werden kann:

```
int i=1;
while (i<=1000){
    System.out.println(i+" zum Quadrat ist "+(i*i));
    i=i+1;
}
```

Die Schleife wird einfach solange ausgeführt, bis die Bedingung hinter `while` nicht mehr erfüllt ist.

```
1 while (Bedingung die erfüllt sein muss, damit es weiter geht) {
2     Anweisungen
3 }
```

Hier ist natürlich auch die Gefahr gegeben, dass es unendlich lang weiter geht, wenn man aus Versehen eine Endlosschleife, wie in diesem Beispiel, konstruiert:



```
int i=1;
while (i<=1000){
    System.out.println(i+" zum Quadrat ist "+(i*i));
}
```

Wir haben einfach vergessen, `i` zu erhöhen.

Abschließend können wir zu der `while`-Schleife sagen, dass zunächst eine Bedingung erfüllt sein muss, bevor das erste Mal die nachfolgende Anweisung ausgeführt wird. Wenn diese Bedingung nicht erfüllt ist, überspringen wir den Abschnitt und machen an anderer Stelle weiter.

### 3.3.3 do-while-Schleife

Die `do-while`-Schleife führt im Gegensatz zur `while`-Schleife zuerst den Anweisungsblock einmal aus und prüft anschließend die Bedingung.

```
do {
    <Anweisung>;
} while (<Bedingung>);
```

Schauen wir uns dazu mal folgendes Beispiel an:

```
int i=0;
do {
    i++;
    System.out.println("Wert von i: "+i);
} while (i<5);
```

Wir erhalten folgende Ausgabe:

```
C:\Java>java Schleifen
Wert von i: 1
Wert von i: 2
Wert von i: 3
Wert von i: 4
Wert von i: 5
```

Wenn wir die Bedingung so ändern, dass sie von vornherein nicht erfüllt ist,

```
int i=0;
do {
    i++;
    System.out.println("Wert von i: "+i);
} while (i<0);
```

geschieht trotzdem das Folgende:

```
C:\Java>java Schleifen
Wert von i: 1
```

Das war auch zu erwarten, da die Überprüfung der Bedingung erst nach der ersten Ausführung stattfindet.

## 3.4 Sprunganweisungen

Manchmal ist es notwendig, eine Schleife vorzeitig zu beenden. Zum Beispiel indem die Zählvariable einer for-Schleife innerhalb der Schleife auf einen Wert gesetzt wird, der die Bedingung zum Weiterlaufen nicht mehr erfüllt. Das ist aber sehr unschön, weil man nun nicht mehr so einfach erkennen kann, wann die Schleife verlassen wird. Daher gibt es Sprunganweisungen.

### 3.4.1 break

Der einfachste Sprungbefehl ist break. Diese Anweisung schließt nicht nur die case-Fälle bei switch, sondern beendet eine while-, do-while-, und for-Schleife auch unmittelbar.

```

int wert;
for (int i=0; i<=100; i++){
    wert = (i*i) + 2;
    System.out.println("Wert i=" + i + ", Funktionswert von i=" + wert);
    if (wert>100)
        break;
}

```

In diesem Beispiel berechnen wir für  $i$  die Funktion  $f(i) = i^2 + 2$ . Wir wollen alle  $i$  von 0 bis 100 durchlaufen, aber aufhören, sobald  $f(i) > 100$  oder  $i > 100$  ist.

```

C:\Java>java Spruenge
Wert i=0, Funktionswert von i=2
Wert i=1, Funktionswert von i=3
Wert i=2, Funktionswert von i=6
Wert i=3, Funktionswert von i=11
Wert i=4, Funktionswert von i=18
Wert i=5, Funktionswert von i=27
Wert i=6, Funktionswert von i=38
Wert i=7, Funktionswert von i=51
Wert i=8, Funktionswert von i=66
Wert i=9, Funktionswert von i=83
Wert i=10, Funktionswert von i=102

```

Wir haben die `for`-Schleife einfach abgebrochen und sind ans Ende gesprungen, um dort weiterzumachen. Bei verschachtelten Schleifen wird immer die aktuelle innere Schleife beendet. Wir können aber noch einen zweiten Sprung realisieren. In Programmen lassen sich Marken unterbringen („markierte Anweisungen“ [1]). Diese Marken werden mit folgender Syntax angegeben:

<Marke>:

Wenn hinter `break` eine Marke steht, dann springt das Programm zu der Marke und arbeitet dort sequentiell weiter.

`break <Marke>;`

Die Idee besteht darin, mehrere Schleifen zu beenden und mit einer bestimmten weiterzumachen. Hier ein auf den ersten Blick etwas kompliziert aussehendes Beispiel dazu:

```

SprungZuI:           // Sprungmarke
for (int i=0; i<=2; i++){
    System.out.println("... jetzt sind wir hier bei i");
    SprungZuJ:           // Sprungmarke
    for (int j=0; j<=2; j++){
        System.out.println("... jetzt sind wir hier bei j");
        for (int k=0; k<=2; k++){
            System.out.println("... jetzt sind wir hier bei k");
            if (k==1)
                break SprungZuI;
        }
    }
System.out.println("hier sind wir...");

```

Wir springen zur Sprungmarke **SprungZuI**. Das bewirkt folgendes:

```
C:\Java>java Spruenge
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
hier sind wir ...
```

Jetzt ändern wir das Programm so, dass wir zu der Sprungmarke **SprungZuJ** springen.

```
SprungZuI:           // Sprungmarke
for (int i=0; i<=2; i++){
    System.out.println("... jetzt sind wir hier bei i");
    SprungZuJ:           // Sprungmarke
    for (int j=0; j<=2; j++){
        System.out.println("... jetzt sind wir hier bei j");
        for (int k=0; k<=2; k++){
            System.out.println("... jetzt sind wir hier bei k");
            if (k==1)
                break SprungZuJ;
        }
    }
}
System.out.println("hier sind wir...");
```

Nun muss der Computer ein wenig mehr arbeiten:

```
C:\Java>java Spruenge
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
... jetzt sind wir hier bei i
... jetzt sind wir hier bei j
... jetzt sind wir hier bei k
... jetzt sind wir hier bei k
hier sind wir ...
```

Da wir in die erste Schleife reinspringen, wird diese für alle noch ausstehenden *i* ausgeführt. Falls der Leser hier Schwierigkeiten beim Nachvollziehen hat, so weise ich darauf hin, dass in den Übungsaufgaben noch einmal darauf eingegangen wird. Spätestens beim Lösen und Ausprobieren der Übungsaufgaben ist alles klar.

### 3.4.2 continue

Die Anweisung `continue` beendet nicht die aktuelle innere Schleife wie `break`, sondern springt zum Start der Schleife zurück, verändert entsprechend die Variable um die angegebene Schrittweite und setzt die Arbeit fort.

Schauen wir uns dazu ein Beispiel an:

```

1 public class Sprunganweisungen{
2     public static void main(String[] args){
3         for (int i=0; i<3; i++){
4             System.out.println("Schleife i="+i+", Code 1");
5             for (int j=0; j<3; j++){
6                 System.out.println("    Schleife j="+j+", Code 1");
7                 if (j==1){
8                     System.out.println("        continue-Anweisung");
9                     continue;
10                }
11                System.out.println("    Schleife j="+j+", Code 2");
12            }
13            if (i==1){
14                System.out.println("continue-Anweisung");
15                continue;
16            }
17            System.out.println("Schleife i="+i+", Code 2");
18        }
19    }
20 }
```

Wenn wir das Programm starten, erhalten wir sehr anschaulich ein Beispiel für die Funktionalität von `continue`:

```
C:\Java>java Sprunganweisungen
Schleife i=0, Code 1
    Schleife j=0, Code 1
    Schleife j=0, Code 2
    Schleife j=1, Code 1
    continue-Anweisung
    Schleife j=2, Code 1
    Schleife j=2, Code 2
Schleife i=0, Code 2
Schleife i=1, Code 1
    Schleife j=0, Code 1
    Schleife j=0, Code 2
    Schleife j=1, Code 1
    continue-Anweisung
    Schleife j=2, Code 1
    Schleife j=2, Code 2
continue-Anweisung
Schleife i=2, Code 1
    Schleife j=0, Code 1
    Schleife j=0, Code 2
    Schleife j=1, Code 1
    continue-Anweisung
    Schleife j=2, Code 1
    Schleife j=2, Code 2
Schleife i=2, Code 2
```

Auch die Anweisung `continue` lässt sich mit einer Sprungmarke versehen. Der Aufruf ist identisch zu dem im Abschnitt `break` beschriebenen Beispiel.

## 3.5 Klassen

Angenommen wir haben eine schöne Ausgabe<sup>2</sup> für Zahlen programmiert und möchten nach jedem Berechnungsschritt diese Ausgabe ausführen. Momentan würden wir es noch so schreiben:

```

1 // Ausgabe.java
2 public class Ausgabe{
3     public static void main(String [] args){
4         int a=4;
5
6         System.out.println();
7         System.out.println("***** Wert der Variable ist "+a);
8         System.out.println("***** Wert der Variable ist "+a);
9         System.out.println();
10
11        a=(a*13)%12;
12
13
14        System.out.println();
15        System.out.println("***** Wert der Variable ist "+a);
16        System.out.println("***** Wert der Variable ist "+a);
17        System.out.println();
18
19        a+=1000;
20
21
22        System.out.println();
23        System.out.println("***** Wert der Variable ist "+a);
24        System.out.println("***** Wert der Variable ist "+a);
25        System.out.println();
26    }
27 }

```

Um Redundanz zu vermeiden und die Programmzeilen nur einmal aufzuschreiben, lagern wir diese Zeilen in eine Funktion aus. In unserem einfachen Verständnis von Klassen schreiben wir vor jeder Funktion `public static`. Später werden wir erfahren, was es damit auf sich hat und welche Alternativen es gibt.

### 3.5.1 Funktionen in Java

Funktionen repräsentieren einen Programmabschnitt, der einmal formuliert beliebig oft aufgerufen und verwendet werden kann. Eine Funktion<sup>3</sup> erhält dabei einen eindeutigen Namen (beginnend mit einem Kleinbuchstaben) und kann Parameter entgegen nehmen. Dazu schreiben wir hinter dem Programmnamen in Klammern die zu erwartenden Datentypen und vergeben Namen. Diese Variablen gelten zunächst nur innerhalb der Methode (daher werden sie auch lokale Variablen genannt) auch wenn

<sup>2</sup> Über Schönheit lässt sich anderer Meinung sein. Hier dient es lediglich der Einführung in die Arbeit mit Funktionen.

<sup>3</sup> Wir machen hier keine Unterscheidung zwischen dem Begriff **Funktion** und **Methode**.

in der `main`-Methode eine Variable mit dem gleichen Namen existiert, haben diese beiden nichts miteinander zu tun.

Aus der Mathematik wissen wir, dass Funktionen auch ein Ergebnis liefern. Genau eines. Auch für unsere Funktionen gilt das. Falls, wie in unserem Fall, kein Rückgabewert existiert, dann schreiben wir als Rückgabewert das Schlüsselwort `void`.

```
public static <Rückgabewert> Funktionsname (Parameter) {
    // Funktionskörper
}
```

`main` ist auch eine Funktion, die es in unserem Programm einmal gibt und mit z. B. `java Ausgabe` genau einmal aufgerufen wird.

In unserem Ausgabebeispiel ist `a` ein Eingabeparameter für die neue Funktion. Die Funktion schreiben wir laut Konvention über die `main`-Funktion. Für die Lesbarkeit des Programms sind sprechende Namen, in unserem Beispiel `gebeAus`, unerlässlich.

Wir werden, wie in der Mathematik üblich, Funktionen vor der Verwendung erst definieren<sup>4</sup>.

```
1 public class AusgabeFunktion{
2     public static void gebeAus(int a){          // neue Funktion
3         System.out.println();
4         System.out.println("***** Wert der Variable ist "+a);
5         System.out.println("*****");
6         System.out.println();
7     }
8
9
10    // main-Funktion
11    public static void main(String [] args){
12        int a=4;
13        gebeAus(a);
14        a=(a*13)%12;
15        gebeAus(a);
16        a+=1000;
17        gebeAus(a);
18    }
19 }
```

Hier noch ein weiteres Beispiel:

Berechnen wir in einer Funktion für die Eingabe `x` den Wert  $f(x)=x \cdot 13 - 1024 \% (34+12)$ :

```
1 public class Funktion{
2     public static int funktion(int x){
3         int wert=(x*13)-1024%(34+12);
4         return wert;
5     }
}
```

<sup>4</sup> Obwohl die Funktionen auch in einer beliebigen Reihenfolge im Programm stehen können, fördert die mathematische Reihenfolge die Lesbarkeit doch erheblich.

```

6  public static void main(String [] args){
7      for (int i=0; i<10; i++)
8          System.out.println("x=" + i + " und f(x) = " + funktion(i));
9  }
10 }
11 }
```

In diesem Beispiel gibt die Funktion einen int-Wert mit der Anweisung `return` zurück und wird beendet. Sollten noch Zeilen nach einem `return` stehen, so gibt Java einen Fehler aus, denn diese Zeilen können nie ausgeführt werden.

Als Ausgabe erhalten wir:

```
C:\Java>java Funktion
x=0 und f(x)=-12
x=1 und f(x)=1
x=2 und f(x)=14
x=3 und f(x)=27
x=4 und f(x)=40
x=5 und f(x)=53
x=6 und f(x)=66
x=7 und f(x)=79
x=8 und f(x)=92
x=9 und f(x)=105
```

## 3.6 Zusammenfassung und Aufgaben

### Zusammenfassung

Mit dem einfachen Klassenkonzept ist es uns möglich, erste Programme in Java zu schreiben. Wir haben Werkzeuge in Java kennengelernt, mit denen wir die im vorhergehenden Kapitel vorgestellten Methoden der Programmentwicklung verwirklichen können. Neben der ausreichenden Kommentierung eines Programms haben wir gesehen wie es möglich ist, Programmabschnitte, die mehrfach verwendet werden, in Funktionen auszulagern.

### Aufgaben

Übung 1) Geben Sie ein Programm in Java an, das folgende Formeln berechnet:

$$(i) f_1(x) = x, (ii) f_2(x) = x^2 / 2 + 17 * 2, (iii) f_3(x) = \frac{(x-1)^3 - 14}{2}.$$

Übung 2) Schreiben Sie ein Programm, das für alle Zahlen  $i = 1 \dots 20$  folgende Funktion  $f(i) = i!$  berechnet und für jedes  $i$  eine Zeile ausgibt.

Übung 3) Berechnen Sie ein paar Summen:

$$(i) \sum_{i=0}^{28} (i-1)^2, (ii) \sum_{i=1}^{100} \frac{i*(i+1)}{2}, (iii) \sum_{i=1}^{25} \frac{(i+1)}{i}$$

Übung 4) In Abschnitt 2.4.1 wurden Typumwandlungen mittels Casten vorgestellt.

Überprüfen Sie, ob der größte darstellbare Wert für einen long in einen float passt (kleiner Hinweis: der größte darstellbare long ist Long.MAX\_VALUE), indem Sie zunächst den Inhalt des long in den float speichern, zurückcasten und beide, den Startwert und den neuen Wert, vergleichen.

Übung 5) Welche der folgenden Ausdrücke sind äquivalent (vergleichen Sie dabei die Resultate)?

(i)

- a) if (i==2) {j = n++;}
- b) if (i==(4/2)) {j = ++n;}
- c) if (i==((2\*3)+5-9)) {j = n+1;}

(ii)

- a) for (int i=0; i<10;i++) {System.out.println("Nummer: "+i+);}
- b) int i=0; while (i<10) {System.out.print("Nummer: "+i+" \n"); i++;}

Übung 6) Schreiben Sie eine einzige if-Anweisung, die zwei Integervariablen auf ihre Größe hin überprüft und anschließend den Wert der kleineren in die größere kopiert.

Übung 7) Schreiben Sie die Funktion istPrim, die als Eingabe einen Integerwert  $n$  erhält und mit einem boolean zurückgibt, ob  $n$  eine Primzahl ist oder nicht. Eine Primzahl  $n$  ist eine Zahl, die nur durch 1 und sich selber ganzzahlig ohne Rest teilbar ist. Die ersten Primzahlen sind: 2,3,5,7,11,13,...

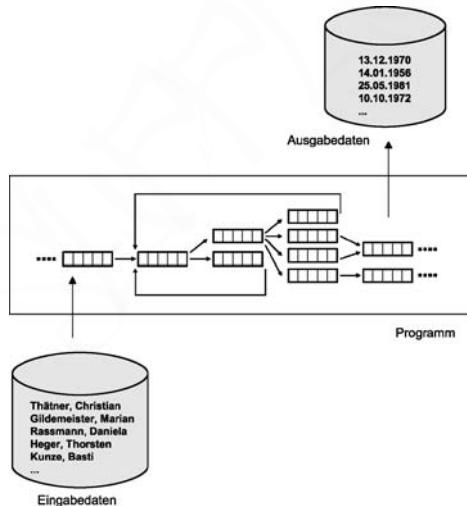
Übung 8) Verändern Sie die Funktion aus Übung 7 so, dass alle Primzahlen bis  $n$  auf dem Bildschirm ausgegeben werden.

Übung 9) Analysieren Sie den folgenden Programmabschnitt schrittweise und ver- gegenwärtigen Sie sich die Funktionsweise:

```
SpringZuI:
for (int i=0; i<=2; i++){
    SprungZuJ:
    for (int j=0; j<=2; j++){
        for (int k=0; k<=2; k++){
            if (k==1)
                break SprungZuJ;
        }
    }
}
```

## Tag 4: Daten laden und speichern

In vielen Fällen ist es erforderlich, in Dateien vorliegende Daten für die Ausführung eines Programms zu verwenden. Es könnte z. B. eine Datei vorliegen, in der die Namen, Matrikelnummern und erreichten Punkte in den Übungszetteln gespeichert sind und es stellt sich die Frage, wer von den Studenten nun die Zulassung für die Klausur erworben hat.



Ebenso ist es oft erforderlich, berechnete Daten nicht nur in der Konsole auszugeben, sondern diese in einer Datei abzulegen. Diese beiden Verfahren und die Möglichkeit, beim Start eines Java-Programms Parameter zu übergeben, werden wir in diesem Kapitel behandeln.

Ein Lernziel an dieser Stelle wird sein, die im ersten Augenblick unverständlich erscheinenden Programmteile, einfach mal zu verwenden. Wir müssen am Anfang

einen Mittelweg finden zwischen dem absoluten Verständnis für jede Programmzeile und der Verwendung einer gegebenen Teillösung. Spätestens bei dem Aufarbeitungsabschnitt in Kapitel 7 werden die hier verwendeten Konzepte klar verständlich sein.

## 4.1 Externe Programmeingaben

Die einfachste Methode, einem Programm ein paar Daten mit auf dem Weg zu geben, ist die Übergabe von Parametern auf der Kommandozeile. Wir haben ja schon gesehen, dass ein Javaprogramm nach der Kompilierung mit

```
C:\>javac MirIstWarm.java
C:\>java MirIstWarm
```

aufgerufen und gestartet wird. Wenn z. B. das Programm **MirIstWarm.java** wie weiter unten angegeben aussieht, wird einfach der Text „Mir ist heute zu warm, ich mache nix :.“ ausgegeben.

```
1 // MirIstWarm.java
2 public class MirIstWarm{
3     public static void main(String[] args){
4         System.out.println("Mir ist heute zu warm, ich mache nix :.");
5     }
6 }
```

In Zeile 3 sehen wir den Parameter `String[] args`. Er steht für solche Fälle zur Verfügung, in denen wir dem Programm einen oder mehrere Parameter vor dem Start mitgeben wollen. Ein Beispiel, an dem klar wird, wie es funktioniert:

```
1 // MeineEingaben.java
2 public class MeineEingaben{
3     public static void main(String[] args){
4         System.out.println("Eingabe 1: >" + args[0] + "< und");
5         System.out.println("Eingabe 2: >" + args[1] + "<");
6     }
7 }
```

Das Programm **MeineEingaben.java** gibt nun aus, welche Eingaben es erhalten hat:

```
C:\>javac MeineEingaben.java
C:\>java MeineEingaben Hallo 27
Eingabe 1: >Hallo< und
Eingabe 2: >27<
```

In diesem Fall war es wichtig zu wissen, wie viele Eingaben wir erhalten haben. Sollten wir auf einen Eintrag in der Stringliste args zugreifen, die keinen Wert erhalten hat, dann passiert folgendes:

```
C:\>java MeineEingaben Hallo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
at MeineEingaben.main(MeineEingaben.java:5)
```

Eine Fehlermeldung mit dem Hinweis „`ArrayIndexOutOfBoundsException`“ wird vom Programm geliefert. Wir haben also in *Zeile 5* auf einen Index zugegriffen, der nicht existiert und damit einen Fehler verursacht. Dieses Problem können wir mit folgenden zusätzlichen Zeilen umgehen:

```
1 // MeineEingaben.java
2 public class MeineEingaben{
3     public static void main(String[] args){
4         for (int i=0; i<args.length; i++)
5             System.out.println("Eingabe "+i+": >" + args[i] + "<");
6     }
7 }
```

Dieser Fehler lässt sich vermeiden, wenn wir zuerst die Anzahl der übergebenen Elemente überprüfen. In einer Schleife laufen wir anschließend durch die Elemente und geben sie nacheinander aus.

```
C:\>java MeineEingaben Hallo 27 und noch viel mehr!
Eingabe 0: >Hallo<
Eingabe 1: >27<
Eingabe 2: >und<
Eingabe 3: >noch<
Eingabe 4: >viel<
Eingabe 5: >mehr!<
```

Jetzt ist das Programm ohne Fehler in der Lage, beliebig viele Eingaben auszugeben, ohne es jedes Mal neu kompilieren zu müssen.

## 4.2 Daten aus einer Datei einlesen

Wenn wir es aber mit der Verwaltung einer Datenbank zu tun haben, wollen wir nicht über die Kommandozeile die vielen Daten übergeben. Wir können die Daten aus einer Datei lesen. Es gibt viele Möglichkeiten in Java, mit denen das zu bewerkstelligen ist.

Obwohl wir die Begriffe Objektorientierung, Klassen, try-catch usw. erst in Kapitel 7 behandeln werden, müssen wir hier auch schon externe Klassen und deren Funktionen verwenden. Das soll uns aber nicht weiter stören, wir sehen an einem Beispiel,

wie einfach es funktioniert. Hier geht es erstmal darum, Programmteile zu benutzen, wenn das Verwendungsschema bekannt ist. Später werden wir uns diese Programme noch einmal anschauen und sie besser verstehen.

```

1 import java.io.*;
2 public class LiesDateiEin{
3     public static void main(String[] args){
4         // Dateiname wird übergeben
5         String filenameIn = args[0];
6         try{
7             FileInputStream fis      = new FileInputStream(filenameIn);
8             InputStreamReader isr   = new InputStreamReader(fis);
9             BufferedReader bur     = new BufferedReader(isr);
10
11            // die erste Zeile wird eingelesen
12            String sLine = bur.readLine();
13
14            // lies alle Zeilen aus, bis keine mehr vorhanden sind
15            // und gib sie nacheinander aus
16            // falls von vornherein nichts in der Datei enthalten
17            // ist, wird dieser Programmabschnitt übersprungen
18            int zaehler = 0;
19            while (sLine != null) {
20                System.out.println("Zeile "+zaehler+": "+sLine);
21                sLine = bur.readLine();
22                zaehler++;
23            }
24            // schließe die Datei
25            bur.close();
26        } catch (IOException eIO) {
27            System.out.println("Folgender Fehler trat auf: "+eIO);
28        }
29    }
30 }
```

Dieses Programm bedarf zunächst keiner weiteren Erläuterung, da eine fürs Erste ausreichende Kommentierung vorhanden ist. Es gehört zum Erlernen einer Programmiersprache auch zu entscheiden, an welchen Stellen des Programms Kommentare notwendig sind. Oft verminderen zu viele unnötige Hinweise die Lesbarkeit eines Programms, aber das ist Geschmackssache.

Verwenden könnten wir **LiesDateiEin.java**, z. B. mit der Datei **namen.dat**. Den Inhalt einer Datei kann man sich auf der Konsole mit dem Befehl `type` anschauen:

```

C:\>type namen.dat
Harald Liebchen
Gustav Peterson
Gunnar Heinze
Paul Freundlich

C:\>java LiesDateiEin namen.dat
Zeile 0: Harald Liebchen
Zeile 1: Gustav Peterson
Zeile 2: Gunnar Heinze
Zeile 3: Paul Freundlich
```

Unser Programm kann eine Datei zeilenweise auslesen und gibt das eingelesene gleich auf der Konsole aus.

## 4.3 Daten in eine Datei schreiben

Um Daten in eine Datei zu speichern, schauen wir uns mal folgendes Programm an:

```

1 import java.io.*;
2 public class SchreibeInDatei{
3     public static void main(String [] args){
4         // Dateiname wird übergeben
5         String filenameOutput = args[0];
6         try{
7             BufferedWriter myWriter =
8                 new BufferedWriter(new FileWriter(filenameOutput, false));
9
10            // schreibe zeilenweise in die Datei filenameOutput
11            myWriter.write("Hans Mueller\n");
12            myWriter.write("Gundel Gaukel\n");
13            myWriter.write("Fred Feuermacher\n");
14
15            // schliesse die Datei
16            myWriter.close();
17        } catch (IOException eIO) {
18            System.out.println("Folgender Fehler trat auf: "+eIO);
19        }
20    }
21 }
```

Jetzt testen wir unser Programm und verwenden zur Überprüfung die vorher besprochene Klasse **LiesDateiEin.java**.

```
C:\>java SchreibeInDatei namen2.dat
C:\>java LiesDateiEin namen2.dat
Zeile 0: Hans Mueller
Zeile 1: Gundel Gaukel
Zeile 2: Fred Feuermacher
```

Wir stellen fest: Es hat funktioniert!

## 4.4 Daten von der Konsole einlesen

Wir sind nun in der Lage, Daten beim Programmstart mitzugeben und Dateien auszulesen, oft ist es aber wünschenswert eine Interaktion zwischen Benutzer und Programm zu haben. Beispielsweise soll der Benutzer eine Entscheidung treffen oder eine Eingabe machen. Das ist mit der Klasse **BufferedReader** schnell realisiert:

```

1 import java.io.*;
2 public class Einlesen{
3     public static void main(String [] args){
4         System.out.print("Eingabe: ");
5         try{
6             InputStreamReader isr = new InputStreamReader(System.in);
```

```

7     BufferedReader bur      = new BufferedReader(isr);
8
9     // Hier lesen wir einen String ein:
10    String str = bur.readLine();
11
12    // und geben ihn gleich wieder aus
13    System.out.println(str);
14 } catch(IOException e){}
15 }
16 }
```

Noch ein kleiner Test:

```
C:\>java Einlesen
Eingabe: Ich gebe etwas ein 4,5 a 1/2
Ich gebe etwas ein 4,5 a 1/2
```

## 4.5 Zusammenfassung und Aufgaben

### Zusammenfassung

Softwareprojekte sind kaum ohne ein gutes Datenverwaltungsmanagement zu realisieren. Seien es beispielsweise Benutzereinstellungen in Programmen oder Ergebnisse, die gespeichert werden müssen. Das Ein- bzw. Auslesen von Daten aus Dateien ist ein wichtiges Instrument für die Gestaltung von Programmen.

Neben den zwei vorgestellten Programmen zum Ein- und Auslesen von Dateien wurde auch das Konzept der externen Parameterübergabe (Eingabe über die Konsole), vor und während eines Programmablaufs, in Java besprochen.

### Aufgaben

Übung 1) Wir haben gesehen was passiert, wenn **weniger** Parameter übergeben werden als das Programm sie verlangt oder erwartet (siehe 4.1). Testen Sie, wie das Programm reagiert, wenn Sie **mehr** als die gewünschte Anzahl von Parametern übergeben.

Übung 2) Erzeugen Sie mit der Klasse **ErzeugeVieleWerte** eine Datei, die 100 Zeilen á 10 Spalten besitzt und in jedem Eintrag folgender Wert steht: Eintrag(Zeile  $i$ , Spalte  $j$ ) =  $i^2 + j$ .

Übung 3) Schreiben Sie ein Programm, das das Alphabet in Groß- und Kleinbuchstaben nacheinander in eine Datei „Alphabet.dat“ schreibt. Anschließend soll die Datei eingelesen werden und die Anzahl der enthaltenen Buchstaben gezählt und ausgegeben werden.

Übung 4) Modifizieren Sie Ihre Lösung zur Berechnung der **Primzahlen** mit folgenden Eigenschaften:

- die Funktion wird von der Klasse MeineFunktionen angeboten
- nach Verwendung wird eine schöne Ausgabe geboten
- Die Klasse soll ausreichend kommentiert werden und den bereits besprochenen Konventionen entsprechen

# 5

---

## Tag 5: Verwendung einfacher Datenstrukturen

Bisher haben primitive Datentypen für die Lösung einfacher Probleme ausgereicht. Jetzt wollen wir uns mit zusammengesetzten Datentypen auseinandersetzen, den sogenannten Datenstrukturen. Das Wort Datenstruktur verrät schon, dass es sich um Daten handelt, die in irgendeiner Form in Strukturen, die spezielle Eigenschaften besitzen, zusammengefasst werden. Diese Eigenschaften können sich z. B. darin auswirken, dass ein bestimmtes Datum<sup>1</sup> schneller gefunden oder eine große Datenmenge platzsparender gespeichert werden kann (eine gute Referenz zu diesem Thema ist [15]).

Als Einstiegspunkt sind Arrays, die in fast jeder Programmiersprache angeboten werden, gut geeignet. Mit Hilfe eines zweidimensionalen Arrays werden wir **Conway's Game of Life** implementieren, dazu müssen wir uns aber zunächst mit der Handhabung eines Arrays vertraut machen.

### 5.1 Arrays und Matrizen

#### 5.1.1 Erzeugung eines Arrays

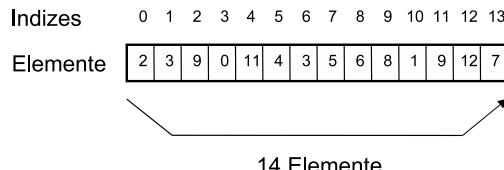
Beginnen wir mit der einfachsten Datenstruktur. Nehmen wir an, wir möchten nicht nur einen `int`, sondern viele davon verwalten. Dann könnten wir es, mit dem uns bereits bekannten Wissen, in etwa so bewerkstelligen:

```
int a, b, c, d, e, f;  
a=0;  
b=1;  
c=2;  
d=3;  
e=4;  
f=5;
```

---

<sup>1</sup> An dieser Stelle ist nicht das zeitliche Datum, sondern der Singular von Daten gemeint.

Das ist sehr aufwändig und unschön. Einfacher wäre es, wenn wir sagen könnten, dass wir  $k$  verschiedene int-Werte haben und diese dann über einen Index ansprechen. Genau das nennen wir eine **Liste** oder ein **Array**. Zu beachten ist, dass das erste Element eines Arrays mit dem Index 0 und das letzte der  $k$  Elemente mit dem Index  $k-1$  angesprochen werden.



Daran müssen wir uns gewöhnen und es ist eine beliebte Fehlerquelle. Es könnte sonst passieren, dass wir z. B. in einer Schleife alle Elemente durchlaufen möchten, auf das  $k$ -te Element zugreifen und einen Fehler verursachen.

```
 int[] a;
a = new int[10];
for (int i=0; i<=10; i++)
    System.out.println("a["+i+"]="+a[i]);
```

Wenn wir diesen Programmabschnitt aufrufen, erhalten wir folgende Ausgabe mit Fehler:

```
C:\Java>java Array
a[0]=0
a[1]=0
a[2]=0
a[3]=0
a[4]=0
a[5]=0
a[6]=0
a[7]=0
a[8]=0
a[9]=0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at Array.main(Array.java:5)
```

Wieder erhalten wir die schon bekannte `ArrayIndexOutOfBoundsException`. Sie tritt immer dann auf, wenn wir auf den Index eines Arrays zugreifen, den es gar nicht gibt. Glücklicherweise liefert Java an dieser Stelle einen Fehler. Das ist bei anderen Programmiersprachen, wie z. B. C++, nicht immer der Fall.

Die im letzten Kapitel besprochene Möglichkeit, Daten an ein Programm bei dessen Aufruf zu übergeben, hat auch Gebrauch von einem Array gemacht. Dem Array `args`, in diesem Fall ein Array von Strings<sup>2</sup>.

<sup>2</sup> Die Bezeichnung `args` trifft man zwar in der Literatur sehr häufig an, ist aber beliebig und kann einfach geändert werden.

Im letzten Beispiel haben wir bereits gesehen, wie ein int-Array erzeugt werden kann. Um ein k-elementiges Array zu erzeugen, schreiben wir:

```
<Datentyp>[] <name>;
<name> = new <Datentyp>[k];
```

oder in einer Zeile:

```
<Datentyp>[] <name> = new <Datentyp>[k];
```

Mit dem Befehl new wird Speicher für das Array bereitgestellt. Eine genauere Beschreibung werden wir später in Kapitel 7 vornehmen.

Das obere Beispiel hat gezeigt, wie auf die einzelnen Elemente über einen Index zugegriffen werden kann. Wir haben auch gesehen, dass die Einträge eines int-Arrays mit 0 initialisiert wurden. Die Initialisierung findet aber nicht immer statt, daher sollte man, falls nötig, anschließend in einer Schleife selber dafür Sorge tragen. In dem folgenden Beispiel wollen wir jetzt neue Daten in das Array schreiben:

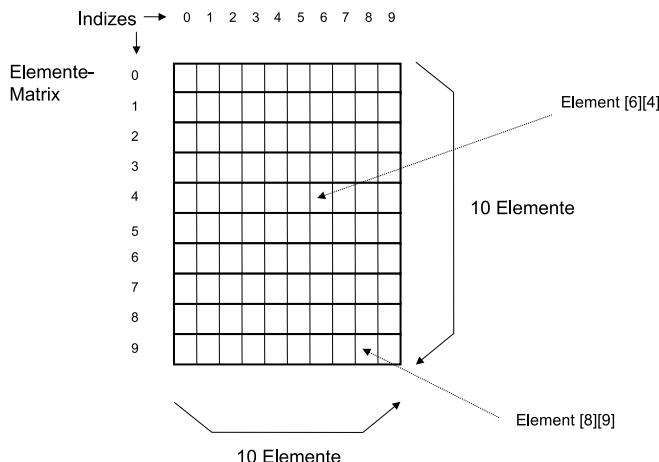
```
int[] a = new int[2];
a[0] = 3;
a[1] = 4;
```

Sollten wir schon bei der Erzeugung des Arrays wissen, welchen Inhalt die Elemente haben sollen, dann können wir das so vornehmen („Literale Erzeugung“ [8]):

```
int[] a = {1,2,3,4,5};
```

### 5.1.2 Matrizen oder multidimensionale Arrays

Für verschiedene Anwendungen ist es notwendig, Arrays mit mehreren Dimensionen anzulegen und zu verwenden. Ein Spezialfall mit der Dimension 2 ist die Matrix. Wir haben ein Feld von Elementen der Größe  $n \times m$ .



Wir erzeugen eine  $n \times m$  Matrix, indem wir zum Array eine Dimension dazunehmen:

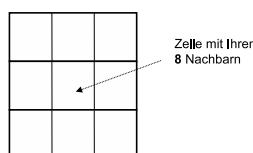
```
int [][] a = new int [n][m];
a[4][1] = 27;
```

Auf diese Weise können wir sogar noch mehr Dimensionen erzeugen:

```
int [][][][] a = new int [k][l][m][n];
```

### 5.1.3 Conway's Game of Life

Damit wir unsere neu gewonnenen Kenntnisse über Matrizen gleich einmal anwenden können, beschäftigen wir uns mit *Conway's Game of Life*<sup>3</sup> (z. B. bei Wikipedia [40] nachzulesen). Man stelle sich vor, die Welt bestünde nur aus einer 2-dimensionalen Matrix. Jeder Eintrag, wir nennen ihn jetzt mal Zelle oder Zellulärer Automat, kann zwei Zustände annehmen, er ist entweder lebendig oder tot. Jede Zelle interagiert mit ihren 8 Nachbarn.




---

<sup>3</sup> Mein Großvater brachte mir damals die Programmierung mit diesem Beispiel in BASIC bei. Es hat einen nachhaltigen Eindruck hinterlassen und motivierte mich weiter zu machen. Man könnte sagen, dass *Conway's Game of Life* mein Interesse für die Informatik und gerade für die Künstliche Intelligenz geweckt hat.

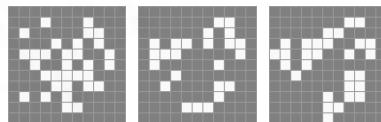
Diese Interaktion unterliegt den folgenden vier Regeln:

1. jede *lebendige* Zelle, die weniger als zwei lebendige Nachbarn hat, *stirbt* an Einsamkeit
2. jede *lebendige* Zelle mit mehr als drei lebendigen Nachbarn *stirbt* an Überbevölkerung
3. jede *lebendige* Zelle mit zwei oder drei Nachbarn fühlt sich wohl und *lebt* weiter
4. jede *tote* Zelle mit genau drei lebendigen Nachbarn wird wieder *zum Leben erweckt*.

Die Idee besteht nun darin, eine konkrete oder zufällige Konstellation von lebendigen und toten Zellen in dieser Matrix vorzugeben. Das bezeichnen wir dann als die erste Generation. Die zweite Generation wird durch die Anwendung der vier Regeln auf jede der Zellen erzeugt. Es wird geprüft, ob Zellen lebendig bleiben, sterben oder neu entstehen.

Lässt man ein solches System mit einer zufälligen Konstellation laufen, das heißt, wir schauen uns, wie in einem Film, die Generationen nacheinander an, dann erinnert uns das Zusammenspiel von Leben und Tod z. B. an Bakterienkulturen in einer Petrischale.

Es gibt inzwischen viele Variationen und verschiedenste Darstellungsmöglichkeiten. Wir werden eine traditionelle Darstellung für unser Matrizenbeispiel verwenden. Schauen wir uns die ersten drei Generationen, einer zufälligen Startkonfiguration der Zellen, in einer Welt mit den Maßen  $11 \times 11$  an:



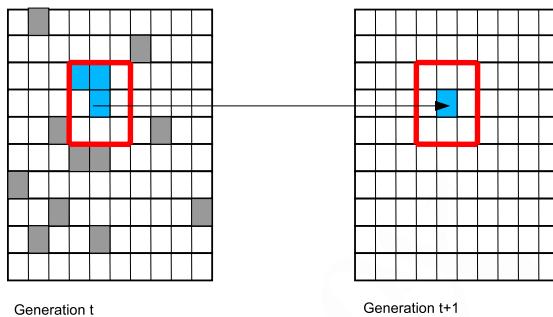
In den Jahren des Experimentierens sind viele interessante Konstellationen aufgetaucht, beispielsweise gibt es Muster, die scheinbar endlos neue Generationen erzeugen, also in der Lage sind, sich zu reproduzieren, oder andere, die zyklisch bestimmte Zustände annehmen.

### 5.1.3.1 Einfache Implementierung

Unsere Aufgabe wird es nun sein, eine sehr einfache Version zu implementieren. Wir benötigen dazu eine  $n \times m$  Matrix `welt` und setzen die Elemente `welt[i, j]` auf 0 für *tot* und 1 für *lebendig*. Damit hätten wir uns eine einfache Umgebung definiert. Jetzt müssen wir diese Matrix mit Werten füllen. Erst in einem späteren Kapitel werden wir besprechen, wie es möglich ist, Zufallszahlen in Java zu erzeugen, daher müssten

wir an dieser Stelle die Werte eigentlich „per Hand“ setzen. Aber, um uns die Arbeit zu ersparen, verwenden wir sie einfach schon.

Angenommen wir haben diese Matrix bereits und jedes Element steht entweder auf 0 oder 1. Um nun eine Folgegeneration zu berechnen, müssen wir eine zweite, leere Matrix `welt_neu` erzeugen. Anschließend gehen wir systematisch über alle Elemente der Matrix `welt` und prüfen die jeweiligen Nachbarn. Die Regeln liefern eine eindeutige Entscheidung, ob wir in der neuen Matrix an der gleichen Stelle den Wert 0 oder 1 setzen.



Wir könnten systematisch durch die Matrix `welt` laufen, indem wir zwei `for`-Schleifen verwenden.

Da wir über jede Generation informiert werden wollen, gibt es eine kleine Ausgabe-funktion.

```

1 import java.util.Random; // erläutern wir später
2 public class GameOfLife{
3     public static void gebeAus(boolean[][] m){
4         // Ein "X" symbolisiert eine lebendige Zelle
5         for (int i=0; i<10; i++){
6             for (int j=0; j<10; j++){
7                 if (m[i][j]) System.out.print("X ");
8                 else System.out.print("  ");
9             }
10            System.out.println();
11        }
12    }
}

```

Um die Regeln zu überprüfen, müssen wir die Anzahl der in der Nachbarschaft befindlichen lebenden Zellen ermitteln. Mit einem kleinen Trick können wir die Funktion sehr kurz halten<sup>4</sup>.

---

<sup>4</sup> An dieser Stelle muss ich darauf hinweisen, dass dieser Trick zwar machbar, aber unter Programmierern verpönt ist. Es gilt die Regel: „*Do not use exceptions for flow control!*“. Das bedeutet, dass unsere Schreibweise zwar recht kurz ist, aber das auf Kosten der Performance. Der Leser sei also ermuntert, eine eigene Methode zu schreiben, die die Nachbarschaftsregeln überprüft.

```

14 // Diese Methode lässt sich sicherlich schöner schreiben – wir
15 // nutzen hier die Tatsache aus, dass Java einen Fehler erzeugt,
16 // wenn wir auf ein Element außerhalb der Matrix zugreifen
17 public static int zahleUmgebung(boolean[][] m, int x, int y){
18     int ret = 0;
19     for (int i=(x-1);i<(x+2);++i){
20         for (int j=(y-1);j<(y+2);++j){
21             try{
22                 if (m[i][j])
23                     ret += 1;
24             }
25         catch (IndexOutOfBoundsException e){}
26     }
27     // einen zuviel mitgezählt?
28     if (m[x][y])
29         ret -= 1;
30
31     return ret;
32 }
```

In der main-Methode werden zwei Matrizen für zwei aufeinander folgende Generationen bereitgestellt. Exemplarisch werden die Zellkonstellationen einer Generation, gemäß den zuvor definierten Regeln, berechnet und ausgegeben.

```

29 public static void main(String[] args){
30     // unsere Welt soll aus 10x10 Elemente bestehen
31     boolean[][] welt      = new boolean[10][10];
32     boolean[][] welt_neu = new boolean[10][10];
33
34     // ****
35     // Erzeugt eine zufällige Konstellation von Einsen und Nullen
36     // in der Matrix welt. Die Chancen liegen bei 50%, dass eine
37     // Zelle lebendig ist.
38     Random generator = new Random();
39     double zufallswert;
40     for (int i=0; i<10; i++){
41         for (int j=0; j<10; j++){
42             zufallswert = generator.nextDouble();
43             if (zufallswert >= 0.5)
44                 welt[i][j] = true;
45         }
46     }
47     // ****
48
49     // Ausgabe der ersten Generation
50     System.out.println("Generation 1");
51     GameOfLife.gebeAus(welt);
52
53     int nachbarn;
54     for (int i=0; i<10; i++){
55         for (int j=0; j<10; j++){
56             // Zahle die Nachbarn
57             nachbarn = zahleUmgebung(welt, i, j);
58
59             if (welt[i][j]){
60                 // Regel 1, 2:
61                 if ((nachbarn<2) || (nachbarn>3))
62                     welt_neu[i][j] = false;
63
64                 // Regel 3:
65                 if ((nachbarn==2) || (nachbarn==3))
```

```

66         welt_neu[ i ][ j ] = true ;
67     }
68     else {
69         // Regel 4:
70         if (nachbarn==3)
71             welt_neu[ i ][ j ] = true ;
72     }
73 }
74 }
75 // Ausgabe der zweiten Generation
76 System.out.println("Generation 2");
77 GameOfLife.gebeAus(welt_neu);
78 }
79 }
```

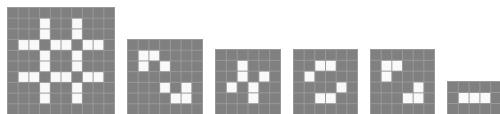
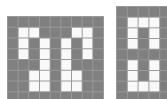
Unser Programm **GameOfLife.java** erzeugt zunächst zwei Matrizen der Größe  $10 \times 10$ . Dann wird die Matrix `welt` zufällig mit 1 und 0 gefüttert. Mit Zufallsgeneratoren beschäftigen wir uns aber erst später in Abschnitt 8.3. Die mit Zufallszahlen gefütterte `welt` lassen wir uns auf dem Bildschirm ausgeben, wobei ein X für eine lebendige und eine Lücke für eine tote Zelle steht. Im Anschluss daran gehen wir über alle Elemente der `welt` und wenden die vier Regeln zur Erzeugung einer neuen Generation `welt_neu` an.

```
C:\JavaCode>java GameOfLife
Generation 1
X X X X           X
      X   X X X
X X   X   X X
X X   X X   X
      X   X X
X X X   X X   X X
X   X X   X
X X X   X X
      X X   X
X X   X X X X X X
Generation 2
X X X           X
      X   X
X   X   X   X
X       X
      X
X       X
X   X   X X
      X
X X X   X X X X X
```

Schließlich geben wir die neu berechnete Generation wieder aus und können überprüfen, ob die Regeln eingehalten wurden.

### 5.1.3.2 Eine kleine Auswahl besonderer Muster

Für den interessierten Leser ist hier eine kleine Sammlung besonderer Zellkonstellationen zusammengestellt. Aber Vorsicht: *Conway's Game of Life* macht süchtig!

*Zyklische Muster**Gleiter**Interessante Startkonstellationen*

## 5.2 Zusammenfassung und Aufgaben

### Zusammenfassung

Die Verwendung von Datenstrukturen verbessert die Erstellung von Programmen enorm. Je nach Anwendung und gewünschter Funktionalität kann der Entwickler zwischen verschiedenen Datenstrukturen wählen. Listen, in unserem Fall Arrays, sind einfach zu verwenden und kommen in fast jedem Programm vor. Die Dimension der Datenstruktur Liste lässt sich erweitern. Für unser Projekt **Conway's Game of Life** haben wir Matrizen verwendet und daran die Handhabung geübt. In einem späteren Kapitel kommen wir auf dieses Projekt zurück und es wird Aufgabe des Lesers sein, eine dynamische Oberfläche zu entwickeln, die eine schickere Ausgabe erzeugt als die in unserem einfachen Beispiel.

### Aufgaben

Übung 1) Arbeiten Sie sich in die Themen **Stack** und **Warteschlange** in Java ein. Sie können beispielsweise folgende Literatur [15] studieren. Versuchen Sie nach dieser Anleitung einen Stack und eine Warteschlange mit einem Array zu realisieren (kleiner Hinweis: bei der Implementierung der Warteschlange sollten Sie ein zyklisches Array simulieren). Sie können dabei voraussetzen, dass die Einträge in die Datenstrukturen nur vom Datentyp int sind.

Übung 2) Schreiben Sie eine Methode, die als Eingaben zwei int-Arrays `a` und `b` erhält. Die Funktion liefert die elementweise Addition beider Vektoren, falls Sie die gleiche Dimension besitzen und ansonsten null ist.

Übung 3) Analog zu Übung 2 sollen Funktionen für Vektorsubtraktion und das Produkt aus Vektor und Skalar implementiert werden.

Übung 4) Schreiben Sie eine Funktion, die die Matrizenmultiplikation ausführt.

Übung 5) Experimentieren Sie mit dem Programm „**Conway's Game of Life**“. Erweitern Sie das Programm so, dass beliebig viele Generationen ausgeführt werden können. Finden Sie weitere Muster, die sich immer reproduzieren oder Besonderheiten aufweisen.

Übung 6) Schreiben Sie ein Programm, das neben der `main`-Funktion eine weitere enthält. Erzeugen Sie in der `main` zwei identisch gefüllte int-Listen der Länge 10. Übergeben Sie eine der beiden Listen per Parameterübergabe an die Funktion. Die Funktion soll die Reihenfolge der Werte innerhalb dieses Parameters umdrehen, aber keinen Rückgabewert liefern. Geben Sie die beiden Listeninhalte in der `main` nach Verwendung der Funktion aus.

Hat Sie das Ergebnis überrascht? Den Grund erfahren Sie später in Abschnitt 7.2.

Übung 7) Programmieren Sie eine **Räuber-Beute-Simulation** mit Füchsen und Schneeschuhhasen. Gegeben ist dabei eine Matrix, die eine Lebenswelt definiert. In dieser Welt leben Füchse und Schneeschuhhasen. Wenn ein Fuchs direkt auf einen Hasen trifft, wird dieser gefressen. Füchse, die über einen längeren Zeitraum keine Nahrung erhalten haben, sterben. Beide Arten besitzen eine gewisse Populationsdynamik.

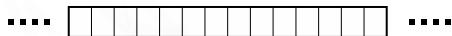
Beim Start soll eine zufällige Anzahl von Hasen und Füchsen in der Welt platziert werden. Treffen Hase und Fuchs auf demselben Feld zusammen, stirbt ein Hase. Jedes Exemplar beider Arten hat Eigenschaften, wie `alter`, `zeiteinheitOhneNahrung`, `position`. Hasen sterben nach 15 Zeiteinheiten und gebären pro Zeiteinheit jeweils fünf Kinder. Füchse bekommen alle zwei Zeiteinheiten zwei Kinder und sterben nach 20 Zeiteinheiten oder wenn sie vier Zeiteinheiten lang keine Nahrung gefunden haben.

## Tag 6: Debuggen und Fehlerbehandlungen

Zum Handwerkzeug eines Programmierers gehört die Fähigkeit, Fehler in seinen und anderen Programmen aufzuspüren und zu behandeln. Die beste Methode allerdings ist die Investition in ein gutes Konzept, um viele Fehler vor der Implementierung auszuschließen.

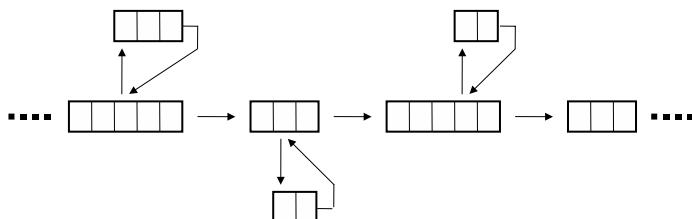
### 6.1 Das richtige Konzept

Es sollte bei der Entwicklung darauf geachtet werden, dass nicht allzu viele Programmzeilen in eine Funktion gehören.



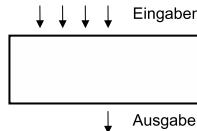
Das Problem sind oft wiederverwendete Variablen oder Seiteneffekte.

Besser ist es, das Programm zu gliedern und in Programmabschnitte zu unterteilen. Zum einen wird damit die Übersicht gefördert und zum anderen verspricht die Modularisierung den Vorteil, Fehler in kleineren Programmabschnitten besser aufzuspüren und vermeiden zu können.



Ein wichtiges Werkzeug der Modularisierung stellen sogenannte **Constraints** (Einschränkungen) dar. Beim Eintritt in einen Programmabschnitt (z. B. eine Funktion)

müssen die Eingabeparameter überprüft und damit klargestellt werden, dass der Programmteil mit den gewünschten Daten arbeiten kann. Das gleiche gilt für die Ausgabe. Der Entwickler der Funktion ist dafür verantwortlich, dass auch die richtigen Berechnungen durchgeführt und zurückgeliefert werden. Dies stellt sicher, dass ein innerhalb eines Moduls auftretender Fehler, auch dort gesucht und behandelt werden muss und nicht im aufrufenden Modul, welches eventuell fehlerhafte Daten liefert hat.



Kommentare sind hier unverzichtbar und fördern die eigene Vorstellung der Funktionsweise eines Programmabschnitts. Als Beispiel schauen wir uns mal die Fakultätsfunktion an:

```

1 public class Fakultaet{
2     /*
3         Fakultätsfunktion liefert für i=1 .. 13 die entsprechenden
4         Funktionswerte i! = i*(i-1)*(i-2)*....*1
5
6         Der Rückgabewert liegt im Bereich 1 .. 479001600
7
8         Sollte eine falsche Eingabe vorliegen, so liefert das Programm
9         als Ergebnis -1.
10    */
11    public static int fakultaet(int i){
12        // Ist der Wert ausserhalb des erlaubten Bereichs?
13        if ((i<=0)||(i>=13))
14            return -1;
15
16        // Rekursive Berechnung der Fakultaet
17        if (i==1)
18            return 1;
19        else
20            return i*fakultaet(i-1);
21    }
22
23    public static void main(String[] args){
24        for (int i=0; i<15; i++)
25            System.out.println("Fakultaet von "+i+" liefert "+fakultaet(i));
26    }
27 }
```

Damit haben wir erst einmal die Ein- und Ausgaben überprüft und sichergestellt, dass wir mit den gewünschten Daten arbeiten, nun müssen wir dafür Sorge tragen, die Module fehlerfrei zu halten.

Leider treten auch nach den besten Vorbereitungen Fehler in verschiedenen Formen auf. Dabei unterscheiden wir zwischen Fehlern, die das Programm zum Absturz bringen und fehlerhaften Berechnungen. Schauen wir uns zunächst an, wie wir Fehler in Java vermeiden können, die das Programm zum Absturz bringen.

## 6.2 Exceptions in Java

Wenn ein Fehler während der Ausführung eines Programms auftritt, wird ein Objekt einer Fehlerklasse (Exception) erzeugt. Da der Begriff Objekt erst später erläutert wird, stellen wir uns einfach vor, dass ein Programm gestartet wird, welches den Fehler analysiert und wenn der Fehler identifizierbar ist, können wir dieses Programm nach dem Fehler fragen und erhalten einen Hinweis, der Aufschluss über die Fehlerquelle gibt.

Schauen wir uns ein Beispiel an:

```

1 public class ExceptionTest{
2     public static void main( String [] args){
3         int d = Integer.parseInt(args[0]);
4         int k = 10/d;
5         System.out.println("Ergebnis ist "+k);
6     }
7 }
```

Auf den ersten Blick ist kein Fehler erkennbar, aber ein Test zeigt schon die Fehleranfälligkeit des Programms.

```

C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Exception in thread "main" java.lang.ArithmaticException: / by zero
        at ExceptionTest.main(ExceptionTest.java:5)

C:\>java ExceptionTest d
Exception in thread "main" java.lang.NumberFormatException: For input
string: "d"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at ExceptionTest.main(ExceptionTest.java:4)
```

Zwei Fehlerquellen sind hier erkennbar, die Eingabe eines falschen Typs und die Eingabe einer 0, die bei der Division einen Fehler verursacht. Beide sind für Java wohlbekannte Fehler, daher gibt es auch in beiden Fällen entsprechende Fehlerbezeichnungen `NumberFormatException` und `ArithmaticException`.

Um nun Fehler dieser Art zu vermeiden, müssen wir zunächst auf die Fehlersuche gehen und den Abschnitt identifizieren, der den Fehler verursacht. Wir müssen uns dazu die Frage stellen: In welchen Situationen (unter welchen Bedingungen) stürzt das Programm ab? Damit können wir den Fehler einengen und den Abschnitt besser lokalisieren. Dann müssen wir Abhängigkeiten überprüfen und die Module identifizieren, die für die Eingaben in diesen Abschnitt zuständig sind.

Angenommen, wir haben einen Bereich lokalisiert und wollen diesen nun beobachten, dazu verwenden wir die `try-catch`-Klausel.

### 6.2.1 Einfache try-catch-Behandlung

Die Syntax dieser Klausel sieht wie folgt aus:

```
try {
    <Anweisung>;
    ...
    <Anweisung>;
} catch(Exception e) {
    <Anweisung>;
}
```

Die try-catch Behandlung lässt sich als „*Versuche dies, wenn ein Fehler dabei auftritt, mache das.*“ lesen. Um unser Programm aus Abschnitt 6.2 vor einem Absturz zu bewahren, wenden wir diese Klausel an und testen das Programm.

```
1 public class ExceptionTest2{
2     public static void main(String[] args){
3         try{
4             int d = Integer.parseInt(args[0]);
5             int k = 10/d;
6             System.out.println("Ergebnis ist "+k);
7         } catch(Exception e){
8             System.out.println("Fehler ist aufgetreten...");
9         }
10    }
11 }
```

Wir testen nun die gleichen Eingaben.

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Fehler ist aufgetreten ...

C:\>java ExceptionTest d
Fehler ist aufgetreten ...
```

Einen Teilerfolg haben wir nun schon zu verbuchen, da das Programm nicht mehr abstürzt. Der Bereich in den geschweiften Klammern nach dem Schlüsselwort `try` wird gesondert beobachtet. Sollte ein Fehler auftreten, so wird die weitere Abarbeitung innerhalb dieser Klammern abgebrochen und der Block nach dem Schlüsselwort `catch` ausgeführt.

## 6.2.2 Mehrfache try-catch-Behandlung

Dummerweise können wir nun die Fehler nicht mehr eindeutig identifizieren, da sie die gleiche Fehlermeldung produzieren. Um die Fehler aber nun eindeutig abzufangen, lassen sich einfach mehrere catch-Blöcke mit verschiedenen Fehlertypen angeben:

```

try {
    <Anweisung>;
    ...
    <Anweisung>;
} catch (Exceptiontyp1 e1) {
    <Anweisung>;
} catch (Exceptiontyp2 e2) {
    <Anweisung>;
} catch (Exceptiontyp3 e3) {
    <Anweisung>;
}

```

Wenden wir die neue Erkenntnis auf unser Programm an:

```

1 public class ExceptionTest3{
2     public static void main(String[] args){
3         try{
4             int d = Integer.parseInt(args[0]);
5             int k = 10/d;
6             System.out.println("Ergebnis ist "+k);
7         } catch(NumberFormatException nfe){
8             System.out.println("Falscher Typ! Gib eine Zahl ein ...");
9         } catch(ArithmetricException ae){
10            System.out.println("Division durch 0! ...");
11        } catch(Exception e){
12            System.out.println("Unbekannter Fehler aufgetreten ...");
13        }
14    }
15 }

```

Bei den schon bekannten Eingaben liefert das Programm nun folgende Ausgaben:

```

C:\>java ExceptionTest3 2
Ergebnis ist 5

C:\>java ExceptionTest3 0
Division durch 0! ...

C:\>java ExceptionTest3 d
Falscher Typ! Gib eine Zahl ein ...

```

Hier sei nur am Rande erwähnt, dass die try-catch-Klausel noch weitere Eigenschaften besitzt und Fehlerklassen selber programmiert werden können. Da wir aber

noch kein Verständnis für Objektorientierung entwickelt haben und das erst im Folgekapitel besprochen wird, lege ich dem interessierten Leser hier ans Herz, sich zunächst mit dem erweiterten Klassenkonzept auseinander zu setzen und später das erweiterte Fehlerkonzept nachzuarbeiten (z. B. hier [1, 9]).

## 6.3 Fehlerhafte Berechnungen aufspüren

Im vorhergehenden Abschnitt haben wir uns mit den Fehlern auseinander gesetzt, die das Programm unweigerlich zum Absturz bringen und eine Technik kennengelernt, diese Abstürze aufzuspüren und zu vermeiden.

Beim Programmieren wird leider ein nicht unwesentlicher Teil der Zeit mit dem Aufsuchen von Fehlern verbracht. Das ist selbst bei sehr erfahrenen Programmierern so und gerade, wenn die Projekte größer und unübersichtlicher werden, sind effiziente Programmietechniken, die Fehler vermeiden, unabdingbar. Sollte sich aber doch ein Fehler eingeschlichen haben, gibt es einige Vorgehensweisen, die die zum Auffinden benötigte Zeit auf ein Mindestmaß reduzieren.

Wir wollen hier eine einfache, aber effektive Denkweise zum Aufspüren von Fehlern vorstellen. Fangen wir mit einem Beispiel an.

### 6.3.1 Berechnung der Zahl pi

Folgendes Programm soll eine Näherung für die Kreiszahl  $\pi$  berechnen. Gottfried Wilhelm Leibniz gab 1682 für die Näherung von  $\frac{\pi}{4}$  eine Berechnungsvorschrift an, die als **Leibniz-Reihe** bekannt ist. Am Ende der Berechnung müssen wir das Ergebnis also noch mit 4 multiplizieren, um eine Näherung für  $\pi$  zu erhalten.

Die Vorschrift besagt:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Der Nenner wird also immer um 2 erhöht, während das Vorzeichen jeden Schritt wechselt. Eine Schleife bietet sich zur Berechnung an.

```

1 public class BerechnePI{
2     static int max_iterationen = 100;
3     static public double pi(){
4         double PI = 0;
5         int vorzeichen = 1;
6         for (int i=1; i<=max_iterationen*2; i+=2){
7             PI += vorzeichen*(1/i);
8             vorzeichen *= -1;
9         }
10        return 4*PI;
11    }
12 }
```

```

13  public static void main(String [] args){
14      double PI = pi();
15      System.out.println("Eine Näherung fuer PI ist "+PI);
16  }
17 }
```

$\pi$  ist ungefähr 3.141592..., wir erwarten nach 100 Iterationen also ein ähnliches Ergebnis. Stattdessen erhalten wir:

```
C:\JavaCode>java BerechnePI
Eine Näherung fuer PI ist 4.0
```

Das ist keine wirklich gute Näherung. Erhöhen wir die Iterationsanzahl auf 1000, bleibt das Ergebnis sogar unverändert.

Um den Fehler aufzuspüren, versuchen wir die Berechnungen schrittweise nachzuvollziehen. Überprüfen wir zunächst, ob das Vorzeichen und der Nenner für die Berechnung stimmen. Dazu fügen wir in die Schleife folgende Ausgabe ein:

```

...
for (int i=1; i<max_iterationen; i+=2){
    System.out.println("i:"+i+" vorzeichen:"+vorzeichen);
    PI += vorzeichen*(1/i);
    vorzeichen *= -1;
}
...
```

Als Ausgabe erhalten wir:

```
C:\JavaCode>java BerechnePI
i:1 vorzeichen:1
i:3 vorzeichen:-1
i:5 vorzeichen:1
i:7 vorzeichen:-1
...
```

Das Vorzeichen alterniert, ändert sich also in jeder Iteration. Das ist korrekt. Die Ausgabe erweitern wir, um zu sehen, wie sich PI im Laufe der Berechnungen verändert.

```
System.out.println("i:"+i+" vorzeichen:"+vorzeichen+" PI:"+PI);
```

Wir erhalten:

```
C:\JavaCode>java BerechnePI
i:1 vorzeichen:1 PI:0.0
i:3 vorzeichen:-1 PI:1.0
i:5 vorzeichen:1 PI:1.0
i:7 vorzeichen:-1 PI:1.0
...
```

Vor dem ersten Schritt hat `PI` den Initialwert 0. Nach dem ersten Schritt ist `PI=1`. Soweit so gut. Allerdings ändert sich `PI` in den weiteren Iterationen nicht mehr. Hier tritt der Fehler zum ersten Mal auf.

Werfen wir einen Blick auf die Zwischenergebnisse.

```
double zwischenergebnis = vorzeichen * (1/i);
System.out.println("i:" + i + " Zwischenergebnis:" + zwischenergebnis);
```

Jetzt sehen wir, an welcher Stelle etwas schief gelaufen ist:

```
C:\JavaCode>java BerechnePI
i:1 Zwischenergebnis:1.0
i:3 Zwischenergebnis:0.0
i:5 Zwischenergebnis:0.0
i:7 Zwischenergebnis:0.0
...
```

Der Fehler ist zum Greifen nah. Die Berechnung `vorzeichen*(1/i)` liefert in jedem Schritt, außer dem ersten, den Wert 0.0 zurück. Im Abschnitt 2.4.2 wurde auf dieses Problem hingewiesen.

Die Berechnung `(1/i)` liefert immer das Ergebnis 0, da sowohl 1 als auch `i` vom Typ `int` sind. Der Compiler verwendet die ganzzahlige Division, was bedeutet, dass alle Stellen nach dem Komma abgerundet werden. Dieses Problem lässt sich leicht lösen, indem wir die 1 in `1.0` ändern und damit aus dem implizit angenommenen `int` ein `double` machen. Eine andere Möglichkeit wäre das Casten von `i` zu einem `double`. In beiden Fällen liefert das korrigierte Programm eine gute Näherung.

```
C:\JavaCode>java BerechnePI
Eine Näherung für PI ist 3.1315929035585537
```

Ein etwas geübter Programmierer erahnt einen solchen Fehler bereits beim Lesen des Codes. Das strategisch günstige Ausgeben von Zwischenergebnissen und Variablenwerten ist jedoch auch ein probates Mittel zum Auffinden von schwierigeren Fehlern. Wird das Programm jedoch größer, ist ein Code von anderen Programmierern beteiligt und sind die Berechnungen unübersichtlich, dann helfen die von den meisten Programmierumgebungen bereitgestellten Debugger<sup>1</sup> enorm.

---

<sup>1</sup> Ein Debugger ermöglicht, durch die Kontrolle des Programmablaufs während der Ausführung, Fehler gezielter zu finden. Die meisten Systeme erlauben Breakpoints (Haltepunkte), bei denen alle aktuell im Speicher befindlichen Variablen und Zustände ausgelesen werden können.

### 6.3.2 Zeilenweises Debuggen und Breakpoints

Wir haben gesehen, wie wichtig es sein kann, den Inhalt von Variablen während des Programmablaufs zu überwachen. Dies lässt sich mit geschickt platzierten Konsoleausgaben lösen, wir wollen es als **zeilenweises Debuggen** bezeichnen.

Ist der Fehler allerdings noch nicht ausreichend lokalisiert, hat man entweder die Wahl, extrem viele Ausgaben einzubauen oder das Programm immer nur um eine Ausgabe zu ergänzen und es dann neu zu compilieren. Die erste Möglichkeit hat den großen Nachteil, dass die Übersicht der Ausgaben und des Programms meistens verschlechtert wird, was die Suche nach einem Fehler nicht gerade erleichtert. Bei der zweiten Variante kann der Zeitaufwand sehr groß sein, besonders wenn das Programm groß ist und das Compilieren bereits Minuten verbraucht. Ein Debugger bietet dem Programmierer die Möglichkeit, das Programm jederzeit zu unterbrechen und sich den Wert jeder Variablen anzusehen. Anschließend kann man das Programm weiterlaufen lassen oder gezielt Zeile für Zeile abarbeiten.

Das Unterbrechen eines Programms geschieht mit Hilfe sogenannter **Breakpoints** (Haltpunkte). In den meisten Entwicklungsumgebungen (z.B. Eclipse, NetBeans) klickt man, um ein Breakpoint zu setzen, links neben die entsprechende Programmzeile. Dort erscheint dann ein meist rot markierter Punkt. Das Programm kann jetzt gestartet werden (darauf achten, dass man es mit Debugging startet und nicht einfach mittels Run/Start). Sobald die Programmausführung an einer Zeile, die mit einem Breakpoint versehen ist, ankommt, wird das Programm angehalten.

Oft ist es hilfreich, nicht jedes Mal, wenn eine bestimmte Zeile ausgeführt wird, das Programm anzuhalten. Es könnte zum Beispiel sein, dass wir bereits wissen, dass der Fehler erst nach 100 Schleifendurchläufen auftritt. Wir wollen den Breakpoint also in einer Schleife platzieren, aber nicht erst 100 mal das Programm fortsetzen müssen, bevor es interessant wird. Dazu gibt es **konditionale Breakpoints**. Man verknüpft den Breakpoint also mit einer Bedingung, z. B.  $k > 100$ .

## 6.4 Zusammenfassung und Aufgaben

### Zusammenfassung

Um Funktionen übersichtlich zu halten und Fehler zu vermeiden oder besser finden zu können, müssen Programme modularisiert werden. Das richtige Konzept ist dabei entscheidend.

Programme können Fehler verursachen, die sie zum Absturz bringen, oder Berechnungen nicht in der Weise durchführen, wie es eigentlich gewünscht ist. Für den ersten Fall haben wir die Exception-Klasse kennengelernt und wissen, wie sie verwendet wird. Mit dem zeilenweisen Debuggen oder mit der Hilfe von Breakpoints können wir fehlerhafte Berechnungen aufspüren.

## Aufgaben

Übung 1) In Aufgabensammlung 2 Aufgabe 3 (ii) sollte die Summe berechnet werden. Leider liefert der folgende Programmcode nicht die gewünschten Ergebnisse. Finden Sie die Fehler:

```
public static double sum1(){
    int i, startwert=1;
    double d, h;
    for (i==startwert; i>100; i++)
        System.out.println(d);
    {h=(i*i*i)/2;
    d=d+h;
    }
    return d;
}
```

Übung 2) Das **Wallis-Produkt**, 1655 von John Wallis formuliert, gibt eine Näherung für  $\frac{\pi}{2}$  mit der Vorschrift  $\frac{\pi}{2} = \frac{2}{1} * \frac{2}{3} * \frac{4}{3} * \frac{4}{5} * \frac{6}{5} * \frac{6}{7} * \dots$ . Implementieren Sie dieses Verfahren und geben Sie eine Näherung für  $\pi$  an.

## Tag 7: Erweitertes Klassenkonzept

Bisher haben wir ein sehr einfaches Klassenkonzept verwendet, um die ersten Programmierübungen beginnen zu können. Jetzt ist es an der Zeit, den Begriff **Objektorientierung** näher zu beleuchten und zu verstehen. Wir werden mit dem Spielprojekt *Fußballmanager* verschiedene Konzepte erörtern. Parallel zu den Grundideen werden wir eine Motivation für die Verwendung der **Vererbung** entwickeln.

In den vorhergehenden Kapiteln sind einige Fragen aufgekommen, die mit dem Konzept der Objektorientierung zu erklären sind. Wir werden jetzt in der Lage sein, diese Fragen zu beantworten und blättern dabei die ersten Kapitel noch einmal gemeinsam durch. Um für den erfolgreichen Start eines Projekts ausgerüstet zu sein, werden wir Klassendiagramme und ihre Funktionen besprechen.

### 7.1 Entwicklung eines einfachen Fußballmanagers

Es gibt Standardbeispiele für die Einführung in die Objektorientierung, da wären **Autos** und **Autoteile** oder **Professoren** und **Studenten** zu nennen. Dieses Buch möchte davon abweichen und gleich mit einem ersten richtigen Softwareprojekt starten. Wir werden Schritt für Schritt einen kleinen Fußballmanager entwickeln und am Ende des Kapitels sogar zwei Mannschaften in einem Freundschaftsspiel gegeneinander antreten lassen. An vielen Stellen wurde aufgrund der Übersicht immer die einfachste Variante verwendet. Es sei dem interessierten Leser überlassen, diese Fußballmanager-Version auszubauen und mit weiteren interessanten Funktionen auszustatten.

#### 7.1.1 Spieler und Trainer

Beginnen wir mit den wichtigen Akteuren einer Mannschaft, da wären der **Trainer** und die elf **Spieler**. Spieler und Trainer besitzen Eigenschaften. Bevor wir mit

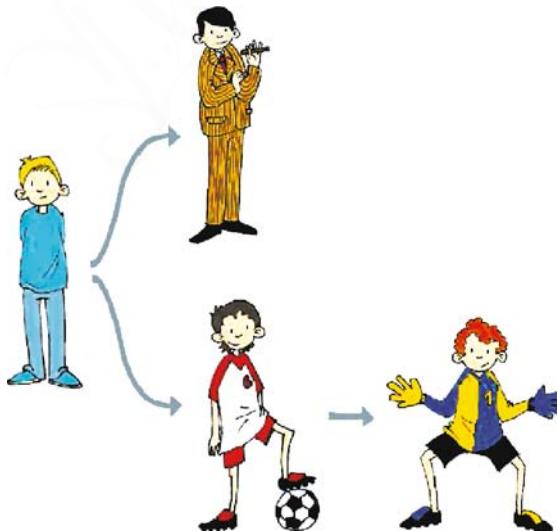
dem Entwurf der Klassen beginnen, müssen wir zunächst das Konzept der **Klassifizierung** behandeln. Dabei spielen die Begriffe Generalisierung und Spezialisierung eine entscheidende Rolle.

### 7.1.1.1 Generalisierung und Spezialisierung

Unter den beiden Begriffen **Generalisierung** und **Spezialisierung** verstehen wir zwei verschiedene Vorgehensweisen, Kategorien und Stammbäume von Dingen zu beschreiben. Wenn wir bei Dingen Gemeinsamkeiten beschreiben und danach kategorisieren, dann bezeichnen wir das als **Generalisierung**.

Beispielsweise wurden und werden die evolutionären Stammbäume der Pflanzen und Tiere so aufgestellt, dass bei einer großen Übereinstimmung morphologischer Eigenschaften mit größerer Wahrscheinlichkeit ein gemeinsamer Vorfahre existiert haben muss. Dieser Vorfahre erhält einen Namen und besitzt meistens die Eigenschaften, die die direkten Nachfahren gemeinsam haben. Er ist eine Generalisierung seiner Nachfahren. Wir erhalten auf diese Weise einen Stammbaum von den Blättern zu der Wurzel.

Mit der **Spezialisierung** beschreiben wir den umgekehrten Weg. Aus einem „Ur-Ding“ können wir durch zahlreiche Veränderungen der Eigenschaften neue Dinge kreieren, die Eigenschaften übernehmen oder neue entwickeln. Dazu lesen wir die folgende Abbildung von links nach rechts. Eine Person kann sich zu einem Trainer oder Spieler spezialisieren. Mit zusätzlichen Eigenschaften ausgestattet, lässt sich ein Spieler, der immer noch eine Person ist, in einen Torwart erweitern.



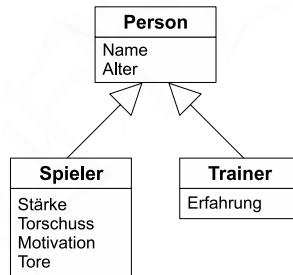
Wir haben elf Spieler, z. B. mit den Eigenschaften *Name*, *Alter*, *Stärke*, *Torschuss*, *Motivation* und *Tore*. Neben den Spielern haben wir einen Trainer mit den Eigenschaften *Name*, *Alter* und *Erfahrung*. Wir sehen schon, dass es Gemeinsamkeiten gibt:

Spieler	Trainer
Name Alter Stärke Torschuss Motivation Tore	Name Alter Erfahrung

Es wäre natürlich vollkommen legitim, Spieler und Trainer auf diese Weise zu modellieren. Aber wir wollen das Konzept der **Vererbung** kennenlernen und anwenden, daher versuchen wir Gemeinsamkeiten zu finden und diese besser darzustellen.

### 7.1.1.2 Klassen und Vererbung

Die Gemeinsamkeiten können wir zu einer eigenen Kategorie zusammenfassen und die Eigenschaften an Spieler und Trainer vererben. Diese neue Kategorie nennen wir z. B. Person. Spieler, Trainer und Person sind Kategorien oder **Klassen**.



Die Pfeile zeigen in die Richtung des Vorfahren (siehe 15.2 oder UML [42]). Wir haben also eine Repräsentation gefunden, die die Daten nicht mehr unnötig doppelt darstellt, wie es bei *name* und *alter* gewesen wäre, sondern sie übersichtlich nur einmal in der Klasse **Person** abgelegt. Des Weiteren wurde die Spezialisierung der **Person** zu den Klassen **Spieler** und **Trainer** durch zusätzliche Eigenschaften beschrieben. Um diese Darstellung in einem Java-Programm zu implementieren, müssen wir drei Klassen **Person**, **Spieler** und **Trainer** anlegen.

Beginnen wir mit der Klasse **Person**:

```

1 public class Person{
2     // Eigenschaften einer Person:
3     public String name;
4     public int alter;
5 }
```

Jetzt wollen wir die Klasse **Spieler** von **Person** ableiten und alle Eigenschaften, die die Klasse **Person** anbietet, übernehmen. In Java erweitern wir die Definition einer Klasse mit dem Befehl extends und den Namen der Klasse, von der wir ableiten wollen.

```
public class A extends B{
}
```

Jetzt können wir **Spieler** von **Person** ableiten:

```
1 public class Spieler extends Person{
2     // Zusätzliche Eigenschaften eines Spielers:
3     public int staerke;      // von 1 (schlecht) bis 10 (super)
4     public int torschuss;    // von 1 (schlecht) bis 10 (super)
5     public int motivation;  // von 1 (schlecht) bis 10 (super)
6     public int tore;
7 }
```

Das war alles. Jetzt haben wir zwei Klassen **Spieler** und **Person**. Alle Eigenschaften oder Attribute der Klasse **Person** sind nun auch in **Spieler** enthalten, darüber hinaus hat ein Spieler noch die Attribute staerke, torschuss, motivation und tore. Mit diesen beiden Klassen haben wir eine einfache Vererbung realisiert.

### 7.1.1.3 Modifizierer public und private

Nehmen wir an, dass zwei verschiedene Programmierer diese Klassen geschrieben haben und dass der Programmierer der Klasse **Person** für die Variable **alter** nur positive Zahlen zwischen 1 und 100 akzeptieren möchte. Er hat aber keinen Einfluss auf die Verwendung, da die Variable **alter** mit dem zusätzlichen Attribut **public** versehen wurde. Das bedeutet, dass jeder, der diese Klasse verwenden möchte, auf diese Attribute uneingeschränkt zugreifen kann.

Eine Klasse kann Eigenschaften besitzen, wie **name**, **alter**, ... das sind Daten oder Fakten, aber auch Methoden oder Funktionen. Es gibt die Möglichkeit diese Variablen vor Zugriff zu schützen, indem das Attribut **private** verwendet wird. Jetzt kann diese Variable nicht mehr außerhalb der Klasse angesprochen werden. Um aber die Variablen verändern und lesen zu können, schreiben wir zwei Funktionen und geben ihnen das Attribut **public**. Diese beiden Funktionen sind sogenannte **get-set-Funktionen**. Es gibt meistens eine get- und eine set-Funktion pro Klassenattribut. Für die Klasse **Person** könnte es so aussehen:

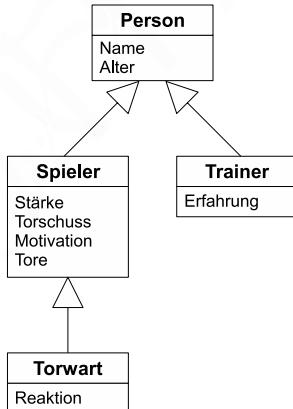
```
1 public class Person{
2     // Eigenschaften einer Person:
3     private String name;
4     private int alter;
5
6     // Funktionen (get und set):
```

```

7  public String getName(){
8      return name;
9  }
10
11 public void setName( String n){
12     name = n;
13 }
14
15 public int getAlter(){
16     return alter;
17 }
18
19 public void setAlter(int a){
20     alter = a;
21 }
22 }
```

Um die Variable `name` zu verändern, kann der externe Aufruf `setName("neuer Name")` verwendet werden. Später werden wir die Zugriffe auf diese Klassen noch einmal genauer besprechen, jetzt geht es uns erstmal um eine einfache Darstellungsform.

Bei den Spielern gibt es noch einen Sonderling, den Torwart. Als zusätzliche Eigenschaft hat er `reaktion`, damit wird später entschieden, ob er die Torschüsse hält oder nicht. Sicherlich könnten wir an dieser Stelle die Spieler noch in Abwehr, Mittelfeld und Angriff unterscheiden, aber fürs erste soll die Spezialisierung der Klasse **Spieler** zum **Torwart** als Spezialfall eines Spielers genügen:



#### 7.1.1.4 Objekte und Instanzen

Bisher haben wir die Dinge klassifiziert und die Gemeinsamkeiten in zusätzlichen Klassen beschrieben, aber noch keinen Spieler oder Trainer, der diese Eigenschaften und Funktionen besitzt erzeugt und untersucht. Wenn von einer Klasse ein Exemplar

erzeugt wird (die Klasse stellt sozusagen den Bauplan fest), dann nennt man das ein **Objekt** oder eine **Instanz** dieser Klasse.

Hier die aktuelle Klasse **Trainer** mit den get-set-Funktionen:

```

1 public class Trainer extends Person{
2     // Zusätzliche Eigenschaften eines Trainers:
3     private int erfahrung;
4
5     // Funktionen (get und set):
6     public int getErfahrung(){
7         return erfahrung;
8     }
9
10    public void setErfahrung(int e){
11        erfahrung = e;
12    }
13 }
```

Wenn wir beispielsweise ein Programm schreiben wollen, das mit Jürgen Klinsmann als Trainer arbeiten möchte, dann erzeugen wir eine neue Instanz der Klasse **Trainer** und geben ihm die Informationen name="Jürgen Klinsmann", alter=42 und erfahrung=7.

```

1 public class Test{
2     public static void main(String[] args){
3         Trainer trainer = new Trainer();
4         trainer.setName("Jürgen Klinsmann");
5         trainer.setAlter(42);
6         trainer.setErfahrung(7);
7     }
8 }
```

Wir erzeugen eine Instanz der Klasse **Trainer**. Ähnlich wie bei den primitiven Datentypen müssen wir Speicherplatz reservieren und machen das hier in der dritten Zeile.

```
Trainer trainer = new Trainer();
```

Jetzt können wir mit dem Objekt `trainer` arbeiten. Sollten die Daten des Objekts geändert werden, so können wir das über die mit `public` versehenen Funktionen der Klasse **Trainer** und **Person** tun. Wir sehen schon, dass die Funktion `setErfahrung` in der Klasse **Trainer** definiert wurde und verwendet werden kann. Die Funktion `setName` wurde aber in **Person** definiert und kann trotzdem verwendet werden. Das verdanken wir der Vererbung.

### 7.1.1.5 Konstruktoren in Java

Im vorhergehenden Beispiel war es etwas umständlich, erst ein Objekt zu erzeugen und anschließend die Parameter über die set-Methoden zu setzen. Es geht bedeutend

einfacher mit den **Konstruktoren**. Sie sind quasi die Funktionen, die bei der Reservierung des Speicherplatzes, bei der Erzeugung eines Objekts, ausgeführt werden. Beim **Trainer**-Beispiel könnten wir folgende Zeile schreiben, um die ganze Sache zu verkürzen, meinen aber das Gleiche:

```

1 public class Test{
2     public static void main(String [] args){
3         Trainer trainer = new Trainer("Jürgen Klinsmann", 42, 7);
4     }
5 }
```

Wir könnten festlegen, dass an der ersten Stelle der name, dann alter und erfahrung stehen. Das machen wir nun mit folgender Änderung in der Klasse **Trainer**:

```

1 public class Trainer extends Person{
2     // Zusätzliche Eigenschaften eines Trainers:
3     private int erfahrung;
4
5     // Konstruktoren
6     public Trainer(String n, int a, int e){
7         super(n, a);
8         erfahrung = e;
9     }
10
11    // Funktionen (get und set):
12    ...
13 }
```

Der **Konstruktor** entspricht der Syntax:

```
public <Klassenname>(Parameterliste) {  
}
```

Die Anweisung super mit den Parametern name und alter ruft den **Konstruktor** der Klasse auf, von der geerbt wird. In diesem Beispiel also den Konstruktor der Klasse **Person**. Da aber name und alter in der Klasse **Person** gespeichert sind und wir dort ebenfalls mit einem **Konstruktor** eine einfachere Initialisierung eines Objekts haben möchten, ändern wir die Klasse **Person**, wie folgt:

```

1 public class Person{
2     // Eigenschaften einer Person:
3     private String name;
4     private int alter;
5
6     // Konstruktoren
7     public Person(String n, int a){
8         name = n;
9         alter = a;
10    }
11
12    // Funktionen (get und set):
13    ...
14 }
```

Die Begriffe Generalisierung, Spezialisierung, Klasse, Objekt, Instanz, Konstruktor, Vererbung, `super`, `public` und `private` sollten jetzt keine größeren Probleme mehr darstellen. Der Vollständigkeit halber wollen wir uns noch die Klassen **Spieler** und **Torwart** anschauen:

```

1 import java.util.Random;
2
3 public class Spieler extends Person{
4     // Zusätzliche Eigenschaften eines Spielers:
5     private int staerke;      // von 1 (schlecht) bis 10 (super)
6     private int torschuss;    // von 1 (schlecht) bis 10 (super)
7     private int motivation;  // von 1 (schlecht) bis 10 (super)
8     private int tore;
9
10    // Konstruktoren
11    public Spieler(String n, int a, int s, int t, int m){
12        super(n, a);
13        staerke      = s;
14        torschuss    = t;
15        motivation   = m;
16        tore         = 0;
17    }
18
19    // Funktionen (get und set):
20    ...
21
22    // Spielerfunktionen:
23
24    // Der spieler hat ein Tor geschossen
25    public void addTor(){
26        tore++;
27    }
28
29    // eine Zahl von 1-10 liefert die Qualität des Torschusses mit
30    // einem kleinen Zufallswert +1 oder -1
31    public int schiesstAufTor(){
32        Random r = new Random();
33        // Entfernungspauschale :
34        torschuss = Math.max(1, Math.min(10, torschuss - r.nextInt(3)));
35        // +-1 ist hier die Varianz
36        int ret = Math.max(1, Math.min(10, torschuss + r.nextInt(3)-1));
37        return ret;
38    }
39}
```

Die Klasse **Spieler** hat eine Funktion `schiesstAufTor`, falls dieser Spieler in der Partie eine Torchance erhält, dann wird eine zufällige Zahl im Bereich von 1 – 10 gewählt, die abhängig von der Torschussqualität des Spielers ist. Der Torwart erhält eine Funktion, die entscheidet, ob der abgegebene Torschuss pariert oder durchgelassen wird (ebenfalls mit einem zufälligen Ausgang):

```

1 import java.util.Random;
2
3 public class Torwart extends Spieler{
4     // Zusätzliche Eigenschaften eines Torworts:
5     private int reaktion;
6
7     // Konstruktoren
8     public Torwart(String n, int a, int s, int t, int m, int r){
```

```

9     super(n, a, s, t, m);
10    reaktion = r;
11 }
12
13 // Funktionen (get und set):
14 ...
15
16 // Torwartfunktionen:
17
18 // Als Parameter erhält der Torwart die Torschussstärke und nun muss
19 // entschieden werden, ob der Torwart hält oder nicht
20 public boolean haeltDenSchuss(int schuss){
21     Random r = new Random();
22     // +1 ist hier die Varianz
23     int ret = Math.max(1, Math.min(10, reaktion + r.nextInt(3)-1));
24     if (ret>=schuss)
25         return true; // gehalten
26     else
27         return false; // TOR!!!
28 }
29 }
```

## 7.1.2 Die Mannschaft

Wir haben die Spieler und den Trainer modelliert. Jetzt ist es an der Zeit, eine Mannschaft zu beschreiben. Eine **Mannschaft** ist eine Klasse mit den Eigenschaften **name**, **Trainer**, **Torwart** und **Spieler**. Es gibt wieder einen Konstruktor und die get-set-Funktionen.

```

1 public class Mannschaft{
2     // Eigenschaften einer Mannschaft:
3     private String name;
4     private Trainer trainer;
5     private Torwart torwart;
6     private Spieler[] kader;
7
8     // Konstruktoren
9     public Mannschaft(String n, Trainer t, Torwart tw, Spieler[] s){
10        name          = n;
11        trainer       = t;
12        torwart      = tw;
13        kader        = s;
14    }
15
16    // Funktionen (get und set):
17    ...
18
19    // Mannschaftsfunktionen:
20
21    // liefert die durchschnittliche Mannschaftsstärke
22    public int getStaerke(){
23        int summ = 0;
24        for (int i=0; i<10; i++)
25            summ += kader[i].getStaerke();
26        return summ/10;
27    }
28
29    // liefert die durchschnittliche Mannschaftsmotivation
30    public int getMotivation(){
31        int summ = 0;
```

```

32     for (int i=0; i<10; i++)
33         summ += kader[i].getMotivation ();
34     return summ/10;
35 }
36 }
```

Zusätzlich besitzt die Klasse **Mannschaft** die Funktionen `getStaerke` und `getMotivation`, die die durchschnittliche Stärke, bzw. Motivation der Mannschaft als Zahlenwert wiedergeben.

### 7.1.3 Turniere, Freundschaftsspiele

Die Klasse **Mannschaft** und alle dazugehörigen Klassen **Spieler**, **Torwart** und **Trainer** wurden definiert. Es ist an der Zeit, ein Freundschaftsspiel zweier Mannschaften zu realisieren. Auf die hier vorgestellte Weise könnten ganze Ligen und Turniere implementiert werden. Das wird Teil der Übungsaufgaben und der eigenen Motivation sein.

Wir lernen dabei eine weitere wichtige Vererbungsvariante kennen. Zunächst beschreiben wir allgemein, wie wir uns ein Freundschaftsspiel vorstellen und implementieren es dann.

#### 7.1.3.1 Ein Interface festlegen

Ein Freundschaftsspiel findet zwischen zwei Mannschaften statt. Es könnten Schach-, Hockey- oder z. B. Fußballmannschaften sein. Das wissen wir nicht genau, aber wir wissen, dass jede Mannschaft einen Namen und ihre im Spiel erreichten Punkte hat. Dann könnten wir einen Ergebnistext, der das Resultat beschreibt, ebenfalls als Funktion verlangen. Das sollte erst einmal genügen.

Wir implementieren jetzt eine Schnittstelle Freundschaftsspiel, an der sich alle anderen Programmierer orientieren können. Jeder, der ein Freundschaftsspiel, so wie wir es verstehen, implementieren möchte, kann sich an dieser Schnittstelle orientieren und bleibt mit unserem Kontext kompatibel. Es werden in diesem **Interface** (Schnittstelle) nur die Funktionen beschrieben, die jeder implementieren muss. Es gibt keine funktionsfähigen Programme, sondern nur die Funktionsköpfe.

```

1 public interface Freundschaftsspiel{
2     String getHeimMannschaft();
3     String getGastMannschaft();
4     int getHeimPunkte();
5     int getGastPunkte();
6     String getErgebnisText();
7 }
8 }
```

Anhand dieses Interfaces können wir speziell für den Fußball eine neue Klasse **FussballFreundschaftsspiel** entwerfen. Wir **müssen** alle vorgegebenen Funktionen eines Interfaces implementieren. Es können noch mehr dazu kommen, aber es dürfen keine fehlen.

### *Implementierungsbeispiel des Interfaces Freundschaftsspiel*

```

1 import java.util.Random;
2
3 public class Fussballfreundschaftsspiel implements Freundschaftsspiel{
4     private String nameHeimMannschaft;
5     private String nameGastMannschaft;
6     private int punkteHeim;
7     private int punkteGast;
8
9     // Konstruktor
10    public Fussballfreundschaftsspiel(){
11        punkteHeim      = 0;
12        punkteGast      = 0;
13    }
14
15    // Methoden des Interface , die implementiert werden müssen:
16    public String getHeimMannschaft(){
17        return nameHeimMannschaft;
18    }
19
20    public String getGastMannschaft(){
21        return nameGastMannschaft;
22    }
23
24    public int getHeimPunkte(){
25        return punkteHeim;
26    }
27
28    public int getGastPunkte(){
29        return punkteGast;
30    }

```

Nachdem die Funktionen `getHeimMannschaft`, `getGastMannschaft`, `getHeimPunkte` und `getGastPunkte` implementiert sind, folgt die Methode `starteSpiel`.

```

32 // Ein Fussballfreundschaftsspiel zwischen beiden Mannschaften wird
33 // gestartet.
34 public void starteSpiel(Mannschaft m1, Mannschaft m2){
35     nameHeimMannschaft      = m1.getName();
36     nameGastMannschaft      = m2.getName();
37     punkteHeim              = 0;
38     punkteGast              = 0;
39
40     // jetzt starten wir das Spiel und erzeugen für die 90 Minuten
41     // Spiel plus Nachspielzeit die verschiedenen Aktionen
42     // (wahrscheinlichkeitsbedingt) für das Freundschaftsspiel
43     Random r = new Random();
44
45     boolean spiellaeuft = true;
46     int spieldauer       = 90 + r.nextInt(5);
47     int zeit             = 1;
48     int naechsteAktion;
49
50     // solange das Spiel laeuft , koennen Torchancen entstehen ...

```

```

51     while (spiellaeuft){
52         naechsteAktion = r.nextInt(15)+1;
53
54         // Ist das Spiel schon zu Ende?
55         if ((zeit + naechsteAktion>spieldauer )||( zeit>spieldauer )){
56             spiellaeuft = false;
57             break;
58         }
59
60         // ****
61         // Eine neue Aktion findet statt ...
62         zeit = zeit + naechsteAktion;

```

Jetzt müssen wir entscheiden, wer den Torschuss abgibt, wie stark er schießt und ob der Torwart den Ball hält. Das berechnen wir nach folgender Wahrscheinlichkeitsverteilung:

1. Wähle eine Mannschaft für eine Torchance aus, als Kriterien dienen Stärke und Motivation der Mannschaften sowie die Erfahrung des Trainers. Nach der Berechnung aller Einflüsse, hat die bessere Mannschaft eine größere Torchance.

```

64     // Einfluss der Motivation auf die Stärke:
65     float staerke_1 = (m1.getStaerke ()/2.0f) +
66         ((m1.getStaerke ()/2.0f) * (m1.getMotivation ()/10.0f));
67     float staerke_2 = (m2.getStaerke ()/2.0f) +
68         ((m2.getStaerke ()/2.0f) * (m2.getMotivation ()/10.0f));
69
70     // Einfluss des Trainers auf die Stärke:
71     int abweichung = r.nextInt(2);
72     if (staerke_1 > m1.getTrainer ().getErfahrung ())
73         abweichung = -abweichung;
74     staerke_1 = Math.max(1, Math.min(10, staerke_1 + abweichung));
75
76     abweichung = r.nextInt(2);
77     if (staerke_2 > m2.getTrainer ().getErfahrung ())
78         abweichung = -abweichung;
79     staerke_2 = Math.max(1, Math.min(10, staerke_2 + abweichung));

```

2. Wähle zufällig einen Spieler aus dieser Mannschaft, berechne den Torschuss und gib dem Torwart der anderen Mannschaft die Möglichkeit, diesen Ball zu halten.

```

81     int schuetze      = r.nextInt(10);
82
83     if ((r.nextInt(Math.round(staerke_1+staerke_2))-staerke_1)<=0){
84         Spieler s      = m1.getKader ()[schuetze ];
85         Torwart t      = m2.getTorwart ();
86         int torschuss    = s.schiessstAufTor ();
87         // haelt er den Schuss?
88         boolean tor      = !t.haeltDenSchuss(torschuss );
89
90         System.out.println();
91         System.out.println(zeit+" Minute: " );
92         System.out.println(" Chance fuer "+m1.getName ()+" ... ");
93         System.out.println(" "+s.getName ()+" zieht ab" );
94
95         if (tor) {
96             punkteHeim++;
97             s.addTor ();

```

```

98         System.out.println("    TOR!!!      "+punkteHeim+":"
99             "+punkteGast+" "+s.getName()+" ("+s.getTore()+" )");
100        } else {
101            System.out.println("    "+m2.getTorwart().getName()
102                            +" pariert glanzvoll.");
103        }
104    } else {
105        Spieler s      = m2.getKader()[schuetze];
106        Torwart t     = ml.getTorwart();
107        int torschuss   = s.schiesstAufTor();
108        boolean tor     = !t.haeltDenSchuss(torschuss);
109
110        System.out.println();
111        System.out.println(zeit+".Minute: ");
112        System.out.println("    Chance fuer "+m2.getName()+" ...");
113        System.out.println("    "+s.getName()+" zieht ab");
114
115        if (tor) {
116            punkteGast++;
117            s.addTor();
118            System.out.println("    TOR!!!      "+punkteHeim+":"
119                            "+punkteGast+" "+s.getName()+" ("+s.getTore()+" )");
120        } else {
121            System.out.println("    "+ml.getTorwart().getName()
122                            +" pariert glanzvoll.");
123        }
124    }
125    // *****
126}
127

```

Es fehlt für die vollständige Implementierung aller Funktionen des Interfaces noch die Methode `getErgebnisText`.

```

129    public String getErgebnisText(){
130        return "Das Freundschaftsspiel endete \n\n"+nameHeimMannschaft
131            +" - "+nameGastMannschaft+" "+punkteHeim+":"
132            "+punkteGast+".";
133    }
134

```

Jetzt haben wir alle notwendigen Funktionen implementiert und können schon im nächsten Abschnitt mit einem Spiel beginnen.

### 7.1.3.2 Freundschaftsspiel Deutschland–Brasilien der WM-Mannschaften von 2006

Nach der ganzen Theorie und den vielen Programmzeilen können wir uns zurücklehnen und ein, bei der WM 2006 nicht stattgefundenes Spiel Deutschland gegen Brasilien, anschauen. Dazu müssen wir zunächst beide Mannschaften definieren<sup>1</sup> und anschließend beide mit der Klasse Fußballfreundschaftsspiel eine Partie spielen lassen.

<sup>1</sup> Die Daten, die in diesem Beispiel verwendet werden, eignen sich gut, um sie in eine Datei auszulagern.

```

1 public class FussballTestKlasse{
2     public static void main(String[] args){
3         // ****
4         // Mannschaft 1
5         Trainer t1      = new Trainer("Juergen Klinsmann", 34, 9);
6         Torwart tw1    = new Torwart("J. Lehmann", 36, 8, 1, 9, 7);
7
8         Spieler[] sp1   = new Spieler[10];
9         sp1[0]          = new Spieler("P. Lahm", 23, 9, 5, 9);
10        sp1[1]         = new Spieler("C. Metzelder", 25, 8, 2, 7);
11        sp1[2]         = new Spieler("P. Mertesacker", 22, 9, 2, 8);
12        sp1[3]         = new Spieler("M. Ballack", 29, 7, 5, 8);
13        sp1[4]         = new Spieler("T. Borowski", 26, 9, 8, 9);
14        sp1[5]         = new Spieler("D. Odonkor", 22, 7, 5, 8);
15        sp1[6]         = new Spieler("B. Schweinsteiger", 22, 2, 3, 2);
16        sp1[7]         = new Spieler("L. Podolski", 21, 7, 8, 9);
17        sp1[8]         = new Spieler("M. Klose", 28, 10, 9, 7);
18        sp1[9]         = new Spieler("O. Neuville", 33, 8, 8, 7);
19         // ****
20
21         // ****
22         // Mannschaft 2
23         Trainer t2      = new Trainer("Carlos Alberto Parreira", 50, 3);
24         Torwart tw2    = new Torwart("Dida", 25, 9, 1, 6, 8);
25
26         Spieler[] sp2   = new Spieler[10];
27         sp2[0]          = new Spieler("Cafu", 33, 8, 4, 6);
28         sp2[1]          = new Spieler("R. Carlos", 32, 9, 9, 2);
29         sp2[2]          = new Spieler("Lucio", 29, 10, 9, 9);
30         sp2[3]          = new Spieler("Ronaldinho", 25, 10, 9, 5);
31         sp2[4]          = new Spieler("Zé Roberto", 27, 7, 7, 4);
32         sp2[5]          = new Spieler("Kaká", 22, 10, 8, 10);
33         sp2[6]          = new Spieler("Juninho", 26, 7, 10, 3);
34         sp2[7]          = new Spieler("Adriano", 23, 8, 8, 4);
35         sp2[8]          = new Spieler("Robinho", 19, 9, 8, 9);
36         sp2[9]          = new Spieler("Ronaldo", 28, 4, 10, 2);
37         // ****
38
39         Mannschaft m1 = new Mannschaft("Deutschland WM 2006", t1, tw1, sp1);
40         Mannschaft m2 = new Mannschaft("Brasilien WM 2006", t2, tw2, sp2);
41         Fussballfreundschaftsspiel f1 = new Fussballfreundschaftsspiel();
42
43         System.out.println("-----");
44         System.out.println("Start des Freundschaftspiels zwischen");
45         System.out.println();
46         System.out.println(m1.getName());
47         System.out.println(" Trainer: "+m1.getTrainer().getName());
48         System.out.println();
49         System.out.println(" und ");
50         System.out.println();
51         System.out.println(m2.getName());
52         System.out.println(" Trainer: "+m2.getTrainer().getName());
53         System.out.println("-----");
54
55         f1.starteSpiel(m1, m2);
56
57         System.out.println();
58         System.out.println("-----");
59         System.out.println(f1.getErgebnisText());
60         System.out.println("-----");
61     }
62 }
```

Das Spiel ist vorbereitet und die Akteure warten ungeduldig auf den Anpfiff ...

Folgender Spielverlauf mit einem gerechten Endstand von 1 : 1 wurde durch unser Programm erzeugt<sup>2</sup>:

```
C:\Java\FussballManager>javac FussballTestKlasse.java
C:\Java\FussballManager>java FussballTestKlasse
```

---

Start des Freundschaftsspiels zwischen

Deutschland WM 2006  
Trainer: Juergen Klinsmann

und

Brasilien WM 2006  
Trainer: Carlos Alberto Parreira

---

12. Minute:

Chance fuer Deutschland WM 2006 ...  
T. Borowski zieht ab  
Dida pariert glanzvoll.

23. Minute:

Chance fuer Brasilien WM 2006 ...  
ZÚ Roberto zieht ab  
J. Lehmann pariert glanzvoll.

31. Minute:

Chance fuer Deutschland WM 2006 ...  
T. Borowski zieht ab  
Dida pariert glanzvoll.

41. Minute:

Chance fuer Deutschland WM 2006 ...  
T. Borowski zieht ab  
Dida pariert glanzvoll.

44. Minute:

Chance fuer Deutschland WM 2006 ...  
C. Metzelder zieht ab  
Dida pariert glanzvoll.

Halbzeitpfiff. Kurz durchatmen und weiter geht es:

54. Minute:

Chance fuer Brasilien WM 2006 ...  
R. Carlos zieht ab  
TOR!!! 0:1 R. Carlos (1)

55. Minute:

Chance fuer Deutschland WM 2006 ...  
M. Klose zieht ab  
TOR!!! 1:1 M. Klose (1)

68. Minute:

Chance fuer Deutschland WM 2006 ...  
P. Lahm zieht ab  
Dida pariert glanzvoll.

---

<sup>2</sup> An dieser Stelle muss ich zugeben, dass es, aus deutscher Sicht, der beste von fünf Durchläufen war :).

79. Minute:  
 Chance fuer Deutschland WM 2006 ...  
 C. Metzelder zieht ab  
 Dida pariert glanzvoll.

84. Minute:  
 Chance fuer Deutschland WM 2006 ...  
 P. Lahm zieht ab  
 Dida pariert glanzvoll.

---

Das Freundschaftsspiel endete

Deutschland WM 2006 – Brasilien WM 2006 1:1.

---

Das Beispielprogramm kann als Basis für einen professionelleren Fußballmanager verwendet werden. Im Aufgabenteil sind ein paar Anregungen zu finden.

### 7.1.3.3 Interface versus abstrakte Klasse

Im Vergleich zu einem **Interface**, bei dem es nur Funktionsköpfe gibt, die implementiert werden müssen, gibt es bei der **abstrakten Klasse** die Möglichkeit, Funktionen bereits bei der Definition der Klasse zu implementieren. Eine abstrakte Klasse muss mindestens eine **abstrakte Methode** (also ohne Implementierung) besitzen. Demzufolge ist das Interface ein Spezialfall der abstrakten Klasse, denn ein Interface besteht nur aus abstrakten Methoden und Konstanten.

Schauen wir uns ein ganz kurzes Beispiel dazu an. Die Klasse **A** verwaltet die Variable `wert` und bietet bereits die Methode `getWert()`. Die Methode `setWert()` soll aber implementiert werden und deshalb wird sie mit dem Schlüsselwort `abstract` versehen.

```

1 public abstract class A{
2     protected int wert;
3
4     public int getWert(){
5         return wert;
6     }
7
8     public abstract void setWert(int w);
9 }
```

Die Klasse **B** erbt die Informationen der Klasse **A**, also `wert` und die Methode `getWert()`, muss aber die Methode `setWert()` implementieren. Durch das Schlüsselwort `protected` ist es der Klasse **B** nach der Vererbung erlaubt, auf die Variable `wert` zuzugreifen.

```

1 public class B extends A{
2     public void setWert(int w){
3         this.wert = w;
4     }
5 }
```

Jetzt nehmen wir noch eine Testklasse **Tester** dazu und testen mit ihr die gültige Funktionalität. Als erstes wollen wir versuchen, eine Instanz der Klasse **A** zu erzeugen.

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



Das schlägt mit der folgenden Ausgabe fehl:

```
C:\JavaCode>javac Tester.java
Tester.java:3: A is abstract; cannot be instantiated
    A a = new A();
               ^
1 error
```

Es ist nicht erlaubt von einer abstrakten Klasse eine Instanz zu erzeugen!

```
B b = new B();
b.setWert(4);
System.out.println(" "+b.getWert());
```

Wir erzeugen eine Instanz der Klasse **B**, die nun keine abstrakten Methoden enthält und verwenden beide Methoden.

```
C:\JavaCode>javac Tester.java
C:\JavaCode>java Tester
4
```

Mit Erfolg.

## 7.2 Aufarbeitung der vorhergehenden Kapitel

Bevor wir an dieser Stelle weitermachen, werden wir noch einmal anhand der in den vorhergehenden Kapiteln verwendeten Klassen ein paar weitere Konzepte von Java besprechen.

### 7.2.1 Referenzvariablen

Zur Erinnerung, wir haben in Abschnitt 7.1.1.5 die Klasse **Person** implementiert. Hier noch einmal der Programmcode dieser Klasse:

```

1 public class Person{
2   // Eigenschaften einer Person:
3   private String name;
4   private int alter;
5
6   // Konstruktoren
7   public Person(String n, int a){
8     name = n;
9     alter = a;
10  }
11
12  // Funktionen (get und set):
13  ...
14 }
```

Um zu verstehen, dass Referenzen Adressverweise auf einen reservierten Speicherplatz sind, schauen wir uns folgendes Beispiel an.

```

Person p1;
p1 = new Person("Hugo", 12);
Person p2;
p2 = p1;
if (p1 == p2)
  System.out.println("Die Referenzen sind gleich");
```

In der ersten Zeile wird `p1` deklariert. `p1` ist eine Referenzvariable und beinhaltet eine Adresse. Diese Adresse ist momentan nicht gegeben, sollten wir versuchen auf `p1` zuzugreifen, würde Java einen Fehler beim Kompilieren mit der Begründung verursachen: „Variable `p1` ist nicht initialisiert“. Die zweite Zeile stellt nun aber einen Speicherplatz bereit und erzeugt ein Objekt der Klasse **Person** und vergibt den Attributen gleich Werte. `p1` zeigt nun auf diesen Speicherbereich.

Die dritte Zeile verhält sich äquivalent zur ersten. In Zeile vier weisen wir aber nun die Adresse von `p1` der Variablen `p2` zu. Beide Variablen zeigen nun auf den gleichen Speicherplatz, auf dasselbe Objekt. Sollten wir Veränderungen in `p1` vornehmen, so treten diese Veränderungen ebenfalls bei `p2` auf, denn es handelt sich um dasselbe Objekt!

Das ist auch der Grund für das Ergebnis von Übung 6 aus Kapitel 5.

### 7.2.2 Zugriff auf Attribute und Methoden durch Punktnotation

Wir haben diese Syntax bereits schon mehrfach verwendet, aber nun werden wir es konkretisieren. Existiert eine Referenz auf ein Objekt, so können wir mit einem Punkt nach der Referenz und dem Namen des entsprechenden Attributs (das nennen wir dann **Instanzvariable**) bzw. der entsprechenden Methode (analog **Instanzmethode**) auf diese zugreifen.

`Referenz.Attribut`  
`Referenz.Methode`

Dazu schauen wir uns ein kleines Beispiel an und werden den Unterschied zwischen Variablen, die primitive Datentypen repräsentieren und Variablen, die Referenzen repräsentieren, erläutern.

```
int a = 2;
Person p = new Person("Hans", 92);
// An dieser Stelle hat p.getName() den Rückgabewert "Hans"
komischeFunktion(a, p);
```

Wir werden nun a und p an eine Funktion übergeben. Für den primitiven Datentypen a gilt die Übergabe „**call by value**“. Das bedeutet, dass der Inhalt, also die 2 in die Funktion übergeben wird. Anders sieht es bei dem Referenzdatentyp p aus, hier wird die Referenz, also die Adresse übergeben, wir nennen das „**call by reference**“. Sehen wir nun, welche Auswirkungen das hat:

```
public void komischeFunktion(int x, Person y){
    // Wir ändern die Werte der Eingabeparameter.
    x = 7;
    y.setName("Gerd");
}
```

Sollten wir nach Ausführung der Zeile komischeFunktion(a, p); wieder den Rückgabewert von p erfragen, so erhalten wir "Gerd". Die Variable a bleibt indes unangetastet.

### 7.2.3 Die Referenzvariable this

Wir haben schon in Abschnitt 7.1.1.3 gesehen, wie sich die Instanzvariablen von „außen“ setzen lassen. Jedes Objekt kann aber auch mittels einer Referenzvariablen auf sich selbst refflektieren und seine Variablen und Funktionen ansprechen. Dazu wurde die Klasse **Person** im folgenden Beispiel vereinfacht.

```
1 public class Person{
2     private String name;
3     public void setName(String name){
4         this.name = name;
5     }
6 }
```

Wir können mit der Referenzvariablen **this** auf „unsere“ Instanzvariablen und -methoden zugreifen.

### 7.2.4 Prinzip des Überladens

Oft ist es sinnvoll, eine Funktion für verschiedene Eingabetypen zur Verfügung zu stellen. Damit wir nicht jedes Mal einen neuen Funktionsnamen wählen müssen, gibt es die Möglichkeit der Überladung in Java. Wir können einen Funktionsnamen mehrfach verwenden, falls sich die Signatur der Funktionen eindeutig unterscheiden lässt. Als Signatur definieren wir die Kombination aus *Rueckgabewert*, *Funktionsname* und *Eingabeparameter*, wobei nur der Typ der Eingabe und nicht die Bezeichnung entscheidend ist.

Schauen wir uns ein Beispiel an:

```
// Max-Funktion für Datentyp int
int max(int a, int b){
    if (a<b) return b;
    return a;
}
// Max-Funktion für Datentyp double
double max(double a, double b){
    ... // analog zu der ersten Funktion
}
```

Beim Aufruf der Funktion `max` wird die passende Signatur ermittelt und diese Funktion entsprechend verwendet. Wir können uns merken, trotz Namensgleichheit sind mehrere Funktionen mit verschiedenen Signaturen erlaubt.

### 7.2.5 Überladung von Konstruktoren

Dieses Prinzip lässt sich auch auf Konstruktoren übertragen. Nehmen wir wieder als Beispiel die Klasse **Person** und erweitern die Konstruktoren. Dann haben wir die Möglichkeit, auch ein Objekt der Klasse **Person** zu erzeugen, wenn nur der Name, aber nicht das Alter bekannt sein sollte.

```
1 public class Person{
2     private String name;
3     private int alter;
4     // Konstruktor 1
5     public Person(){
6         name      = "";    // z.B. als Defaultwert
7         alter     = 1;     // z.B. als Defaultwert
8     }
9     // Konstruktor 2
10    public Person(String name){
11        this.name = name;
12        alter     = 1;     // z.B. als Defaultwert
13    }
14    // Konstruktor 3
15    public Person(String name, int alter){
16        this.name = name;
17        this.alter = alter;
18    }
19 }
```

Kleiner Hinweis: Genau dann, wenn kein Konstruktor angegeben sein sollte, denkt sich Java den **Defaultkonstruktor** hinzu, der keine Parameter erhält und keine Attribute setzt. Es kann trotzdem ein Objekt dieser Klasse erzeugt werden.

Nun aber ein Beispiel zu der neuen **Person**-Klasse:

```
Person p1 = new Person();
Person p2 = new Person("Herbert", 30);
```

Die erste Zeile verwendet den passenden Konstruktor (Konstruktor 1) mit der parameterlosen Signatur und die zweite verwendet den dritten Konstruktor.

### 7.2.5.1 Der Copy-Konstruktor

Bei der Übergabe von Objekten werden diese nicht kopiert, sondern lediglich die Referenz übergeben, das wissen wir bereits. Wir müssen also immer daran denken, eine lokale Kopie des Objekts vorzunehmen. Elegant lässt sich das mit dem Copy-Konstruktor lösen. Wir legen einfach bei der Implementierung einer Klasse, zusätzlich zu den vorhandenen Konstruktoren, einen weiteren Konstruktor an, der als Eingabeparameter einen Typ der Klasse selbst enthält.

Hier ein kurzes Beispiel:

```
1 public class CopyKlasse{
2     public int a, b, c, d;
3
4     public CopyKlasse(){
5         a=0, b=0, c=0, d=0;
6     }
7
8     public CopyKlasse(CopyKlasse ck){
9         this.a = ck.a;
10        this.b = ck.b;
11        this.c = ck.c;
12        this.d = ck.d;
13    }
14 }
```

Zuerst erzeugen wir eine Instanz der Klasse **CopyKlasse** und anschließend erstellen wir eine Kopie:

```
CopyKlasse a = new CopyKlasse();
CopyKlasse copyVonA = new CopyKlasse(a);
```

## 7.2.6 Garbage Collector

Bisher haben wir andauernd Speicher für unsere Variablen reserviert, aber wann wird dieser Speicher wieder frei gegeben? Java besitzt mit dem Garbage Collector eine „Müllabfuhr“, die sich automatisch um die Freigabe des Speichers kümmert. Das ist gegenüber anderen Programmiersprachen ein Komfort. Dieser Komfort ist nicht umsonst, Java entscheidet eigenständig, den Garbage Collector zu starten. Es könnte also auch dann geschehen, wenn gerade eine performancekritische Funktion gestartet wird.

Um es nochmal deutlich zu machen, in Java braucht sich der Programmierer nicht um das Speichermanagement zu kümmern.

## 7.2.7 Statische Attribute und Methoden

Im erweiterten Klassenkonzept war die Nutzung von Attributen und Methoden an die Existenz von Objekten gebunden.

```
Random r = new Random();
int zuffi = r.nextInt(7);
```

Um die Funktion `nextInt` der Klasse **Random** verwenden zu können, mussten wir als erstes ein Objekt der Klasse erzeugen.

Mit dem Schlüsselwort `static` können wir nun aus Instanzvariablen und -methoden unabhängig verwendbare **Klassenvariablen** und **-methoden** umwandeln. Schauen wir uns dazu zunächst ein Beispiel für die Verwendung von Klassenvariablen an. Wir wollen zählen, wie viele Instanzen einer Klasse erzeugt werden.

```
1 public class Tasse{
2     public static int zaehler=0;
3     public Tasse(){
4         zaehler++;
5     }
6     public int getZaehler(){
7         return zaehler;
8     }
9 }
```

Testen wir unser Programm:

```
Tasse t1 = new Tasse();
Tasse t2 = new Tasse();
Tasse t3 = new Tasse();
int k = t1.getZaehler();
```

Die Variable `k` hat nach dem dreifachen Aufruf des Konstruktors den Wert 3. Unabhängig welche Instanz wir dazu befragen würden. Die statische Variable `zaehler` existiert nur einmal und wird von allen Instanzen geteilt.

Das gleiche gilt für Funktionen.

```

1 public class MeinMathe{
2     public static int max(int a, int b){
3         return a<b ? b : a;
4     }
5 }
```

In Java gibt es die Möglichkeit ein `if`-Statement folgendermaßen auszudrücken:

```
<Bedingung> ? <Anweisung wenn Bedingung true> :  
<Anweisung wenn false>
```

In der Klasse **MeinMathe** könnten beispielsweise verschiedene Funktionen zusammengetragen werden.

Da nun die Funktion `max` statisch gemacht wurde, können wir sie verwenden, ohne ein Objekt der Klasse **MeinMathe** zu erzeugen.

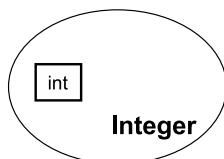
```
int d = MeinMathe.max(3, 22);
```

Sollten wir das `static` weglassen, so müssen wir wieder ein Objekt erzeugen und können durch die Referenz an die Funktion gelangen:

```
MeinMathe m = new MeinMathe();  
int d = m.max(3, 22);
```

## 7.2.8 Primitive Datentypen und ihre Wrapperklassen

Zu jedem primitiven Datentyp gibt es eine entsprechende **Wrapperklasse** (=Hülle). In dieser Klasse werden unter anderem eine Reihe von Konvertierungsmethoden angeboten. Man kann es sich so vorstellen, dass im Kern dieser Wrapperklasse der primitive Datentyp gespeichert ist und drum herum weitere Funktionen existieren. Beispielsweise existiert für den primitiven Datentypen `int` die Wrapperklasse **Integer**.



Schauen wir uns dazu einen kleinen Programmabschnitt an:

```
int a = 4;
Integer i = new Integer(a);
int b = i.intValue();

String s = i.toString();
String l = "7";
Integer k = new Integer(l);
```

In der zweiten Zeile wird ein Objekt der Klasse **Integer** erzeugt und mit dem int-Wert von a gefüllt. Die Funktion intValue liefert den int-Wert zurück. Über die Referenzvariable i gelangen wir nun an die Instanzmethoden. Eine Methode konvertiert den int-Wert zu einem String (siehe Zeile 5). In Zeile 6 erzeugen wir einen String mit der Zeichenkette "7". Die siebente Zeile zeigt nun, wie wir ein Integerobjekt erzeugen, indem wir im Konstruktor einen String übergeben.

Die Wrapperklassen für die primitiven Datentypen heißen: **Boolean**, **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float** und **Double**. Neben den Konvertierungsmethoden liefern die Wrapperklassen auch die Randwerte ihres entsprechenden Wertebereichs.

```
byte b_min = Byte.MIN_VALUE; // liefert kleinstmöglichen Wert (-128)
float f_max = Float.MAX_VALUE;
```

## 7.2.9 Die Klasse String

Ein **String** repräsentiert eine Zeichenkette. Objekte vom Typ **String** sind nach der Initialisierung nicht mehr veränderbar.

```
String name1 = "Frodo";
String name2 = new String("Bilbo");
String name3 = "Beutlin";

// Verkettung von Zeichenketten
String zusammen1 = name3 + ", " + name2;
String zusammen2 = name3.concat(", ") + name1;
String zusammen3 = (name3.concat(", ")).concat(name1);
System.out.println("zusammen1: "+zusammen1); // Beutlin , Bilbo
System.out.println("zusammen2: "+zusammen2); // Beutlin , Frodo
System.out.println("zusammen3: "+zusammen3); // Beutlin , Frodo

// Länge einer Zeichenkette ermitteln
int laenge = name1.length();

// Teile einer Zeichenkette extrahieren
String teil = name1.substring(2, 5); // Zeile 17
System.out.println("teil: "+teil);
```

Die erste Zeile zeigt uns eine angenehme Eigenschaft von Java. Wir müssen nicht jedes Mal den **String**-Konstruktor aufrufen, wenn eine Zeichenkette erzeugt werden soll. Im Prinzip geschieht aber genau das. Zeile zwei zeigt, wie Java die erste

Zeile interpretiert (es wird für unser Beispiel aber eine neue Zeichenkette erzeugt). Wir können nun die Zeichenketten verketten, also hintereinander hängen und so mit eine größere Zeichenkette konstruieren. Die Zeilen 6-8 zeigen die verschiedenen Möglichkeiten. Der Operator +, wie er in Zeile 6 angewendet wird, kann als Aufruf der concat-Funktion interpretiert werden. In Zeile 14 sehen wir die Verwendung der Längenfunktion, dabei wird die Anzahl der Zeichen (auch Leer- oder Sonderzeichen!) gezählt. Zeile 17 zeigt ein kleines Beispiel der Verwendung der substring-Methode, wobei der erste Index dem Startindex in der Zeichenkette entspricht (begonnen wird wie bei den Arrays mit dem Index 0) und der zweite dem Endindex.

### 7.2.9.1 Vergleich von Zeichenketten

Wenn wir uns daran erinnern, dass wir es auch in diesem Beispiel mit Referenzvariablen zu tun haben, dann erklärt es sich einfach, dass Stringvergleiche einer Methode bedürfen.

```
String name1 = "Hugo";
String name2 = "Hugo";
if (name1 == name2)
    ...
```



Die beiden Variablen name1 und name2 sind Referenzvariablen. Sie repräsentieren jeweils eine Adresse im Speicher. Wenn wir nun versuchen diese Adressen zu vergleichen, erhalten wir selbstverständlich ein false. Es sind ja zwei verschiedene Objekte. Was wir aber eigentlich mit einem Vergleich meinen, ist die Überprüfung des Inhalts beider Strings.

```
String name1 = "Hugo";
String name2 = "Hugo";
if (name1.equals(name2))
    ...
```

Mit der von der Stringklasse angebotenen Funktion equals lassen sich Strings auf ihren Inhalt vergleichen. Nun liefert der Vergleich auch true.

Ein kleiner Rückblick an dieser Stelle, wir erinnern uns an die Klasse **String** aus dem Kapitel 7.1 :

```
1 public class Person{
2     // Eigenschaften einer Person:
3     private String name;
4     private int alter;
5
6     public void setName(String name){
7         this.name = name;
8     }
9     ...
10 }
```

Wenn wir uns dieses Beispiel anschauen, dann erinnern wir uns an die Problematik mit der Übergabe von Referenzen an Funktionen. Nichts anderes geschieht in diesem Beispiel. Falls es möglich wäre, den Inhalt des Strings `name` in der Klasse **Person** zu ändern, würde das auch einen Einfluss auf den übergebenen String, der vor dem Aufruf der Funktion `setName` existiert haben könnte, haben. Die Tatsache, dass es hier trotzdem erlaubt ist und auch problemlos funktioniert liegt daran, dass der Inhalt eines Strings nach der Initialisierung nicht mehr verändert werden kann! Falls wir der Instanzvariablen `name` einen anderen String zuweisen, wird auch immer ein neues Objekt erzeugt.

## 7.3 Zusammenfassung und Aufgaben

### Zusammenfassung

In diesem sehr umfangreichen Kapitel haben Sie das Konzept der Objektorientierung kennengelernt. Generalisierung und Spezialisierung sind die Schlüssel für das Verständnis und die Motivation zum Vererbungskonzept. Klassen verstehen wir jetzt als Baupläne für Objekte und wissen, wie Instanzen einer Klasse zu erzeugen und zu verwenden sind. Konstruktoren helfen uns, die Instanzen vernünftig zu initialisieren und get-set-Funktionen die Klassenattribute zu manipulieren.

Klassen können wir direkt erweitern oder über Interfaces und abstrakte Klassen Baupläne für Implementierungen einer Klasse festlegen.

Das Kapitel war deshalb so umfangreich, weil wir offene Fragen und schwierige Passagen aus den vorhergehenden Kapiteln noch einmal erörtert haben. Wichtig in diesem Zusammenhang sind die Begriffe: Referenzvariable, Punktnotation, call by value, call by reference, this, überladen, Garbage Collector, static, Wrapperklassen, protected, String und der Unterschied zwischen Instanzvariablen, bzw. -methoden und Klassenvariablen bzw. -methoden.

### Aufgaben

Übung 1) Überlegen Sie sich analog zu **Spieler**, **Trainer** und **Person** eine Klassenstruktur, die **Studenten** (`name`, `vorname`, `wohnort`, `matrikelnummer`, ...) und **Professoren** (`name`, ..., `gehalt`, `publikationen`, ...) modelliert. Erzeugen Sie Instanzen beider Klassen und experimentieren Sie damit herum.

Übung 2) Erweitern Sie das Fußballmanagerspiel mit folgenden Eigenschaften, bzw. Funktionen:

- (i) Klasse **Spieler**: boolean `verletzt`, int `gelbeKarten`, int `roteKarten`
- (ii) Während des Spiels sollten Aktionen, wie vergebene oder erfolgreiche Torschüsse Einfluss auf die Motivation der Spieler haben.

- (iii) Führen Sie als Aktionen gelbe und rote Karten ein. Sollte ein Spieler beispielsweise eine gelbe Karte erhalten haben, so sinkt seine Motivation.
- (iv) Erweitern Sie den Fußballmanager um einen größeren Kader, bei dem Stamm- und Ersatzspieler verwaltet werden können.
- (v) Schreiben Sie den Turniermodus Ligapokal und lassen verschiedene Mannschaften um den Pokal spielen.

Übung 3) Erläutern Sie das Grundkonzept der Objektorientierung.

Übung 4) Schreiben Sie ein beliebiges kleines Spiel, bei dem zwei Computergegner gegeneinander antreten und entweder zufällig oder hoch intelligent Aktionen ausführen, um zu gewinnen.

## Tag 8: Verwendung von Bibliotheken

Bisher konnten wir recht erfolgreich kleine Probleme durch Java-Programme lösen. Die gesammelten Lösungen müssen wir nicht jedes Mal neu erarbeiten oder die Inhalte von einer Datei in eine andere kopieren. Java bietet dafür das Bibliothekenkonzept. Mit ihm ist es möglich, Funktionen und Klassen zusammenzufassen und später einfach wieder darauf zugreifen zu können.

Java selbst bietet bereits eine Reihe von Bibliotheken standardmäßig an. In diesem Kapitel wollen wir einen Teil dieser Bibliotheken vorstellen und neben der Verwendung aufzeigen, wie eigene Bibliotheken einfach erstellt werden können.

### 8.1 Standardbibliotheken

Bibliotheken in Java werden in **Packages** (Paketen) organisiert. Diese Pakete definieren sinnvolle Gruppen von Klassen und Funktionen, die eine logische Struktur zur Lösung spezifischer Aufgaben liefern.

Zunächst wollen wir mit der Beschreibung einiger Klassen des Pakets `java.lang` beginnen, das sich als wichtigste und meist verwendete Standardbibliothek betrachten lässt. Eine komplette Beschreibung kann der Leser in der Java-Dokumentation [57] finden, die je nach Javaversion variieren kann.

In diesem Beispiel werden die Klassen **Object**, **Integer** und **String** verwendet. Schauen wir uns folgendes Programm an:

```
1 public class DoubleListe {  
2     double liste [];  
3  
4     public DoubleListe(int num) {  
5         liste = new double[num];  
6     }  
7  
8     public static void main(String [] args) {
```

```

9     DoubleListe d1, d2;
10
11    d1 = new DoubleListe(Integer.parseInt(args[0]));
12    d2 = new DoubleListe(Integer.parseInt(args[0]));
13
14    System.out.println(d1);
15    System.out.println(d2);
16
17    if (!d1.equals(d2))
18        System.out.println("Unterschiedliche Listen!");
19    else
20        System.out.println("Gleiche Listen!");
21    }
22 }
```

Nach dem Start mit einem Übergabeparameter erhalten wir:

```
C:\Java>java DoubleListe 3
DoubleListe@187aec
DoubleListe@e48e1b
Unterschiedliche Listen!
```

Der Text mit dem @-Symbol ist die standardmäßige Textdarstellung des Objekts, die aus drei Teilen besteht: (i) die von dem Objekt instanzierte Klasse (in diesem Fall **DoubleListe**), (ii) dem @-Symbol und (iii) die in Java intern verwendete Hexadezimaldarstellung des Objekts.

Die Methode `equals` verwendet diese interne Darstellung zum Vergleich der Objekte, deshalb sind `d1` und `d2` unterschiedlich (obwohl sie das gleiche enthalten). Die Klasse **Objekt** liegt auf dem obersten Platz in der Vererbungsstruktur und ist die Basis für jedes Java-Programm. Die Methoden `equals` und `toString` sind die am meisten verwendeten Methoden der elf in dieser Klasse definierten Methoden. Die eine liefert `true`, falls das Argument einen identischen Inhalt zum betrachteten Objekt hat und die andere bietet eine Stringdarstellung mit dem Inhalt der Listen.

Wenn wir diese Methoden mit folgenden beiden Funktionen überschreiben, können wir die Ausgabe und Funktionalität nach unseren Wünschen anpassen:

```

public String toString() {
    StringBuffer sb = new StringBuffer();
    int i;
    sb.append("DoubleListe[");
    for (i = 0; i < liste.length - 1; i++)
        sb.append(liste[i] + ", ");
    if (i < liste.length)
        sb.append(liste[i]);
    sb.append("]");
    return sb.toString();
}
```

Die Methode `toString` verwendet die Klasse **StringBuffer** zur Erzeugung der Ausgabe. Zuerst wird "DoubleListe[" in den Buffer eingefügt, anschließend die Inhalte der Listenelemente und zu guter Letzt die schließende Klammer "]".

```

public boolean equals(Object obj) {
    if (obj instanceof DoubleListe) {
        DoubleListe dl = (DoubleListe) obj;

        if (dl.liste.length != this.liste.length)
            return false;

        for (int i=0; i<this.liste.length; i++) {
            if (dl.liste[i]!=this.liste[i])
                return false;
        }
        return true;
    }
    return false;
}

```

Die Methode `equals` prüft zunächst, ob `obj` vom Datentyp **DoubleListe** ist und vergleicht anschließend die Inhalte von `obj` mit denen der aktuellen Instanz. Sind alle Vergleiche positiv, so liefert die Funktion ein `true`.

Jetzt haben wir eine Stringdarstellung, die informativ ist, und einen Vergleichsmechanismus, der Typ und Inhalt auf Gleichheit überprüft. Die überschriebene `equals`-Methode verwendet den Operator `instanceof`, der `true` liefert, wenn ein Objekt eine Instanz der Klasse ist. In diesem Fall müssen das Objekt links und die Klasse rechts angegeben werden. Anschließend werden die Inhalte der Listenelemente überprüft.

Testen wir das Programm mit den beiden neuen Funktionen, erhalten wir folgende Ausgabe:

```

C:\Java>java DoubleListe 3
DoubleListe [0.0 ,0.0 ,0.0]
DoubleListe [0.0 ,0.0 ,0.0]
Gleiche Listen!

```

Zum Paket `java.lang` gehören auch die Wrapperklassen (siehe 7.2.8) **Byte**, **Short**, **Boolean**, **Character**, **Integer**, **Long**, **Float** und **Double**, die die entsprechenden primitiven Datentypen als Objekt darstellen.

## 8.2 Mathematik-Bibliothek

Folgende Funktionen bietet die Klasse **Math**:

<b><i>abs</i></b>	liefert den Absolutbetrag
<b><i>min, max</i></b>	Min-, Maxfunktion
<b><i>ceil</i></b>	ganzzahlig aufrunden
<b><i>floor</i></b>	ganzzahlig abrunden
<b><i>round</i></b>	ganzzahlig runden (erste Nachkommastelle)
<b><i>sqrt</i></b>	Quadratwurzel
<b><i>pow</i></b>	Potenzfunktion
<b><i>exp</i></b>	Exponentialfunktion
<b><i>log</i></b>	Logarithmus naturalis
<b><i>sin, cos, tan</i></b>	Sinus, Consinus, Tangens
<b><i>asin, acos, atan</i></b>	Arcussinus, Arcuscosinus, Arcustangens

Da die Methoden statisch definiert wurden, können wir alle entsprechend wie in folgendem Beispiel verwenden:

```
int x = 12, y = 31;
double eukdistanz = Math.sqrt(Math.pow(x,2)+Math.pow(y,2));
```

Dieses Beispiel berechnet den Euklidischen Abstand  $eukdistanz(x,y) = \sqrt{x^2 + y^2}$  zwischen  $x$  und  $y$ . In der Mathebibliothek gibt es auch eine Funktion zur Erzeugung von Zufallszahlen. Diese wird aber in Abschnitt 8.3.2 besprochen.

### 8.3 Zufallszahlen in Java

Es gibt verschiedene Möglichkeiten, Zufallszahlen zu verwenden. Oft benötigt man sie als Wahrscheinlichkeitsmaß im Intervall  $[0, 1]$ . In anderen Fällen ist es wünschenswert, aus einer Menge  $A$  mit  $n$  Elementen eines auszuwählen  $\{1, 2, \dots, n\}$ . Wir unterscheiden zunächst einmal den Datentyp der Zufallszahl. In jedem Fall verwenden wir die Klasse **Random** aus dem Package `java.util`.

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

Um eine der Klassenmethoden verwenden zu können, erzeugen wir eine Instanz der Klasse **Random**.

```
...
Random randomGenerator = new Random();
...
```

### 8.3.1 Ganzzahlige Zufallszahlen vom Typ int und long

Das kleine Lottoprogramm (6 aus 49) dient als Beispiel für die Erzeugung der Funktion `nextInt(n)`. Es werden Zufallszahlen aus dem Bereich  $[0, \dots, n - 1]$  gewählt. Wenn Zahlen aus dem Bereich `long` benötigt werden, so kann die Funktion `nextLong` analog verwendet werden.

```

1 import java.util.*;
2
3 public class Lotto {
4     public static void main(String[] args)
5     {
6         Random rg = new Random();
7         int[] zuf = new int[6];
8
9         System.out.print("Lottotipp (6 aus 49): ");
10        int wert, i=0;
11        aussen:
12        while(i<7){
13            wert = rg.nextInt(49);
14            // schon vorhanden?
15            for (int j=0; j<i; j++)
16                if (zuf[j]==wert)
17                    break aussen;
18            i++;
19            System.out.print(wert + " ");
20        }
21    }
22 }
```

### 8.3.2 Zufallszahlen vom Typ float und double

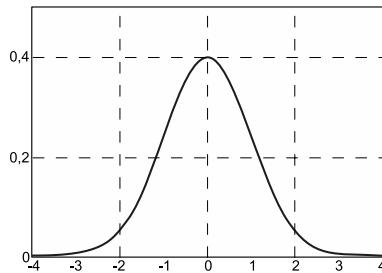
Für die Erzeugung einer Zufallszahl aus dem Intervall  $[0, 1]$  gibt es eine kürzere Schreibweise. In der Klasse `Math` im Package `java.lang` gibt es eine statische Funktion `random`, die eine Instanz der Klasse `Random` erzeugt, die Funktion `nextDouble` aufruft und den erzeugten Wert zurückliefert. Wir schreiben lediglich die folgende Zeile:

```
double zuffi = Math.random();
```

Die Funktionen `nextFloat` und `nextDouble` können aber auch analog zu dem Lotobispiel aus 8.3.1 verwendet werden.

### 8.3.3 Weitere nützliche Funktionen der Klasse Random

Es gibt für den Datentyp `boolean` die Funktion `nextBoolean`. Da in den vorherigen Abschnitten gleichverteilte Zufallszahlen erzeugt wurden, es aber manchmal gewünscht ist normalverteilte zu erhalten, gibt es die Funktion `nextGaussian`, die einen `double`-Wert liefert, der aus einer Gauß-Verteilung mit Mittelwert 0.0 und Standardabweichung 1.0 gewählt wird (Standardnormalverteilung).



Bei der Initialisierung der Klasse **Random** gibt es zwei Varianten. Die erste mit dem parameterlosen Konstruktor initialisiert sich in Abhängigkeit zur Systemzeit und erzeugt bei jedem Start neue Zufallszahlen. Für Programme, bei denen beispielsweise zeitkritische Abschnitte getestet werden, die aber abhängig von der jeweiligen Zufallszahl sind oder Experimente, bei denen die gleichen Stichproben verwendet werden sollen, ist der Konstruktor mit einem long als Parameter gedacht. Wir können beispielsweise einen long mit dem Wert 0 immer als Startwert nehmen und erhalten anschließend immer dieselben Zufallszahlen:

```
long initwert = 0;
Random randomGenerator = new Random(initwert);
```

## 8.4 Das Spiel Black Jack

Um gleich eine praktische Anwendung für die Zufallszahlen zu haben, werden wir das berühmte Kartenspiel Black Jack implementieren. Für unsere einfache Implementierung benötigen wir die Klassen **Spieler**, **Karte**, **Kartenspiel** und **BlackJack**. Bevor wir jedoch mit dem Programmieren beginnen, werden wir uns mit den Spielregeln vertraut machen.

### 8.4.1 Spielkarten

Gespielt wird mit einem 52er-Blatt, also 2, 3, ..., 10, *Bube*, *Dame*, *Koenig*, Ass für ♦, ♥, ♠, ♣.

### 8.4.2 Wertigkeiten der Karten

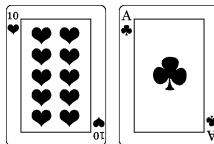
1. Asse zählen nach Belieben **ein** oder **elf Punkte**.
2. Zweier bis Zehner zählen entsprechend ihren Augen **zwei** bis **zehn Punkte**.
3. Bildkarten (Buben, Damen, Könige) zählen **zehn Punkte**.

### 8.4.3 Der Spielverlauf

Wir spielen mit vereinfachten Regeln. Vor Beginn eines Spiels platziert der Spieler seinen Einsatz. Dann werden zwei Karten an den Dealer und zwei an den Spieler vergeben. Die Wertigkeiten der jeweils zwei Karten werden zusammengezählt. Hier kann sich schon entschieden haben, wer gewonnen hat:

1. Hat der Dealer bereits 21 **Punkte** erreicht, verliert der Spieler automatisch.

**Beispiel (Summe der Wertigkeiten 21):**

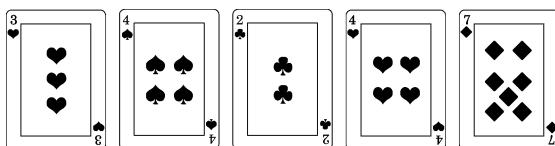


2. Trifft 1 nicht zu und hat der Spieler 21 **Punkte**, so gewinnt er mit Black Jack.

Nun beginnt der Spieler solange neue Karten anzufordern bis er meint, genug Karten zu haben. Die Summe der Wertigkeiten darf dabei 21 nicht überschreiten, sonst hat er verloren und die Einsätze gehen an die Bank. Hat der Spieler genug Karten, beginnt der Dealer seinerseits Karten anzufordern. Auch hier gilt, überschreitet die Summe der Wertigkeiten die 21, so hat der Dealer verloren und das Geld geht an den Spieler. Nimmt der Dealer jedoch keine Karte mehr auf, gelten folgende Entscheidungen:

1. Hat der Dealer fünf Karten in der Hand, verliert der Spieler.

**Beispiel (Summe der Wertigkeiten 20):**



2. Ist die Summe der Wertigkeiten beim Dealer gegenüber dem Spieler größer oder gleich, gewinnt der Dealer.
3. Ansonsten gewinnt der Spieler.

### 8.4.4 Spieler, Karten und Kartenspiel

Ein Spieler hat die Eigenschaften `spielerName`, `geld` und `karten`. Die in der Hand befindlichen Karten werden wir mit dem dynamischen Datentyp **Vector** realisieren. Er bietet zahlreiche Funktionen, wie `removeAllElements()`, `addElement(Element)`, `removeElement(Element)` und viele mehr.

#### 8.4.4.1 Verwendungsbeispiel für Datenstruktur Vector

Im Gegensatz zu einem **Array**, bei dem die Anzahl der Elemente bei der Initialisierung festgelegt wird, verhält sich der von Java angebotene Datentyp **Vector** dynamisch. Wenn wir also vor der Verwendung einer Liste die Anzahl der Elemente nicht kennen, können wir diesen Datentyp nehmen.

Ein kleines Beispiel dazu:

```

1 import java.util.Vector;
2
3 public class VectorTest{
4     public static void main(String[] args){
5         Vector v = new Vector();
6
7         // füge nacheinander Elemente in den Vector ein
8         for (int i=0; i<4; i++)
9             v.addElement(new Integer(i));
10
11        // Anzahl der Elemente im Vector v
12        System.out.println("Vector size = "+v.size());
13
14        // Auslesen der aktuellen Inhalts
15        for (int i=0; i<v.size(); i++){
16            Integer intObjekt = (Integer)v.elementAt(i);
17            int wert = intObjekt.intValue();
18
19            System.out.println("Element "+i+" = "+wert);
20        }
21        System.out.println();
22
23        // wir geben ein neues Element hinzu
24        v.insertElementAt(new Integer(9), 2);
25
26        // und löschen ein Element
27        v.removeElementAt(4);
28
29        // Auslesen der aktuellen Inhalts , kurze Schreibweise
30        for (int i=0; i<v.size(); i++)
31            System.out.println("Element "+i+" = "
32                            +((Integer)v.elementAt(i)).intValue());
33    }
34 }
```

Unser Beispiel liefert folgende Ausgabe:

```
C:\JavaCode>java VectorTest
Vector size = 4
Element 0 = 0
Element 1 = 1
Element 2 = 2
Element 3 = 3

Element 0 = 0
Element 1 = 1
Element 2 = 9
Element 3 = 2
```

#### 8.4.4.2 Implementierung der Klassen Spieler, Karten und Kartenspiel

Vom Spieler wollen wir noch erfahren können, welchen Wert seine „Hand“ besitzt. Schauen wir uns die Klasse **Spieler** einmal an:

```
1 import java.util.Vector;
2
3 public class Spieler {
4     private String spielerName;
5     private int geld;
6     private Vector karten;
7
8     // Konstruktor
9     public Spieler(String n, int g) {
10         spielerName = n;
11         geld = g;
12         karten = new Vector();
13     }
14
15     // Funktionen (get und set)
16     ...
17
18     public void clear() {
19         karten.removeAllElements();
20     }
21
22     public void addKarte(Karte k) {
23         karten.addElement(k);
24     }
25
26     public void remKarte(Karte k) {
27         karten.removeElement(k);
28     }
29
30     public int getAnzahlKarten() {
31         return karten.size();
32     }
33
34     // Spieler-Methoden
35     public Karte getKarte(int p) {
36         if ((p >= 0) && (p < karten.size()))
37             return (Karte)karten.elementAt(p);
38         else
39             return null;
40     }
41
42     public int aktuelleWertung() {
43         int wert, anzahl;
```

```

44 boolean istAss;
45
46     wert      = 0;
47     istAss    = false;
48     anzahl   = getAnzahlKarten();
49     Karte karte;
50     int kartenWert;
51
52     // wir durchlaufen unseren aktuellen Kartenstapel
53     for (int i=0; i<anzahl; i++) {
54         karte          = getKarte(i);
55         kartenWert    = karte.getWert();
56
57         // Bewertung der Bilder
58         if (kartenWert > 10) kartenWert = 10;
59         if (kartenWert == 1) istAss = true;
60
61         wert += kartenWert;
62     }
63
64     // Ass-Wert selber bestimmen
65     if (istAss && (wert + 10 <= 21))
66         wert = wert + 10;
67
68     return wert;
69 }
70 }
```

Dabei wurden vereinfachte Regeln verwendet. Es ist sinnvoll auch für eine Spielkarte eine Klasse zu entwerfen, das macht unser Programm transparent und wieder verwendbar. Eine Karte kann jede beliebige aus einem Pokerkartenspiel mit 52 Karten sein (2, 3, ..., 10, *Bube*, *Dame*, *Koenig*, *Ass* für *Karo*, *Herz*, *Pik* und *Kreuz*).

```

1 public class Karte {
2     // Bewertung der Karten und Definition der Farben
3     public final static int KARO=0, HERZ = 1, PIK    =2, KREUZ=3;
4     public final static int BUBE=11, DAME=12, KOENIG=13, ASS =1;
5     private final int farbe, wert;
6
7     // Konstruktor
8     public Karte(int f, int w) {
9         farbe = f;
10        wert = w;
11    }
12
13     // Funktionen (get und set)
14     ...
15
16     // Karten-Methoden
17     public String Farbe2String() {
18         switch (farbe) {
19             case KARO:
20                 return "Karo";
21             case HERZ:
22                 return "Herz";
23             case PIK:
24                 return "Pik";
25             case KREUZ:
26                 return "Kreuz";
27         }
28         System.out.println("Farbe falsch! : "+farbe);
29         return "-1";
30 }
```

```

30    }
31
32    public String Wert2String() {
33        if ((wert>=2)&&(wert<=10))
34            return ""+wert;
35
36        switch (wert) {
37            case 1:
38                return "A";
39            case 11:
40                return "B";
41            case 12:
42                return "D";
43            case 13:
44                return "K";
45        }
46        return "-1";
47    }
48
49    public String Karte2String() {
50        return Farbe2String() + "-" + Wert2String();
51    }
52}

```

Zusätzlich zu den get-set-Funktionen gibt es noch drei Ausgabemethoden, um die interne Repräsentation der Karten (0,...,12) in eine lesbare Form zu übersetzen (2,3,...,B,D,K,A).

Die Karten können wir zu der Klasse **Kartenspiel** zusammenfassen und Methoden zum Mischen und Herausgeben von Karten anbieten.

```

1 public class Kartenspiel {
2     // 52er Kartenstapel (2–10,B,D,K,A für Karo , Herz , Pik und Kreuz)
3     private Karte[] stapel;
4     private int kartenImSpiel;
5
6     // Konstruktor
7     public Kartenspiel() {
8         stapel = new Karte[52];
9         int zaehler = 0;
10        for (int f=0; f<4; f++ ) {
11            for (int w=1; w<14; w++ ) {
12                stapel[zaehler] = new Karte(f, w);
13                zaehler++;
14            }
15        }
16        mischen();
17    }
18
19    // Kartenspiel–Methoden
20    public void mischen() {
21        Karte temp;
22        for (int i=51; i>0; i--) {
23            int zuff      = (int(Math.random()*(i+1));
24            temp         = stapel[i];
25            stapel[i]    = stapel[zuff];
26            stapel[zuff] = temp;
27        }
28        kartenImSpiel = 52;
29    }
30
31    public int kartenAnzahl() {

```

```

32     return kartenImSpiel;
33 }
34
35 public Karte gibEineKarte() {
36     if (kartenImSpiel == 0)
37         mischen();
38     kartenImSpiel--;
39     return stapel[kartenImSpiel];
40 }
41 }
```

In der Methode `mischen` kommen unsere Zufallszahlen ins Spiel. Wir gehen alle Karten des Spiels durch, ermitteln jeweils eine zufällige Position im Stapel und tauschen die beiden Karten an diesen Positionen.

#### 8.4.5 Die Spielklasse BlackJack

Jetzt wollen wir zur eigentlichen Spielklasse **BlackJack** kommen. Diese Klasse verwaltet Spieler und Dealer, die jeweiligen Geldbeträge und den gesamten Spielablauf.

```

1 import java.io.*;
2
3 public class BlackJack{
4     private KartenSpiel kartenSpiel;
5     private Spieler spieler, dealer;
6     private int einsatz;
7     private boolean spiellaeuft;
8
9     // Konstruktor
10    public BlackJack(String n){
11        kartenSpiel = new KartenSpiel();
12        spieler = new Spieler(n, 500);
13        dealer = new Spieler("Dealer", 10000);
14        einsatz = 0;
15        spiellaeuft = false;
16    }
17
18    // Funktionen (get und set)
19    ...
20
21    public boolean getSpielStatus(){
22        return spiellaeuft;
23    }
24
25    // BlackJack-Methoden
26    public void neuesSpiel(){
27        spieler.clear();
28        dealer.clear();
29        spieler.addKarte(kartenSpiel.gibEineKarte());
30        dealer.addKarte(kartenSpiel.gibEineKarte());
31        spieler.addKarte(kartenSpiel.gibEineKarte());
32        dealer.addKarte(kartenSpiel.gibEineKarte());
33
34        spiellaeuft = true;
35    }
36
37    public void neueKarte(){
```

```

38     spieler.addKarte(kartenSpiel.gibEineKarte());
39 }
40
41 public void dealerIstDran(){
42     while ((dealer.aktuelleWertung()<=16)&&
43             (dealer.getAnzahlKarten()<5))
44         dealer.addKarte(kartenSpiel.gibEineKarte());
45 }
46
47 public boolean erhoeheEinsatz(){
48     if (dealer.getGeld()-50>=0){
49         dealer.setGeld(dealer.getGeld()-50);
50         einsatz+=50;
51     }
52     else {
53         System.out.println();
54         System.out.println("WOW! DU HAST DIE BANK PLEITE GEMACHT!");
55         System.out.println();
56         System.exit(1);
57     }
58     if (spieler.getGeld()-50>=0){
59         spieler.setGeld(spieler.getGeld()-50);
60         einsatz+=50;
61         return true;
62     }
63     return false;
64 }
```

Wir wollen noch drei statische Ausgabemethoden `hilfe`, `ausgabeKartenSpieler` und `kontoDaten`, die unabhängig von der Klasse **BlackJack** sind, definieren.

```

66 // ****
67 // statische Methoden
68 private static void hilfe(){
69     System.out.println();
70     System.out.println("Eingaben: ");
71     System.out.println("    n = eine neue Karte");
72     System.out.println("    d = fertig, Dealer ist dran");
73     System.out.println("    + = Einsatz um 50$ erhöhen");
74     System.out.println("    r = neue Runde");
75     System.out.println("    x = Spiel beenden");
76     System.out.println("    ? = Hilfe");
77     System.out.println();
78 }
79
80 private static void ausgabeKartenSpieler(Spieler s, Spieler d){
81     System.out.println();
82     System.out.print("Du erhältst: ");
83     for (int i=0; i<s.getAnzahlKarten(); i++) {
84         Karte karte = s.getKarte(i);
85         System.out.print(karte.Karte2String()+" ");
86     }
87     System.out.println("(Wertung="+s.aktuelleWertung()+" )");
88     System.out.print("Der Dealer erhält: ");
89     for (int i=0; i<d.getAnzahlKarten(); i++) {
90         Karte karte = d.getKarte(i);
91         System.out.print(karte.Karte2String()+" ");
92     }
93     System.out.println("(Wertung="+d.aktuelleWertung()+" )");
94     System.out.println();
95 }
96
97 private static void kontoDaten(Spieler s, Spieler d){
```

```

98     System.out.println();
99     System.out.println("$$$ "+s.getName()+" : "+s.getGeld()+", Bank: "
100        +d.getGeld()+" $$$");
101    System.out.println();
102 }

```

Für unser einfaches Black Jack-Spiel soll es genügen, den Spielbetrieb in die main-Methode einzufügen. Das macht die Methode zwar sehr lang, aber das Programm insgesamt relativ kurz. Nach einer Willkommenszeile wird die Eingabe des Spieler-names verlangt und schon kann das Spiel mit einem Einsatz begonnen werden.

```

104 public static void main(String[] args){
105     System.out.println("-----");
106     System.out.println("- WILLKOMMEN zu einem Spiel BlackJack! -");
107     System.out.println("-----");
108     hilfe();
109
110     InputStreamReader stdin = new InputStreamReader(System.in);
111     BufferedReader console = new BufferedReader(stdin);
112
113     System.out.print("Geben Sie Ihren Namen an: ");
114     String name = "";
115     try {
116         name = console.readLine();
117     } catch(IOException ioex){
118         System.out.println("Eingabefehler");
119         System.exit(1);
120     }
121
122     System.out.println();
123     System.out.println("Hallo "+name
124             +" , Dir stehen 500$ als Kapitel zur Verfuegung.");
125     System.out.println("Mach Deinen Einsatz(+) und
126                     beginne das Spiel(r).");
127     System.out.println();
128
129     // Nun starten wir eine Runde BlackJack
130     BlackJack blackjack = new BlackJack(name);
131
132     kontoDaten(blackjack.getSpieler(), blackjack.getDealer());

```

Der Spielbetrieb läuft in einer while-Schleife und kann durch die Eingabe 'x' beendet werden. Die zur Verfügung stehenden Eingaben sind: 'n' für eine neue Karte, 'd' für Spielerzug ist beendet und Dealer ist dran, '+' erhöht den Einsatz um 50\$, 'r' eine neue Runde wird gestartet, '?' zeigt die zur Verfügung stehenden Eingaben und wie bereits erwähnt, beendet 'x' das Spiel.

```

134 boolean istFertig = false;
135 String input="";
136 while (!istFertig){
137     try {
138         input = console.readLine();
139     } catch(IOException ioex){
140         System.out.println("Eingabefehler");
141     }
142     if (input.equals("n")){
143         // eine zusätzliche Karte bitte

```

```

144     if (blackjack.getSpieldStatus ()) {
145         blackjack.neueKarte();
146         if (blackjack.getSpieler().aktuelleWertung ()>21){
147             System.out.println("Du hast verloren! Deine Wertung
148                 liegt mit "+
149                 blackjack.getSpieler().aktuelleWertung ()+
150                 " ueber 21.");
151             blackjack.getDealer().setGeld(
152                 blackjack.getDealer().getGeld ()+
153                 blackjack.einsatz );
154             blackjack.einsatz = 0;
155             blackjack.spiellaeuft = false;
156             kontoDaten(blackjack.getSpieler(), 
157                         blackjack.getDealer ());
158         }
159         else {
160             ausgabeKartenSpieler(blackjack.getSpieler(),
161                                   blackjack.getDealer ());
162         }
163     }
164 }
```

Nachdem der Spieler genug Karten genommen hat, wird mit  $d$  der Zug abgeschlossen und der Dealer ist dran. Dabei wurde bereits geprüft, ob die Wertung der Hand des Spielers über 21 lag.

```

165     else if (input.equals("d")){
166         // Das Spiel wird an den Dealer uebergeben
167         if ((blackjack.getEinsatz ()>0)&&
168             (blackjack.getSpieldStatus ()) ) {
169             blackjack.dealerIstDran ();
170             if (blackjack.getDealer().aktuelleWertung ()>21){
171                 System.out.println("Du hast gewonnen! Der Dealer
172                     hat mit "+
173                     blackjack.getDealer().aktuelleWertung ()+
174                     " ueberboten.");
175                 blackjack.getSpieler().setGeld(
176                     blackjack.getSpieler().getGeld ()+
177                     blackjack.einsatz );
178             }
179             else if (blackjack.getDealer().getAnzahlKarten () == 5){
180                 System.out.println("Du hast verloren. Der Dealer
181                     erhielt 5 Karten unter 21.");
182                 blackjack.getDealer().setGeld(
183                     blackjack.getDealer().getGeld ()+
184                     blackjack.einsatz );
185             }
186             else if (blackjack.getDealer().aktuelleWertung ()
187                     > blackjack.getSpieler().aktuelleWertung ()) {
188                 System.out.println("Du hast verloren. "
189                     +blackjack.getDealer().aktuelleWertung ()+
190                     " zu "+
191                     blackjack.getSpieler().aktuelleWertung ()+
192                     ".");
193                 blackjack.getDealer().setGeld(
194                     blackjack.getDealer().getGeld ()+
195                     blackjack.einsatz );
196             }
197             else if (blackjack.getDealer().aktuelleWertung ()
198                     == blackjack.getSpieler().aktuelleWertung ()) {
199                 System.out.println("Du hast verloren. Der Dealer
200                     zog mit "+
201                     blackjack.getDealer().aktuelleWertung ()+
```

```

202             " Punkten gleich.");
203         blackjack.getDealer().setGeld(
204             blackjack.getDealer().getGeld()+
205             blackjack.einsatz);
206     }
207     else {
208         System.out.println("Du hast gewonnen! "
209             +blackjack.getSpieler().aktuelleWertung()+
210             " zu "+
211             blackjack.getDealer().aktuelleWertung()+
212             "!");
213         blackjack.getSpieler().setGeld(
214             blackjack.getSpieler().getGeld()+
215             blackjack.einsatz);
216     }
217     kontoDaten(blackjack.getSpieler(),blackjack.getDealer());
218     blackjack.einsatz = 0;
219     blackjack.spiellaeuft = false;
220 }
221 }
```

Beim Dealer entscheidet sich das Spiel. Er nimmt so viele Karte auf bis er gewonnen oder verloren hat. Dabei wird auch auf die Anzahl der Karten in seiner Hand geachtet, denn hat er fünf Karten mit dem Wert unter 21 auf der Hand, gewinnt er ebenfalls.

Fehlt noch die Erhöhung des Einsatzes vor einem Spiel:

```

222     else if (input.equals("+")){
223         // erhöhe den Einsatz
224         if (blackjack.getSpieldStatus())
225             System.out.println("Spiel laeuft bereits. Keine
226             Erhoehung moeglich");
227         else {
228             if (!(blackjack.erhoeheEinsatz()))
229                 System.out.println("Dein Geld reicht leider
230                     nicht mehr zum Erhöhen.");
231             else
232                 System.out.println("Einsatz wurde um 50$ auf "
233                     +blackjack.getEinsatz()+" erhöht.");
234         }
235     }
```

Um die Spielfunktionalität zu komplettieren, müssen noch die fehlenden drei Funktionen eingebaut werden.

```

236     else if (input.equals("r")){
237         // eine neue Runde wird gestartet, nachdem die Einsätze
238         // gemacht wurden
239         if ((blackjack.getEinsatz()>0)
240             &&(!blackjack.getSpieldStatus())){
241             blackjack.neuesSpiel();
242             kontoDaten(blackjack.getSpieler(),
243                         blackjack.getDealer());
244             ausgabeKartenSpieler(blackjack.getSpieler(),
245                         blackjack.getDealer());
246
247         if (blackjack.getDealer().aktuelleWertung() == 21) {
```

```

248     System.out.println("Schon verloren! Dealer hat 21
249             Punkte");
250     blackjack.einsatz      = 0;
251     blackjack.spiellaeuft  = false;
252     kontoDaten(blackjack.getSpieler(),
253                 blackjack.getDealer());
254 }
255 else if (blackjack.getSpieler().aktuelleWertung() == 21) {
256     System.out.println("Du hast mit BlackJack gewonnen!");
257     blackjack.einsatz      = 0;
258     blackjack.spiellaeuft  = false;
259     kontoDaten(blackjack.getSpieler(),
260                 blackjack.getDealer());
261 }
262 }
263 else if (input.equals("?")){
264     // Welche Programmeingaben sind möglich?
265     hilfe();
266 }
267 else if (input.equals("x")){
268     // Programm wird beendet
269     istFertig=true;
270     break;
271 }
272 }
273 }
274 // ****
275 }
276 }
```

Das Programm steht jetzt. Ein sehr erfolgreicher Testspielverlauf eines Spielers sah anschließend so aus:

```

C:\Java\BlackJack>java BlackJack
-
  WILLKOMMEN zu einem Spiel BlackJack !
-
Eingaben:
n = eine neue Karte
d = fertig , Dealer ist dran
+ = Einsatz um 50$ erhöhen
r = neue Runde
x = Spiel beenden
? = Hilfe

Geben Sie Ihren Namen an: Daniel

Hallo Daniel, Dir stehen 500$ als Kapitel zur Verfüigung.
Mach Deinen Einsatz(+) und beginne das Spiel(r).

$$$ Daniel: 500, Bank: 10000 $$$

+
Einsatz wurde um 50$ auf 100 erhöht.
r

$$$ Daniel: 450, Bank: 9950 $$$

Du erhältst: Kreuz-B Pik-A (Wertung=21)
Der Dealer erhält: Kreuz-8 Kreuz-D (Wertung=18)

Du hast mit BlackJack gewonnen!
```

```

$$$ Daniel: 450, Bank: 9950 $$

+
Einsatz wurde um 50$ auf 100 erhöht.
+
Einsatz wurde um 50$ auf 200 erhöht.
+
Einsatz wurde um 50$ auf 300 erhöht.
+
Einsatz wurde um 50$ auf 400 erhöht.
+
Einsatz wurde um 50$ auf 500 erhöht.
r

$$$ Daniel: 200, Bank: 9700 $$

Du erhaelst: Herz-2 Herz-B (Wertung=12)
Der Dealer erhaelt: Karo-8 Pik-D (Wertung=18)

n

Du erhaelst: Herz-2 Herz-B Pik-9 (Wertung=21)
Der Dealer erhaelt: Karo-8 Pik-D (Wertung=18)

d
Du hast gewonnen! 21 zu 18!

$$$ Daniel: 700, Bank: 9700 $$

```

Die Ausgaben auf der Konsole sind für unsere aktuellen Bedürfnisse ausreichend. Später (in Kapitel 9) werden wir uns mit graphischen Oberflächen beschäftigen und sind je nach Kreativität und Talent dann in der Lage, schickere Programme zu entwickeln.

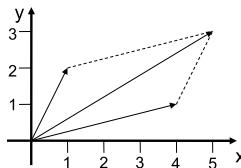
## 8.5 JAMA - Lineare Algebra

JAMA [39] ist die meist genutzte Bibliothek für Methoden der Linearen Algebra. Es lassen sich beispielsweise Matrizenoperationen ausführen, Determinante, Rang und Eigenwerte einer Matrix berechnen und Lineare Gleichungssysteme lösen. Nach der Installation genügt die Importierung der Bibliothek mit folgender Zeile:

```
import Jama.*;
```

Schauen wir uns ein paar einfache Beispiele an, die die Verwendung des JAMA-Pakets aufzeigen.

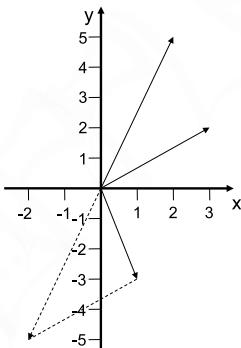
In unserem ersten Beispiel addieren wir zwei Vektoren `vektor1 = (1, 2)` und `vektor2 = (4, 1)`. Da JAMA keinen Vektordatentyp anbietet, verwenden wir eindimensionale Matrizen.



```
double [][] vektor1 = {{1},{2}};
double [][] vektor2 = {{4},{1}};
Matrix v1 = new Matrix(vektor1);
Matrix v2 = new Matrix(vektor2);
Matrix x = v1.plus(v2);
```

Die Vektoren `vektor1` und `vektor2` werden initialisiert und durch den **Matrix**-Konstruktor in eine Matrixform gebracht. Die **Matrix**-Klasse stellt eine Reihe von Funktionen zur Verfügung, unter anderem die Addition der Matrix zu einer weiteren. Der Ergebnisvektor  $x=(5, 3)$  liefert, wie erwartet, das richtige Ergebnis.

Im zweiten Beispiel verfahren wir analog zum ersten und subtrahieren diesmal Vektor `vektor2=(2, 5)` von dem Vektor `vektor1=(3, 2)` ab .



```
double [][] vektor1 = {{3},{2}};
double [][] vektor2 = {{2},{5}};
Matrix v1 = new Matrix(vektor1);
Matrix v2 = new Matrix(vektor2);
Matrix x = v1.minus(v2);
```

Als Ergebnis erhalten wir  $x=(1, -3)$ .

Die Determinante einer Matrix lässt sich ebenfalls mit wenig Aufwand berechnen:

```
double [][] array = {{-2,1},{0,4}};
Matrix a = new Matrix(array);
double d = a.det();
```

Die Variable  $d$  sollte den Wert  $-8$  haben, denn  $\det\begin{pmatrix} -2 & 1 \\ 0 & 4 \end{pmatrix} = (-2) \cdot 4 - 1 \cdot 0 = -8$ .

Zu guter Letzt schauen wir uns noch ein Beispiel für die Lösung eines Linearen Gleichungssystems  $A \vec{x} = B$  an. Die Matrix  $a$  nehmen wir aus dem vorhergehenden Beispiel und  $b$  erzeugen wir zufällig.

```
1 double [][] array = {{-2,1},{0,4}};
2 Matrix a = new Matrix(array);
3 Matrix b = Matrix.random(2,1);
4 Matrix x = a.solve(b);
```

## 8.6 Eine eigene Bibliothek bauen

Die Erzeugung eines eigenen Packages unter Java ist sehr einfach. Wichtig ist das Zusammenspiel aus Klassennamen und Verzeichnisstruktur. Angenommen wir wollen eine Klasse **MeinMax** in einem Package **meinMathe** anbieten. Dann legen wir ein Verzeichnis **meinMathe** an und speichern dort z. B. die folgende Klasse:

```
1 package meinMathe;
2
3 public class MeinMax{
4     public static int maxi(int a, int b){
5         if (a<b) return b;
6         return a;
7     }
8 }
```

Durch das Schlüsselwort `package` haben wir signalisiert, dass es sich um eine Klasse des Package **meinMathe** handelt. Unsere kleine Mathekklasse bietet eine bescheidene `maxi`-Funktion.

Nachdem wir diese Klasse mit `javac` kompiliert haben, können wir außerhalb des Ordners eine neue Klasse schreiben, die dieses Package jetzt verwendet. Dabei ist darauf zu achten, dass der Ordner des neuen Packages entweder im gleichen Ordner wie die Klasse liegt, die das Package verwendet, oder dieser Ordner im *PATH* aufgelistet ist.

Hier unsere Testklasse:

```
1 import MeinMathe.MeinMax;
2
3 public class MatheTester{
4     public static void main(String[] args){
5         System.out.println("Ergebnis = "+MeinMax.maxi(3,9));
6     }
7 }
```

Wir erhalten nach der Ausführung folgende Ausgabe:

```
C:\JavaCode>java MatheTester
Ergebnis = 9
```

## 8.7 Zusammenfassung und Aufgaben

### Zusammenfassung

Die Zusammenfassung von Klassen und Methoden zu Bibliotheken ist ein wichtiges Konzept der Softwareentwicklung. Zu Beginn haben wir einen Blick in die bei Java mitgelieferten Standardbibliotheken geworfen. Als Beispiel haben wir uns mit der Klasse **Math** aus dem Package `java.util` beschäftigt und die Arbeit mit Zufallszahlen kennengelernt. Wir konnten die Zufallszahlen gleich in dem Projekt BlackJack einsetzen.

Nicht nur der Einsatz von den mitgelieferten Standardbibliotheken wurde geübt, sondern auch die Einbindung einer externen Bibliothek für Lineare Algebra. Anschließend haben wir noch gesehen, wie eine eigene Bibliothek erstellt werden kann.

### Aufgaben

Übung 1) Erweitern Sie das BlackJack-Programm auf den kompletten Regelsatz, wie er auf der Wikipediaseite [45] erläutert wird (Beispielsweise 7er-Drilling oder Verteilung des Potts).

Übung 2) Entwickeln Sie ein Pokerspiel. Sie können die einfache Variante mit jeweils fünf Karten pro Spieler und anschließendem Tausch oder komplexere Varianten, wie Texas Hold'em wählen. Beschäftigen Sie sich dabei intensiv mit den Wahrscheinlichkeitsverteilungen und verwenden Sie den Java-Zufallsgenerator.

Übung 3) Schreiben Sie ein Programm, das zufällig eine Zahl vom Typ `float` aus folgenden Intervallen erzeugt:

- (i)  $[1, 2]$
- (ii)  $[2, 4)$
- (iii)  $(0, 1)$
- (iv)  $[-10, 5]$

Übung 4) Schreiben Sie ein Programm, das zufällig eine Zahl vom Typ `int` aus folgenden Intervallen erzeugt:

- (i)  $[-20, 10000]$
- (ii)  $(1, 49)$
- (iii)  $(-1, 1]$

Übung 5) Schreiben Sie ein Programm Vokabelmanager, das folgende Eigenschaften besitzt:

- o Eingabe von Variablen (z. B. Englisch-Deutsch)
- o Bearbeitung der Vokabeldatenbank
- o Vokabeltrainer, der zufällig Vokabeln prüft. Richtig erkannte Variablen werden im aktuellen Test nicht mehr geprüft, falsche wiederholen sich.
- o **Zusatz:** Überlegen Sie sich ein geeignetes Konzept für die Verwaltung mehrerer Bedeutungen eines Wortes. Denken Sie daran, dass auch der Vokabeltrainer später mehrere Bedeutungen überprüfen muss. Ermöglichen Sie beide Teströhungen z. B. Englisch-Deutsch und Deutsch-Englisch.

Übung 6) Realisieren Sie mit Hilfe eines Vectors die abstrakten Datenstrukturen Warteschlange und Stack.

## Tag 9: Grafische Benutzeroberflächen

In Java gibt es verschiedene Möglichkeiten, Klassen für die Grafikausgabe zu verwenden. Wir werden in den folgenden Abschnitten **AWT** (=Abstract Windows Toolkit) kennenlernen. Beginnen werden wir mit der Erzeugung von Fenstern und sehen, wie Text oder einfache Fensterelemente angezeigt werden können. Später kommt die Behandlung von Fenster- und Mausereignissen dazu.

### 9.1 Fenstermanagement unter AWT

Zunächst machen wir kleine Gehversuche und eignen uns Schritt für Schritt den Umgang mit den Fenstern unter Java an. Fenster sind im package `java.awt` eine eigene Klasse. Diese Klasse heißt **Frame** und besitzt viele Funktionen und Eigenschaften.

#### 9.1.1 Ein Fenster erzeugen

Wir können schon mit wenigen Zeilen ein Fenster anzeigen, indem wir eine Instanz der Klasse **Frame** erzeugen und sie sichtbar machen. Bevor wir die Eigenschaft „ist sichtbar“ mit `setVisible(true)` setzen, legen wir mit der Funktion `setSize(breite, hoehe)` die Fenstergröße fest.

```
1 import java.awt.Frame;
2 public class MeinErstesFenster {
3     public static void main(String[] args) {
4         // öffnet ein AWT-Fenster
5         Frame f = new Frame("So einfach geht das?");
6         f.setSize(300, 200);
7         f.setVisible(true);
8     }
9 }
```

Nach dem Start öffnet sich folgendes Fenster:



Nach der Erzeugung des Fensters ist die Ausgabeposition die linke obere Ecke des Bildschirms.

Das Fenster lässt sich momentan nicht ohne Weiteres schließen. Wie wir diese Sache in den Griff bekommen und das Programm nicht jedes Mal mit Tastenkombination **STRG+C** (innerhalb der Konsole) beenden müssen, sehen wir in Abschnitt 9.3. Da das dort vorgestellte Konzept doch etwas mehr Zeit in Anspruch nimmt, experimentieren wir mit den neuen Fenstern noch ein wenig herum.

### 9.1.2 Das Fenster zentrieren

Um das Fenster zu zentrieren, lesen wir die Bildschirmgröße ein, errechnen die Mittelpunkte und setzen die Koordinaten des Fensters mit Hilfe dieser Mittelpunkte neu. Im Unterschied zum vorhergehenden Beispiel erben wir nun alle Fenstereigenschaften der Klasse **Frame** und geben unserer neuen Klasse den Namen **FensterPositionieren**, denn mehr können wir erst einmal noch nicht tun. Dazu werden die dem Konstruktor übergebenen Parameter **x** und **y**, die die Breite und Höhe des Fensters darstellen sollen, wie im vorherigen Beispiel gesetzt.

Wir sehen in diesem Beispiel weiter, wie wir in der sechsten Zeile mit der Funktion **getScreenSize** der Klasse **Toolkit**, die Bildschirmauflösung auslesen können. Daraus können wir den Mittelpunkt für die **x**- und die **y**-Achse ableiten:

$$x = \frac{(\text{Bildschirmbreite} - \text{Fensterbreite})}{2} \text{ und } y = \frac{(\text{Bildschirmhoehe} - \text{Fensterhoehe})}{2} .$$

Mit der Funktion **setLocation** legen wir die Position des Fensters neu fest und machen es anschließend sichtbar.

```

1 import java.awt.*;
2 public class FensterPositionieren extends Frame {
3     public FensterPositionieren(int x, int y){
4         setTitle("Ab in die Mitte!");
5         setSize(x, y);

```

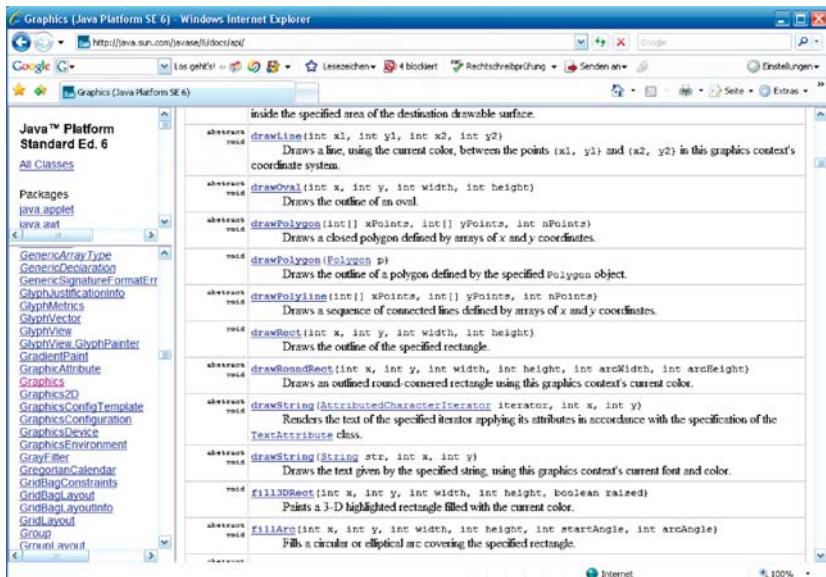
```

6     Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
7     setLocation((d.width-getSize().width)/2,
8                 (d.height-getSize().height)/2);
9     setVisible(true);
10 }
11
12 public static void main(String[] args) {
13     FensterPositionieren f = new FensterPositionieren(200, 100);
14 }
15 }
```

Das Fenster wird jetzt mittig auf dem Monitor platziert.

## 9.2 Zeichenfunktionen innerhalb des Fensters verwenden

AWT bietet eine Reihe von Zeichenfunktionen an. Alle mit ihren Feinheiten zu beschreiben würde wieder ein ganzes Buch füllen. Wir wollen an dieser Stelle nur ein paar grundlegende Beispiele erläutern und den Leser motivieren, spätestens hier die Java API [57] als Hilfsmittel zu verwenden. Um etwas in einem Fenster anzeigen zu können, müssen wir die Funktion `paint` der Klasse **Frame** überschreiben. Als Parameter sehen wir den Typ **Graphics**. Die Klasse **Graphics** beinhaltet unter anderem alle Zeichenfunktionen. Schauen wir dazu mal in die API.



In der linken Auswahl wurde die Klasse **Graphics** ausgewählt und rechts sind die Funktionen sichtbar. Mit der Funktion `drawString` wollen wir beginnen und einen Text ausgeben.

### 9.2.1 Textausgaben

Ein Text wird innerhalb des Fensterbereichs ausgegeben. Zusätzlich zeigt dieses Beispiel, wie sich die Vorder- und Hintergrundfarben über die Funktionen `setBackground` und `setForeground` manipulieren lassen. Die Klasse `Color` bietet bereits vordefinierte Farben und es lassen sich neue Farben, bestehend aus den drei Farbkomponenten *rot*, *blau* und *grün*, erzeugen.

```

1 import java.awt.*;
2 public class TextFenster extends Frame {
3     public TextFenster(String titel) {
4         setTitle(titel);
5         setSize(500, 100);
6         setBackground(Color.lightGray);
7         setForeground(Color.blue);
8         setVisible(true);
9     }
10
11    public void paint(Graphics g) {
12        g.drawString("Hier steht ein kreativer Text.", 120, 60);
13    }
14
15    public static void main(String[] args) {
16        TextFenster t = new TextFenster("Text im Fenster");
17    }
18 }
```

Der Hintergrund ist grau und die Schrift blau.



### 9.2.2 Zeichenelemente

Exemplarisch zeigt dieses Beispiel die Verwendung der Zeichenfunktionen `drawRect` und `drawLine`. Auch hier haben wir eine zusätzliche Funktionalität eingebaut, die Wartefunktion.

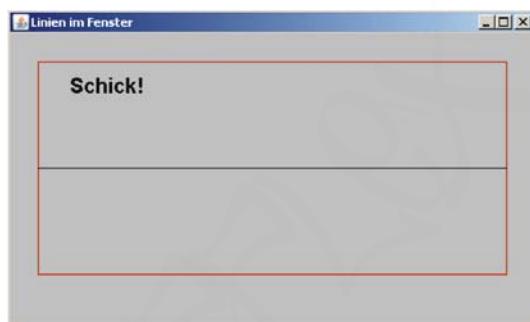
```

1 import java.awt.*;
2 public class ZeichenElemente extends Frame {
3     public ZeichenElemente(String titel) {
4         setTitle(titel);
5         setSize(500, 300);
6         setBackground(Color.lightGray);
7         setForeground(Color.red);
8         setVisible(true);
9     }
10
11    public static void wartemal(long millis){
12        try {
13            Thread.sleep(millis);
14        }
```

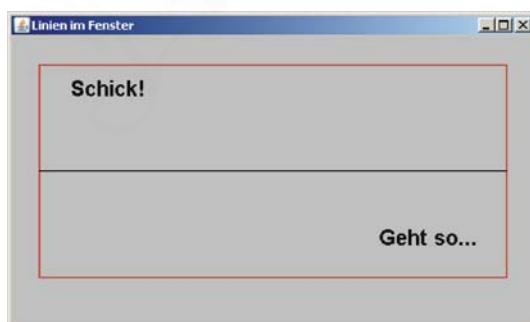
```

14     } catch (InterruptedException e){}
15 }
16
17 public void paint(Graphics g) {
18     g.drawRect(30,50,440,200);
19     g.setColor(Color.black);
20     g.drawLine(30,150,470,150);
21     g.setFont(new Font("SansSerif", Font.BOLD, 20));
22     g.drawString("Schick!", 60, 80);
23     wartemal(3000);
24     g.drawString("Geht so...", 350, 220);
25 }
26
27 public static void main(String[] args) {
28     ZeichenElemente t = new ZeichenElemente("Linien im Fenster");
29 }
30 }
```

Unser Programmcode liefert nach der Ausführung folgende Ausgabe...



... und nach ca. 3 Sekunden ändert sich die Ausgabe wie folgt ...



### 9.2.3 Die Klasse Color verwenden

Für grafische Benutzeroberflächen sind Farben sehr wichtig. Um das **RGB-Farbmodell** zu verwenden, erzeugen wir viele farbige Rechtecke, deren drei Farbkomponenten *rot*, *grün* und *blau* zufällig gesetzt werden. Dazu erzeugen wir ein **Color**-Objekt und setzen dessen Farbwerte.

```

1 import java.awt.*;
2 import java.util.Random;
3
4 public class FarbBeispiel extends Frame {
5     public FarbBeispiel(String titel) {
6         setTitle(titel);
7         setSize(500, 300);
8         setBackground(Color.lightGray);
9         setForeground(Color.red);
10        setVisible(true);
11    }
12
13    public void paint(Graphics g){
14        Random r = new Random();
15        for (int y=30; y<getHeight()-10; y += 15)
16            for (int x=12; x<getWidth()-10; x += 15) {
17                g.setColor(new Color(r.nextInt(256),
18                    r.nextInt(256),
19                    r.nextInt(256)));
20                g.fillRect(x, y, 10, 10);
21                g.setColor(Color.BLACK);
22                g.drawRect(x - 1, y - 1, 10, 10);
23            }
24    }
25
26    public static void main(String[] args) {
27        FarbBeispiel t = new FarbBeispiel("Viel Farbe im Fenster");
28    }
29 }
```

Wir erhalten kleine farbige Rechtecke und freuen uns auf mehr.



#### 9.2.4 Bilder laden und anzeigen

Mit der Methode `drawImage` können wir Bilder anzeigen lassen. In den Zeilen 14 bis 16 geschieht aber leider nicht das, was wir erwarten würden. Die Funktion `getImage` bereitet das Laden des Bildes nur vor. Der eigentliche Ladevorgang erfolgt erst beim Aufruf von `drawImage`. Das hat den Nachteil, dass bei einer Wiederverwendung der Methode `paint` jedes Mal das Bild neu geladen wird.

```

1 import java.awt.*;
2 import java.util.Random;
3
4 public class BildFenster extends Frame {
5     public BildFenster(String titel) {
6         setTitle(titel);
7         setSize(423, 317);
8         setBackground(Color.lightGray);
9         setForeground(Color.red);
10        setVisible(true);
11    }
12
13    public void paint(Graphics g){
14        Image pic = Toolkit.getDefaultToolkit().getImage(
15                    "C:\\kreidefelsen.jpg");
16        g.drawImage(pic, 0, 0, this);
17    }
18
19    public static void main(String[] args) {
20        BildFenster t = new BildFenster("Bild im Fenster");
21    }
22}

```

Jetzt könnten wir sogar schon eine Diashow der letzten Urlaubsbilder realisieren.



Um zu verhindern, dass das Bild neu geladen werden soll, können mit Hilfe der Klasse **Mediatracker** die Bilder vor der eigentlichen Anzeige in den Speicher geladen werden.

Eine globale Variable `img` vom Typ **Image** wird angelegt und im Konstruktor mit dem **Mediatracker** verknüpft. Der **Mediatracker** liest die entsprechenden Bilder ein und speichert sie:

```

1 // wird dem Konstruktor hinzugefügt
2 img = getToolkit().getImage("c:\\kreidefelsen.jpg");
3 Mediatracker mt = new Mediatracker(this);
4 mt.addImage(img, 0);
5 try {
6     mt.waitForAll();
7 } catch (InterruptedException e){}

```

Jetzt können wir in der `paint`-Methode mit dem Aufruf

```
1 g.drawImage(img, 0, 0, this);
```

das Bild aus dem **Mediatracker** laden und anzeigen.

## 9.3 Auf Fensterereignisse reagieren und sie behandeln

Als Standardfensterklasse werden wir in den folgenden Abschnitten immer von dieser erben:

```
1 import java.awt.*;
2
3 public class MeinFenster extends Frame {
4     public MeinFenster(String titel, int w, int h){
5         this.setTitle(titel);
6         this.setSize(w, h);
7
8         // zentriere das Fenster
9         Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
10        this.setLocation((d.width-this.getSize().width)/2,
11                         (d.height-this.getSize().height)/2);
12    }
13 }
```

Die Klasse **MeinFenster** erzeugt ein auf dem Bildschirm zentriertes Fenster und kann mit einem Konstruktor und den Attributen titel, breite und hoehe erzeugt werden.

### 9.3.1 Fenster mit dem Interface WindowListener schließen

Unser folgendes Beispiel erbt zunächst von der Klasse **MeinFenster** und implementiert anschließend das Interface **WindowListener**. In Zeile 4 verknüpfen wir unsere Anwendung mit dem Interface **WindowListener** und erreichen damit, dass bei Ereignissen, wie z. B. „schließe Fenster“, die entsprechenden implementierten Methoden aufgerufen werden.

```
1 import java.awt.*;
2 import java.awt.event.*;
3 public class FensterSchliesst extends MeinFenster
4     implements WindowListener {
5     public FensterSchliesst(String titel, int w, int h){
6         super(titel, w, h);
7         addWindowListener(this); // wir registrieren hier den
8                             // Ereignistyp für WindowEvents
9         setVisible(true);
10    }
11 }
```

```

12 // ****
13 // Hier werden die WindowListener–Methoden implementiert
14 // Methode: Fenster wird geschlossen
15 public void windowClosing(WindowEvent event) {
16     System.exit(0);
17 }
18 public void windowClosed(WindowEvent event) {}
19 public void windowDeiconified(WindowEvent event) {}
20 public void windowIconified(WindowEvent event) {}
21 public void windowActivated(WindowEvent event) {}
22 public void windowDeactivated(WindowEvent event) {}
23 public void windowOpened(WindowEvent event) {}
24 // ****
25
26 public static void main(String[] args) {
27     FensterSchliesst f = new FensterSchliesst("Schliesse mich!", 
28                                         200, 100);
29 }
30 }
```

In Zeile 16 verwenden wir die Funktion `System.exit(0)`. Es werden alle Fenster der Anwendung geschlossen und das Programm beendet. Leider haben wir mit der Implementierung des Interfaces **WindowListener** den Nachteil, dass wir alle Methoden implementieren müssen. Das Programm wird schnell unübersichtlich, wenn wir verschiedene **Eventtypen** abfangen wollen und für jedes Interface alle Methoden implementieren müssen.

Hilfe verspricht die Klasse **WindowAdapter**, die das Interface **WindowListener** bereits mit leeren Funktionskörpern implementiert hat. Wir können einfach von dieser Klasse erben und eine der Methoden überschreiben. Um die restlichen brauchen wir uns nicht zu kümmern.

```

1 import java.awt.*;
2 import java.awt.event.*;
3 public class FensterSchliesstSchick extends MeinFenster{
4     public FensterSchliesstSchick(String titel, int w, int h){
5         super(titel, w, h);
6         // Wir verwenden eine Klasse, die nur die gewünschten Methoden
7         // der Klasse WindowAdapter überschreibt.
8         addWindowListener(new WindowClosingAdapter());
9         setVisible(true);
10    }
11
12    public static void main(String[] args) {
13        FensterSchliesstSchick f =
14            new FensterSchliesstSchick("Schliesse mich!", 200, 100);
15    }
16 }
17
18 /* Etwas nervig war es, alle Methoden, die das Interface WindowListener
19 angeboten hat, überschreiben zu müssen. Angenehmer ist es, die
20 WindowAdapter–Klasse zu verwenden. Sie hat alle Funktionen bereits
21 mit leerem Programmkörper implementiert. Wollen wir nur eine Funktion
22 verwenden, so können wir in einer Subklasse von WindowAdapter die
23 gewünschte Methode überschreiben. In diesem Fall, windowClosing.
24 */
25 class WindowClosingAdapter extends WindowAdapter {
26     public void windowClosing(WindowEvent e) {
27         System.exit(0);
28     }
29 }
```

Wir können auch die Klasse **WindowClosingAdapter** als **innere Klasse** deklarieren.

```

1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class FensterSchliesstSchick2 extends MeinFenster{
5     public FensterSchliesstSchick2(String titel, int w, int h){
6         super(titel, w, h);
7         addWindowListener(new WindowClosingAdapter());
8         setVisible(true);
9     }
10
11 // ****
12 // innere Klasse
13 private class WindowClosingAdapter extends WindowAdapter {
14     public void windowClosing(WindowEvent e) {
15         System.exit(0);
16     }
17 }
18 // ****
19
20 public static void main(String[] args) {
21     FensterSchliesstSchick2 f =
22         new FensterSchliesstSchick2("Schliesse mich!", 200, 100);
23 }
24 }
```

Eine noch kürzere Schreibweise könnten wir erreichen, indem wir die Klasse **WindowAdapter** nur lokal erzeugen und die Funktion überschreiben [37]. Wir nennen solche Klassen **innere, anonyme Klassen**.

```

1 import java.awt.event.*;
2 public class FensterSchliesstSchickKurz extends MeinFenster{
3     public FensterSchliesstSchickKurz(String titel, int w, int h){
4         super(titel, w, h);
5         // Wir verwenden eine innere anonyme Klasse. Kurz und knapp.
6         addWindowListener(new WindowAdapter() {
7             public void windowClosing(WindowEvent e) {
8                 System.exit(0);
9             }
10        });
11        setVisible(true);
12    }
13
14 public static void main(String[] args) {
15     FensterSchliesstSchickKurz f =
16         new FensterSchliesstSchickKurz("Schliesse mich!", 200, 100);
17 }
18 }
```

Dieses Verfahren werden wir ab sofort für kleine Funktionen verwenden. Sollten die Funktionen zu groß werden, greifen wir auf die **lokale Klassen** zurück.

### 9.3.2 GUI-Elemente und ihre Ereignisse

Wir werden uns jetzt exemplarisch Beispiele anschauen, bei denen GUI-Elemente erzeugt werden, die auf Maus-Ereignisse reagieren. Bevor wir damit beginnen, müssen

wir uns mit dem Problem auseinander setzen, wie die Elemente angeordnet werden sollen, wenn der Anwender beispielsweise die Fenstergröße ändert. Da wir an dieser Stelle keinen großen Verwaltungsaufwand betreiben möchten, verwenden wir einen Layout Manager.

### 9.3.2.1 Layout Manager

Java bietet viele vordefinierte Layoutmanager an [7, 1, 10]. Der einfachste Layoutmanager **FlowLayout** fügt die Elemente, je nach Größe, ähnlich einem Textfluss, links oben beginnend, in das Fenster ein.

Diese lassen sich z. B. innerhalb eines Frames mit der folgenden Zeile aktivieren:

```
setLayout(new FlowLayout());
```

Dem Leser sei hier empfohlen, erst nach diesem Kapitel, die entsprechenden Quellen nachzulesen<sup>1</sup>.

### 9.3.2.2 Die Komponenten Label und Button

Zwei Button und ein Label werden in die GUI eingebettet. Die Anordnung der Elemente innerhalb des Fensters kann von einem Layoutmanager übernommen werden. Für dieses Beispiel haben wir den Layoutmanager **FlowLayout** verwendet.

```
1 import java.awt.*;
2 import java.awt.event.*;
3 public class GUI_Button extends MeinFenster{
4     Button button1, button2;
5     Label labell;
6
7     // Im Konstruktor erzeugen wir die GUI-Elemente
8     public GUI_Button(String titel, int w, int h){
9         super(titel, w, h);
10        setSize(w, h);
11
12        // Wir registrieren den WindowListener, um auf
13        // WindowEvents reagieren zu können
14        addWindowListener(new MeinWindowListener());
15
16        // wir bauen einen ActionListener, der nur auf Knopfdruck
17        // reagiert
18        ActionListener aktion = new Knopfdruck();
19
20        setLayout(new FlowLayout());
21        button1 = new Button("Linker Knopf");
22        add(button1);
23        button1.addActionListener(aktion);
```

---

<sup>1</sup> In diesem Buch wurde auf eine umfangreiche Darstellung der Layoutmanager verzichtet, da sie nicht viel zum Verständnis der Java-Programmierung beitragen und ständig erweitert werden.

```

24     button1.setActionCommand("b1");
25     button2 = new Button("Rechter Knopf");
26     add(button2);
27     button2.addActionListener(aktion);
28     button2.setActionCommand("b2");
29     label1 = new Label("Ein Label");
30     add(label1);
31     setVisible(true);
32 }
33
34 // ****
35 // Innere Klassen für das Eventmanagement
36 class MeinWindowListener extends WindowAdapter{
37     public void windowClosing(WindowEvent event){
38         System.exit(0);
39     }
40 }
41
42 class Knopfdruck implements ActionListener{
43     public void actionPerformed(ActionEvent e){
44         label1.setText(e.getActionCommand());
45     }
46 }
47 // ****
48
49 public static void main(String[] args) {
50     GUI_Button f = new GUI_Button("Schliesse mich!", 500, 500);
51 }
52 }
```

Um auf einen Knopfdruck reagieren zu können, erzeugen wir eine Instanz der Klasse **Knopfdruck**, die das Interface **ActionListener** mit der einzigen Methode `actionPerformed` implementiert. Diese Instanz verknüpfen wir mit beiden Buttons. Die Methode `actionPerformed` wird jetzt immer dann aufgerufen, wenn einer der beiden Button gedrückt wurde.



Oder wir reagieren individuell:

```

class Knopfdruck implements ActionListener{
    public void actionPerformed(ActionEvent e){
        // wir behandeln die Ereignisse
        String cmd = e.getActionCommand();
        if (cmd.equals("b1"))
            Button1Clicked();
        if (cmd.equals("b2"))
            Button2Clicked();
    }
}
```

### 9.3.2.3 Die Komponente TextField

Wie wir Eingaben über ein TextField erhalten, zeigt das folgende Beispiel:

```

1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class GUI_Button_TextField extends MeinFenster{
5     Button button1;
6     Label labell;
7     TextField textfield1;
8
9     // Im Konstruktor erzeugen wir die GUI-Elemente
10    public GUI_Button_TextField(String titel , int w, int h){
11        super(titel , w, h);
12        setSize(w, h);
13
14        // Wir registrieren den WindowListener , um auf
15        // WindowEvents reagieren zu können
16        addWindowListener(new MeinWindowListener());
17
18        // wir bauen einen ActionListener , der nur auf Knopfdruck
19        // reagiert
20        ActionListener aktion = new Knopfdruck();
21
22        setLayout(new FlowLayout());
23
24        textfield1 = new TextField("hier steht schon was", 25);
25        add(textfield1);
26
27        button1 = new Button("Knopf");
28        add(button1);
29        button1.addActionListener(aktion);
30        button1.setActionCommand("b1");
31
32        labell = new Label("noch steht hier nicht viel");
33        add(labell);
34
35        setVisible(true);
36    }
37
38    private void Button1Clicked(){
39        String txt = textfield1.getText();
40        labell.setText(txt);
41    }
42
43    // ****
44    // Innere Klassen für das Eventmanagement
45    class MeinWindowListener extends WindowAdapter{
46        public void windowClosing(WindowEvent event){
47            System.exit(0);
48        }
49    }
50
51    class Knopfdruck implements ActionListener{
52        public void actionPerformed (ActionEvent e){
53            // wir behandeln die Ereignisse
54            String cmd = e.getActionCommand();
55            if (cmd.equals("b1")){
56                Button1Clicked();
57            }
58        }
59    // ****
60
61    public static void main( String[] args ) {
62        GUI_Button_TextField f =
63            new GUI_Button_TextField("Klick mich...", 500, 500);
64    }
65}

```

Sollte der Button gedrückt werden, so wird der Textinhalt von `textfield1` in `label1` geschrieben.



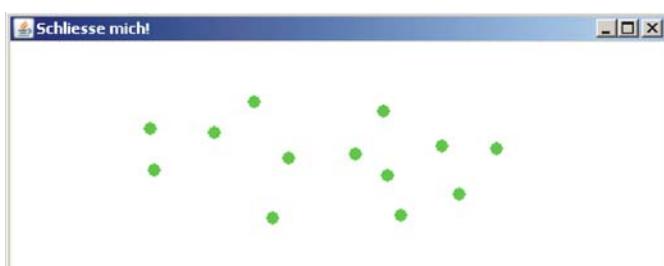
## 9.4 Auf Mausereignisse reagieren

Wir haben mit **WindowListener** und **ActionListener** bereits zwei Interaktionsmöglichkeiten kennen gelernt. Es fehlt noch eine wichtige, die Reaktion auf Mausereignisse. Im folgenden Beispiel wollen wir für einen Mausklick innerhalb des Fensters einen grünen Punkt an die entsprechende Stelle zeichnen. Hier wird einfach das Interface **MouseListener** implementiert oder es werden die leeren Methoden der Klasse **MouseAdapter** überschrieben.

```

1 import java.awt.*;
2 import java.awt.event.*;
3 public class MausKlick extends MeinFenster {
4     public MausKlick(String titel, int w, int h){
5         super(titel, w, h);
6         setSize(w, h);
7         // Wir verwenden eine innere anonyme Klasse. Kurz und knapp.
8         addWindowListener(new WindowAdapter() {
9             public void windowClosing(WindowEvent e) {
10                 System.exit(0);
11             }
12         });
13         addMouseListener(new MouseAdapter() {
14             public void mousePressed(MouseEvent e) {
15                 Graphics g = getGraphics();
16                 g.setColor(Color.green);
17                 g.fillOval(e.getX(), e.getY(), 10, 10);
18             }
19         });
20         setVisible(true);
21     }
22
23     public static void main( String[] args ) {
24         MausKlick f = new MausKlick("Schliesse mich!", 500, 200);
25     }
26 }
```

Liefert nach mehrfachem Klicken mit der Maus, innerhalb des Fensters, folgende Ausgabe:



## 9.5 Zusammenfassung und Aufgaben

### Zusammenfassung

Das Paket `java.awt` bietet mit der Klasse **Frame** eine einfache Möglichkeit, Fenster zu erzeugen und grafische Ausgaben zu machen. Wir können eine Instanz der Klasse erzeugen und diese manipulieren oder sie als Basisklasse verwenden.

Das Konzept der Ereignisbehandlung, also Reaktion und Behandlung von Fenster- und Mausereignissen, wurde besprochen. Neben den Interfaces **WindowListener** und **MouseListener** existieren die Klassen **WindowAdapter** und **MouseAdapter**, die diese Interfaces mit leeren Funktionen implementiert haben. Von diesen Klassen können wir erben und die benötigten Funktionen überschreiben.

Einem erfolgreichen Einsatz dieses Konzepts gehen meistens viele praktische Übungen vor. Der Leser sollte nicht verzweifeln, wenn es nicht auf Anhieb funktioniert, sondern Beispiele aus diesem Buch nehmen und diese etwas abändern.

### Aufgaben

Übung 1) Entwerfen Sie mit Hilfe der Ihnen nun bekannten GUI-Elemente einen **Taschenrechner**, der die Funktionen  $+, -, \cdot, /$  mindestens anbieten sollte.

Übung 2) Entwickeln Sie für Ihre **Räuber-Beute-Simulation** eine entsprechende GUI und visualisieren Sie die Populationsdynamik.

Übung 3) Für **Conway's Game of Life** sollen Sie eine Oberfläche entwerfen, deren Initialisierung der Zellkonstellationen mit Mausereignissen realisiert sind. Bieten Sie Möglichkeiten, Zustände laden und speichern zu können.

## Tag 10: Appletprogrammierung

Java bietet die Möglichkeit, Programme zu entwerfen, die in Webseiten eingebettet werden und dort laufen können. Diese Programme heißen **Applets**. Die meisten auf Frames basierenden Anwendungen können schon mit wenigen Änderungen zu Applets umfunktioniert werden. Bevor wir aber mit den Applets starten, machen wir uns mit dem Aufbau einer einfachen Webseite und der Handhabung von HTML vertraut.

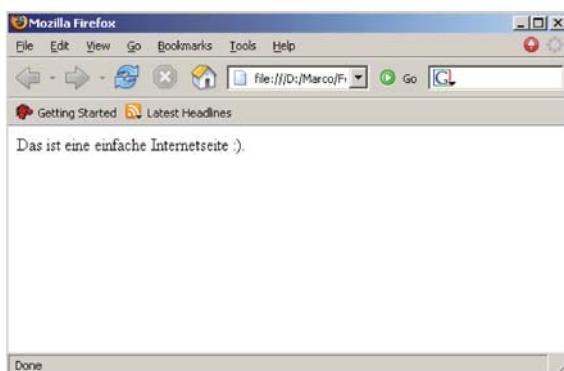
### 10.1 Kurzeinführung in HTML

HTML (= *HyperText Markup Language*) ist eine Sprache, mit der sich Internetseiten erstellen lassen. Jeder Internetbrowser kann diese Sprache lesen und die gewünschten Inhalte darstellen. Mit folgenden Zeilen lässt sich eine sehr einfache HTML-Seite erzeugen. Dazu gibt man die Codezeilen in einem beliebigen Editor ein und speichert sie mit der Endung „.html“.

```
1 <HTML>
2   <BODY>
3     Das ist eine einfache Internetseite :).
4   </BODY>
5 </HTML>
```

HTML

Wenn wir diese Seite mit einem Browser, z. B. dem Mozilla FireFox öffnen, erhalten wir folgende Ausgabe:



Zwischen den Tags, so nennen sich die Befehle in HTML, <HTML> und </HTML> steht der komplette Inhalt der zu schreibenden Internetseite. Es gibt dann noch zusätzliche Angaben zu der Seite, die zwischen <HEAD> und </HEAD> angegeben werden, z. B. den Titel der Seite und andere Meta-Informationen. Zwischen <BODY> und </BODY> steht der Abschnitt, der angezeigt werden soll. In unserem Beispiel ist es eine einfache Textzeile.

Wer sich mit der Gestaltung von Internetseiten beschäftigen möchte, dem würde ich das Buch von Stefan Münz empfehlen [36], der auch eine ausgesprochen gut gelungene Seite zum Thema HTML pflegt [43].

## 10.2 Applets im Internet

Der Kreativität der Programmierer ist es zu verdanken, dass zahlreiche Spiele und Anwendungen im Internet zu finden sind. Viele wurden mit Java geschrieben. Beispielsweise haben Studenten der Freien Universität für das Schachprogramm FUSc# einen Onlinezugang in Form eines Applets<sup>1</sup> für den FUSc#-Schachspielserver geschrieben [48].



So ist es möglich, interaktiv von außerhalb, nur mit einem Internetzugang und einem kleinen Java-Plugin gegen verschiedene Schachspieler oder Schachmotoren anzutreten.

Wir wollen auch am Ende dieses Kapitels selber ein kleines Tetris-Spiel programmieren und es der Öffentlichkeit auf einer Internetseite zum Spielen anbieten.

---

<sup>1</sup> Es ist geplant, den Quellcode auf der Internetseite dieses Buchs zu veröffentlichen.

## 10.3 Bauen eines kleinen Applets

Wir benötigen für ein Applet keinen Konstruktor. Wir verwenden lediglich die Basisklasse **Applet** und können einige Methoden überschreiben.

### **init**

Bei der Initialisierung übernimmt die `init`-Methode die gleiche Funktion wie es ein Konstruktor tun würde. Die Methode wird beim Start des Applets als erstes **genau einmal** aufgerufen. Es können Variablen initialisiert, Parameter von der HTML-Seite übernommen oder Bilder geladen werden.

### **start**

Die `start`-Methode wird aufgerufen, nachdem die Initialisierung abgeschlossen wurde. Sie kann sogar mehrmals aufgerufen werden.

### **stop**

Mit `stop` kann das Applet eingefroren werden bis es mit `start` wieder erwacht wird. Die `stop`-Methode wird z. B. aufgerufen, wenn das HTML-Fenster, indem das Applet gestartet wurde, verlassen<sup>2</sup> wird.

### **destroy**

Bei Beendigung des Applets, z. B. schließen des Fensters, wird `destroy` aufgerufen und alle Speicherressourcen werden wieder freigegeben.

## 10.4 Verwendung des Appletviewers

Damit wir nicht beginnen, ein Applet nur für den einen oder anderen populären Browser zu optimieren, verwenden wir, zur Anzeige unserer selbst entwickelten Applets, das Programm **Appletviewer**. Selbstverständlich wären alle gängigen Browser, wie z. B. „FireFox“ oder „Internet Explorer“ dazu ebenfalls in der Lage. Der Appletviewer hat aber den Vorteil, dass er neben der Ausgabe in einem Fenster, die Aktionen in eine Konsole schreibt.

Nach der Installation von Java steht uns dieses Programm zur Verfügung. Wir können die HTML-Seite, in der ein oder mehrere Applets eingebettet sind, aufrufen. Es wird separat für jedes Applet ein Fenster geöffnet und das Applet darin gestartet. Nun lassen sich alle Funktionen quasi per Knopfdruck ausführen und testen.

So starten wir den Appletviewer:

```
appletviewer <html-seite.html>
```

---

<sup>2</sup> Mit dem Verlassen des Fensters ist gemeint, dass der aktuelle Fokus auf ein anderes Fenster gesetzt wurde.

Verwenden wir für das erste Applet folgenden Programmcode und testen das Applet mit dem Appletviewer.

```

1 import java.awt.*;
2 import java.applet.*;
3
4 public class AppletZyklus extends Applet {
5     private int zaehler;
6     private String txt;
7
8     public void init(){
9         zaehler=0;
10        System.out.println("init");
11    }
12
13    public void start(){
14        txt = "start: "+zaehler;
15        System.out.println("start");
16    }
17
18    public void stop(){
19        zaehler++;
20        System.out.println("stop");
21    }
22
23    public void destroy(){
24        System.out.println("destroy");
25    }
26
27    public void paint(Graphics g){
28        g.drawString(txt, 20, 20);
29    }
30 }
```

Folgende HTML-Seite erzeugt das Applet. Die beiden Parameter width und height legen die Größe des Darstellungsbereichs fest.



```

1 <HTML>
2   <BODY>
3     <APPLET width=200 height=50 code="AppletZyklus.class"></APPLET>
4   </BODY>
5 </HTML>
```

Wir erhalten nach mehrmaligem Stoppen und wieder Starten folgende grafische Ausgabe:



Die Konsole liefert die Informationen zu den verwendeten Methoden:

```
C:\Applets>appletviewer AppletZyklus.html
init
start
stop
start
stop
start
stop
start
stop
destroy
```

## 10.5 Eine Applikation zum Applet umbauen

In den meisten Fällen ist es sehr einfach, eine Applikation zu einem Applet umzufunktionieren. Wir erinnern uns an die Klasse **Farbbeispiel** aus Abschnitt 9.2.3.

```
1 import java.awt.*;
2 import java.util.Random;
3
4 public class Farbbeispiel extends Frame {
5     public Farbbeispiel(String titel) {
6         setTitle(titel);
7         setSize(500, 300);
8         setBackground(Color.lightGray);
9         setForeground(Color.red);
10        setVisible(true);
11    }
12
13    public void paint(Graphics g){
14        Random r = new Random();
15        for (int y=30; y<getHeight()-10; y += 15)
16            for (int x=12; x<getWidth()-10; x += 15) {
17                g.setColor(new Color(r.nextInt(256),
18                                     r.nextInt(256),
19                                     r.nextInt(256)));
20                g.fillRect(x, y, 10, 10);
21                g.setColor(Color.BLACK);
22                g.drawRect(x - 1, y - 1, 10, 10);
23            }
24    }
25
26    public static void main(String[] args) {
27        Farbbeispiel t = new Farbbeispiel("Viel Farbe im Fenster");
28    }
29}
```

Jetzt wollen wir in einfachen Schritten ein Applet daraus erstellen.

### 10.5.1 Konstruktor zu init

Da ein Applet keinen Konstruktor benötigt, wird die Klasse innerhalb einer HTML-Seite erzeugt. Wir können an dieser Stelle die Parameter für width, height und titel übergeben und in init setzen.

```

1 <HTML>
2   <BODY>
3     <APPLET CODE = "FarbBeispielApplet.class" WIDTH=500 HEIGHT=300>
4       <PARAM NAME="width" VALUE=500>
5       <PARAM NAME="height" VALUE=300>
6       <PARAM NAME="titel" VALUE="Farbe im Applet">
7     </APPLET>
8   </BODY>
9 </HTML>

```

```

1 import java.awt.*;
2 import java.applet.*;
3 import java.util.Random;
4
5 public class FarbBeispielApplet extends Applet {
6   private int width;
7   private int height;
8   private String titel;
9
10  public void init() {
11    // Parameterübernahme
12    width = Integer.parseInt(getParameter("width"));
13    height = Integer.parseInt(getParameter("height"));
14    titel = getParameter("titel");
15
16    setSize(width, height);
17    setBackground(Color.lightGray);
18    setForeground(Color.red);
19  }

```

### 10.5.2 paint-Methoden anpassen

Für unser Beispiel können wir den Inhalt der `paint`-Methode komplett übernehmen.

```

21 public void paint(Graphics g){
22   Random r = new Random();
23   for (int y=30; y<getHeight()-10; y += 15)
24     for (int x=12; x<getWidth()-10; x += 15) {
25       g.setColor(new Color(r.nextInt(256),
26                         r.nextInt(256),
27                         r.nextInt(256)));
28       g.fillRect(x, y, 10, 10);
29       g.setColor(Color.BLACK);
30       g.drawRect(x - 1, y - 1, 10, 10);
31     }
32 }

```

Das Applet ist fertig, wir können es mit dem Appletviewer starten und erhalten die gleiche Ausgabe.

### 10.5.3 TextField-Beispiel zum Applet umbauen

Im ersten Applet gab es keine Ereignisbehandlung. Aber auch dafür wollen wir uns ein einfaches Beispiel anschauen. Das TextField-Beispiel aus Abschnitt 9.3.2.3. Nach der Konvertierung zu einem Applet sieht es wie folgt aus:

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import java.applet.*;
4
5 public class GUI_Button_TextField_Applet extends Applet{
6     private int width;
7     private int height;
8     private String titel;
9
10    Button button1;
11    Label labell;
12    TextField textfield1;
13
14    public void init() {
15        // Parameterübergabe
16        width = Integer.parseInt(getParameter("width"));
17        height = Integer.parseInt(getParameter("height"));
18        titel = getParameter("titel");
19        setSize(width, height);
20
21        // wir bauen einen ActionListener, der nur auf Knopfdruck
22        // reagiert
23        ActionListener aktion = new Knopfdruck();
24
25        setLayout(new FlowLayout());
26
27        textfield1 = new TextField("hier steht schon was", 25);
28        add(textfield1);
29
30        button1 = new Button("Knopf");
31        add(button1);
32        button1.addActionListener(aktion);
33        button1.setActionCommand("b1");
34
35        labell = new Label("noch steht hier nicht viel");
36        add(labell);
37    }
38
39    private void Button1Clicked(){
40        String txt = textfield1.getText();
41        labell.setText(txt);
42    }
43
44    // ****
45    // Innere Klassen für das Eventmanagement
46    class Knopfdruck implements ActionListener{
47        public void actionPerformed (ActionEvent e){
48            // wir behandeln die Ereignisse
49            String cmd = e.getActionCommand();
50            if (cmd.equals("b1"))
51                Button1Clicked();
52        }
53    }
54    // ****
55 }
```

Die Ereignisbehandlung verhält sich analog zu den bisher bekannten Applikationen.



## 10.6 Flackernde Applets vermeiden

Bei den ersten eigenen Applets wird eine unangenehme Eigenschaft auftauchen. Sie flackern. Schauen wir uns dazu einmal folgendes Programm an:

HTML

```

1 <HTML>
2   <BODY>
3     <APPLET width=472 height=482 code="BildUndStrukturFlackern.class">
4     </APPLET>
5   </BODY>
6 </HTML>
```

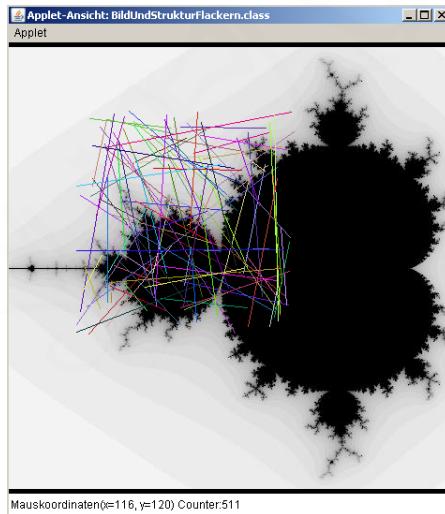
```

1 import java.applet.*;
2 import java.awt.*;
3 import java.awt.event.*;
4 import java.net.URL;
5 import java.lang.Math;
6 import java.util.Random;
7
8 /* Bild als Hintergrund und mausempfindliche Linienstruktur im
9  Vordergrund erzeugt unangenehmes flackern */
10 public class BildUndStrukturFlackern extends Applet {
11   int width, height, mx, my, counter=0;
12   final int N=100;
13   Point[] points;
14   Image img;
15   Random r;
16
17   public void init() {
18     width = getSize().width;
19     height = getSize().height;
20     setBackground(Color.black);
21
22     // Mauskoordinaten und MouseMotionListener
23     addMouseMotionListener(new MouseMotionHelper());
24
25     img = getImage(getDocumentBase(), "apfelmaennchen.jpg");
26     r = new Random();
27   }
28
29   // *****
30   // Klassenmethoden
31   private void zeichneLinien(Graphics g){
32     for (int i=1; i<N; i++) {
33       // wähle zufällige Farbe
34       g.setColor(new Color(r.nextInt(256),
35                           r.nextInt(256),
36                           r.nextInt(256)));
37
38       // verbinde die Punkte
39       g.drawLine(mx+(int)((r.nextFloat()-0.2)*(width/2)),
40                  my+(int)((r.nextFloat()-0.2)*(height/2)),
41                  mx+(int)((r.nextFloat()-0.2)*(width/2)),
42                  my+(int)((r.nextFloat()-0.2)*(height/2)));
43     }
44   }
45
46   private void zeichneBild(Graphics g){
47     g.drawImage(img, 0, 5, this);
48   }
49   // *****
50   // *****
```

```

52 private class MouseMotionHelper extends MouseMotionAdapter{
53     // Maus wird innerhalb der Appletfläche bewegt
54     public void mouseMoved(MouseEvent e) {
55         mx = e.getX();
56         my = e.getY();
57         showStatus("Mauskoordinaten (x=" +mx+ ", y=" +my+ ")"
58                    Counter:"+counter);
59         // ruft die paint-Methode auf
60         repaint();
61         e.consume();
62     }
63 }
64 // ****
65
66 public void paint(Graphics g) {
67     zeichneBild(g);
68     zeichneLinien(g);
69     counter++;
70 }
71 }
```

Bei der Bewegung der Maus über das Apfelmännchen werden zufällige Linien mit zufälligen Farben erzeugt. Die Darstellung ist relativ rechenintensiv und das Applet beginnt zu Flackern. Nebenbei erfahren wir in diesem Beispiel, wie wir dem Browser mit der Methode showStatus eine Ausgabe geben können.



### 10.6.1 Die Ghosttechnik anwenden

Unsere erste Idee an dieser Stelle könnte sein, die Darstellung in der Art zu beschleunigen, dass wir zunächst auf einem unsichtbaren Bild arbeiten und dieses anschließend neu zeichnen.

Diese Technik könnten wir als **Ghosttechnik** bezeichnen, da wir auf einem unsichtbaren Bild arbeiten und es anschließend wie von Geisterhand anzeigen lassen.

```

1 import java.applet.*;
2 import java.awt.*;
3 import java.awt.event.*;
4 import java.net.URL;
5 import java.lang.Math;
6 import java.util.Random;
7
8 /* Bild als Hintergrund und mausempfindliche Linienstruktur im
9 Vordergrund erzeugt unangenehmes Flackern, trotz Ghostimage!
10 */
11 public class BildUndStrukturFlackernGhost extends Applet {
12     int width, height, mx, my, counter=0;
13     final int N=100;
14     Point[] points;
15     Image img;
16
17     // ++++++
18     Image img_ghost;
19     Graphics g_ghost;
20     // ++++++
21
22     Random r;
23
24     public void init() {
25         width = getSize().width;
26         height = getSize().height;
27         setBackground(Color.black);
28
29         // Mauskoordinaten und MouseMotionListener
30         addMouseMotionListener(new MouseMotionHelper());
31
32         img = getImage(getDocumentBase(), "fractalkohl.jpg");
33
34         // ++++++
35         img_ghost = createImage(width, height);
36         g_ghost = img_ghost.getGraphics();
37         // ++++++
38
39         r = new Random();
40     }
41
42     // ****
43     // Klassenmethoden
44     private void zeichneLinien(Graphics g){
45         for (int i=1; i<N; i++) {
46             // wähle zufällige Farbe
47             g.setColor(new Color(r.nextInt(256),
48                                 r.nextInt(256),
49                                 r.nextInt(256)));
50
51             // verbinde N zufällig erzeugte Punkte
52             g.drawLine(mx+(int)((r.nextFloat()-0.2)*(width/2)),
53                         my+(int)((r.nextFloat()-0.2)*(height/2)),
54                         mx+(int)((r.nextFloat()-0.2)*(width/2)),
55                         my+(int)((r.nextFloat()-0.2)*(height/2)));
56         }
57     }
58
59     private void zeichneBild(Graphics g){
60         g.drawImage(img, 0, 0, this);
61     }
62     // ****
63
64     // ****
65     private class MouseMotionHelper extends MouseMotionAdapter{
66         // Maus wird innerhalb der Appletfläche bewegt

```

```

67  public void mouseMoved(MouseEvent e) {
68      mx = e.getX();
69      my = e.getY();
70      showStatus("Mauskoordinaten (x=" +mx+ ", y=" +my+ ")"
71                 Counter:" +counter);
72      // ruft die paint-Methode auf
73      repaint();
74      e.consume();
75  }
76  // ****
77
78  public void paint(Graphics g) {
79      zeichneBild(g_ghost);
80      zeichneLinien(g_ghost);
81
82      // ++++++
83      g.drawImage(img_ghost, 0, 0, this);
84      // ++++++
85
86      counter++;
87  }
88 }

```

Wenn wir dieses Applet starten, müssen wir ebenfalls ein Flackern feststellen, es ist sogar ein wenig schlimmer geworden. Im Prinzip stellt diese Technik eine Verbesserung dar. In Kombination mit dem folgenden Tipp, lassen sich rechenintensive Grafikausgaben realisieren.

### 10.6.2 Die paint-Methode überschreiben

Der Grund für das permanente Flackern ist der Aufruf der update-Funktion. Die update-Funktion sorgt dafür, dass zunächst der Hintergrund gelöscht und anschließend die paint-Methode aufgerufen wird.

Da wir aber von der Appletklasse erben, können wir diese Methode einfach überschreiben!

```

public void update( Graphics g ) {
    zeichneBild(g_ghost);
    zeichneLinien(g_ghost);

    g.drawImage(img_ghost, 0, 0, this);
}

public void paint( Graphics g ) {
    update(g);
    counter++;
}

```

Das gestartete Applet zeigt nun kein Flackern mehr.

## 10.7 Ein Beispiel mit mouseDragged

Abschließend wollen wir uns noch ein Beispiel für die Verwendung der *mouseDragged()*-Methode anschauen. Mit der Maus kann ein kleines Objekt mit der gedrückten linken Maustaste verschoben werden. In diesem Fall wird der kleine Elch im schwedischen Sonnenuntergang platziert.

```

1 import java.applet.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class MausBild extends Applet {
6     int width, height, x, y, mx, my;
7     Image img, img2, img_ghost;
8     Graphics g_ghost;
9     boolean isInBox = false;
10
11    public void init() {
12        width = getSize().width;
13        height = getSize().height;
14        setBackground(Color.black);
15
16        addMouseListener(new MouseHelper());
17        addMouseMotionListener(new MouseMotionHelper());
18
19        img = getImage(getDocumentBase(), "schweden.jpg");
20        img2 = getImage(getDocumentBase(), "elch.jpg");
21
22        img_ghost = createImage(width, height);
23        g_ghost = img_ghost.getGraphics();
24
25        // mittige Position für den Elch zu Beginn
26        x = width/2 - 62;
27        y = height/2 - 55;
28    }
29
30    // ****
31    private class MouseHelper extends MouseAdapter{
32        public void mousePressed(MouseEvent e) {
33            mx = e.getX();
34            my = e.getY();
35            // ist die Maus innerhalb des Elch-Bildes?
36            if (x<mx && mx<x+124 && y<my && my<y+111) {
37                isInBox = true;
38            }
39            e.consume();
40        }
41
42        public void mouseReleased(MouseEvent e) {
43            isInBox = false;
44            e.consume();
45        }
46    }
47    // ****
48
49    // ****
50    private class MouseMotionHelper extends MouseMotionAdapter{
51        public void mouseDragged(MouseEvent e) {
52            if (isInBox) {
53                int new_mx = e.getX();
54                int new_my = e.getY();
55
56                // Offset ermitteln

```

```

57         x += new_mx - mx;
58         y += new_my - my;
59         mx = new_mx;
60         my = new_my;
61
62         repaint();
63         e.consume();
64     }
65 }
66 // ****
67
68 public void update( Graphics g ) {
69     g_ghost.drawImage(img, 0, 0, this);
70     g_ghost.drawImage(img2, x, y, this);
71     g.drawImage(img_ghost, 0, 0, this);
72 }
73
74 public void paint( Graphics g ) {
75     update(g);
76 }
77 }
78 }
```

Die Variablen `mx` und `my` speichern die aktuelle Mausposition, wenn die linke Maustaste gedrückt vorliegt. Die Methode `mousePressed` setzt den boolean `isInBox` auf `true`, wenn die Maus innerhalb des Elchbildes gedrückt wurde.

Die Methode `mouseDragged` berechnet die neuen Koordinaten des Elchbildes und setzt das Bild neu, falls `isInBox` auf `true` gesetzt ist.



## 10.8 Zusammenfassung und Aufgaben

### Zusammenfassung

Applets sind eine feine Sache. Aus Applikationen lassen sich schnell Applets bauen. Wenn einmal verstanden wurde, wie sich rechenintensive Ausgaben durch die

Ghosttechnik und dem Überschreiben der update-Methode realisieren lassen, können Internetanwendungen dem Entwickler und dem Anwender sehr viel Spaß bereiten.

### Aufgaben

Übung 1) Bieten Sie Ihren im vorigen Abschnitt entwickelten **Taschenrechner** oder eines der anderen Anwendungen auf einer Webseite als Applet an.

Übung 2) Entwickeln Sie ein interaktives Brettspiel (Dame, Mühle, TicTacToe, ...).

Übung 3) Entwerfen Sie eine Version eines der Klassiker **Minesweeper**, **Othello** oder **BreakOut**.

## Tag 11: Techniken der Programmierung

In diesem Kapitel werden wir weniger auf die Effizienz einzelner Problemlösungen zu sprechen kommen als vielmehr verschiedene Ansätze aufzeigen, mit denen Probleme gelöst werden können. Es soll ein Einblick in die verschiedenen Programmiertechniken zum Entwurf von Programmen geben. Dieses Kapitel ist so aufgebaut, dass zunächst ein paar Begriffe und Techniken erläutert werden. Anschließend festigen wir diese mit kleinen algorithmischen Problemen und deren Lösungen. Am Ende des Kapitels beschäftigen wir uns mit dem Problem der Sortierung.

### 11.1 Der Begriff Algorithmus

Das Thema **Algorithmik** ist ein weites Feld und beschäftigt sich mit der effizienten Lösung von Problemen. Es gibt verschiedene Möglichkeiten an ein Problem heranzugehen und ein Programm zu schreiben, das dieses Problem löst. Manchmal ist es sogar möglich, ein allgemeines Programm zu formulieren, mit dem sogar viele ähnliche Probleme gelöst werden können.

Wie der Name Algorithmik vermuten lässt, beschäftigen wir uns im folgenden intensiv mit dem Begriff **Algorithmus**. Die Definitionen von Algorithmus und Programm (siehe dazu Abschnitt 2.5) sind sehr ähnlich, wobei der Algorithmus etwas abstrakter zu verstehen ist.

#### Definition des Begriffs *Algorithmus*

Mit **Algorithmen** bezeichnen wir genau definierte Handlungsvorschriften zur Lösung eines Problems oder einer bestimmten Art von Problemen [46]. Es geht darum, eine genau spezifizierte Abfolge von Anweisungen auszuführen, um ein gegebenes Problem zu lösen.

## 11.2 Entwurfs-Techniken

Es gibt verschiedene Techniken für den Entwurf von Algorithmen, einige werden wir in dem folgenden Abschnitt kennenlernen.

### 11.2.1 Prinzip der Rekursion

Im Abschnitt 6.1 haben wir gelernt, dass es sinnvoll ist, Programmteile zusammenzufassen und unsere Programme modular zu gestalten. Unsere Module sind Funktionen, die für bestimmte Eingaben etwas ausführen oder berechnen sollen und möglicherweise sogar ein Ergebnis liefern. Falls wir einen Programmabschnitt mehrmals verwenden wollen, brauchen wir diesen nicht mehrfach zu schreiben, es genügt, den Funktionsnamen und die Eingabeparameter anzugeben.

**Rekursion** bedeutet in diesem Zusammenhang, dass wir eine Funktion aufrufen, die sich selber wieder aufruft.

Das klingt zunächst sehr sonderbar, hat aber durchaus Vorteile. Angenommen wir haben ein Problem zu lösen, das auf einer großen Datenmenge basiert. Die Idee, die der Rekursion innenwohnt, ist die, dass wir das Problem kleiner machen und zuerst versuchen, das kleinere Problem zu lösen. Durch eine geeignete Verknüpfung der Lösung des kleineren Problems mit dem eigentlichen Problem hoffen wir, das Gesamtproblem lösen zu können. Der Aufruf der Funktion durch sich selbst wird so oft vorgenommen bis ein so kleines Problem vorliegt, dessen Lösung leicht zu formulieren ist.

Angenommen, wir wollen viele Daten sortieren, dann könnten wir das kleinste Teilproblem als die Sortierung zweier Daten ansehen. Nach einem Größenvergleich steht die richtige Reihenfolge beider Werte fest.

Das zweite Teilproblem ist die Zusammenfassung einer bereits bestehenden Lösung und eines neuen Datums. Wir haben beispielsweise wieder das Sortierungsproblem und es gibt eine bereits sortierte Liste. Die Frage ist jetzt, wo das neue Datum einzutragen ist, damit die Liste sortiert bleibt. Dazu könnten wir das Element beginnend von vorn mit dem jeweiligen Element in der Liste vergleichen und so entscheiden, ob es an diese Stelle gehört und alle einen Platz aufrutschen müssen, oder ob wir das nächste Element untersuchen. Dieses Sortierverfahren nennt sich *InsertionSort* und wird als Pseudocode und durch eine Implementierung später bei den Sortieralgorithmen vertieft.

Als konkretes Beispiel schauen wir uns die Berechnung der Fakultät an, da den Sortieralgorithmen, aufgrund Ihrer wichtigen Stellung, ein eigener Abschnitt gewährt ist.

### *Rekursionsbeispiel Fakultät*

Die Fakultät  $n!$  für eine natürliche Zahl  $n$  ist definiert als:

$$n! := n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$$

Sicherlich können wir schnell eine Funktion schreiben, die für eine Eingabe  $n$  diese Berechnung vornimmt, z. B.:

```
public static int fakultaet(int n){
    int erg = 1;
    for (int i=n; i>1; i--)
        erg *= i;
    return erg;
}
```

Wir können die Funktion `fakultaet` durch die Anwendung der Rekursionstechnik sehr viel eleganter aufschreiben:

```
public static int fakultaet(int i){
    if (i==1)
        return 1;
    else
        return i * fakultaet(i-1);
}
```

In den meisten Fällen verkürzt sich die Notation durch eine rekursive Formulierung erheblich. Die Programme werden zwar kürzer und anschaulicher, aber der Speicher- und Zeitaufwand nimmt mit jedem Rekursionsschritt zu. Bei der Abarbeitung einer rekursiven Funktion werden in den meisten Fällen alle in der Funktion vorkommenden Parameter erneut im Speicher angelegt und es wird mit diesen weitergearbeitet. Für zeitkritische Programme sollte bei der Implementierung aus Gründen der Effizienz auf Rekursion verzichtet werden<sup>1</sup>.

Die rekursiv formulierte Fakultätsfunktion lässt sich für  $n = 10$  beispielsweise mit folgender Befehlszeile in einem Java-Programm aufrufen:

```
int fakult = fakultaet(10);
```

---

<sup>1</sup> Da beispielsweise Schachprogramme auf rekursiv formulierten Algorithmen basieren und sehr rechenintensiv sind, konnten wir in der Schachprogrammier AG bei einem Test feststellen, dass die Umformulierung zu einer nicht-rekursiven Variante eine erhebliche Performancesteigerung zur Folge hatte [26].

### *Rekursionsbeispiel Fibonacci-Zahlen*

Die Fibonacci-Folge ist ein häufig verwendetes Beispiel für rekursive Methoden. Sie beschreibt beispielsweise das Populationverhalten von Kaninchen. Zu Beginn gibt es ein Kaninchenpaar. Jedes neugeborene Paar wird nach zwei Monaten geschlechtsreif. Geschlechtsreife Paare werfen pro Monat ein weiteres Paar.

Um die ersten  $n$  Zahlen dieser Folge zu berechnen, schauen wir uns die folgende Definition an:

$$fib(0) = 0 \text{ und } fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2), \text{ für } n \geq 2$$

Die Definition selbst ist schon rekursiv. Beginnend bei dem kleinsten Funktionswert lässt sich diese Folge, da der neue Funktionswert gerade die Summe der beiden Vorgänger ist, leicht aufschreiben:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots .$$

Eine Javafunktion dafür zu schreiben ist sehr einfach und als Übungsaufgabe im Aufgabenteil zu finden.

#### **11.2.2 Brute Force**

Die Übersetzung dieser Technik klingt zunächst sehr verwirrend. **Brute Force** bedeutet übersetzt soviel wie „Methode der rohen Gewalt“ [47]. Damit ist gemeint, dass alle möglichen Kombinationen, die für eine Lösung in Frage kommen können, durchprobiert werden. Claude Elwood Shannon, der in diesem Zusammenhang erwähnt werden muss, hat die Schachprogrammieralgorithmen damit revolutioniert.

Seine Idee, Brute-Force mit einer Bewertungsfunktion zu verknüpfen und daraus den besten Zug zu ermitteln, war revolutionär für die Spieltheorie. Das von ihm 1949 veröffentlichte Verfahren ist unter dem Namen **MinMax-Algorithmus** bekannt [27]. 1958 haben Newell, Shaw und Simon [28] dieses Verfahren zu dem **Alpha-Beta-Algorithmus**<sup>2</sup> erweitert und gezeigt, dass man auf die Berechnung einer ganzen Reihe von schlechten Zügen verzichten kann, wenn die guten Züge in den zu untersuchenden Zuglisten weit vorne stehen.

Eine ausführliche Analyse und eine Implementierung des Min-Max-Algorithmus zur Berechnung des besten Spielzugs beim TicTacToe-Spiel wird es in Kapitel 13 geben.

---

<sup>2</sup> Ausführliche und sehr lesenswerte Beschreibungen vieler aktueller Varianten und Erweiterungen lassen sich in der Doktorarbeit von Aske Plaat [29] nachlesen.

Hätten Sie die **Brute Force** Strategie bei einem Schachprogramm vermutet?

Moderne Schachprogramme basieren auf der Brute-Force-Technik. Alle möglichen Züge bis zu einer bestimmten Suchtiefe werden ausprobiert und die resultierenden Stellungen bewertet. Anschließend führen die berechneten Bewertungen dazu, aus den zukünftigen möglichen Stellungen den für die aktuelle Stellung besten Zug zu bestimmen.

Man könnte hier auf die Frage kommen: *Warum können Schachprogramme dann nicht einfach bis zum Partieende rechnen und immer gewinnen?* Das ist in der Tatssache begründet, dass das Schachspiel sehr komplex ist. Es gibt bei einer durchschnittlichen Zuganzahl von 50 Zügen<sup>3</sup>, schätzungsweise  $10^{120}$  unterschiedliche Schachstellungen. Im Vergleich dazu wird die Anzahl der Atome im Weltall auf  $10^{80}$  geschätzt. Das ist der Grund, warum Schach aus theoretischer Sicht noch interessant ist.

Wenn man sich vergegenwärtigt, dass die letzten großen Spiele Mensch gegen Maschine so ausgingen (ich habe eine kleine Auswahl getroffen):

1996	<b>Kasparov</b>	-	Deep Blue	4-2
1997	Kasparov	-	<b>Deep Blue</b>	2.5-3.5
2002	Kramnik	-	Deep Fritz	4-4
2003	Kasparov	-	Deep Junior	3-3
2003	Kasparov	-	X3D Fritz	2-2
2006	Kramnik	-	<b>Deep Fritz</b>	2-4

könnte man der Meinung sein, dass der Mensch eigentlich chancenlos ist.

Obwohl die heutige Maschinengeneration mit den besten Schachspielern der Welt konkurrieren kann, bin ich der festen Überzeugung, dass ein Schach-Großmeister, wenn er sich der Tatsache bewusst ist, dass er gegen eine Maschine spielt und nicht gegen einen Menschen (mit den Informationen, wie eine Maschine tickt), diese auch besiegen kann<sup>4</sup>. Denn nur der Mensch ist in der Lage, tiefe und langfristige Pläne zu entwerfen, eine Maschine nicht, sie probiert einfach alles aus.

### 11.2.3 Greedy

Greedy übersetzt bedeutet *gierig*, und genau so lässt sich diese Entwurfstechnik auch beschreiben. Für ein Problem, das in Teilschritten gelöst werden kann, wählt man für

<sup>3</sup> Ein Zug ist definiert durch zwei Halbzüge, Weiss zieht einmal und Schwarz zieht einmal. Moderne Schachprogramme schaffen in Turnierpartien je nach Komplexität der Stellung eine durchschnittliche Suchtiefe von 18 Halbzügen.

<sup>4</sup> Besonders in Erinnerung ist mir die 3. Partie des Matches **Kasparov-X3D Fritz** im Jahr 2003 geblieben, bei der die Maschine in eine geschlossene Stellung geführt wurde und selbst einem Laien klar war, dass die von der Maschine gespielten Züge schlecht sein müssen.

jeden Teilschritt die Lösung aus, die den größtmöglichen Gewinn verspricht. Das hat aber zur Folge, dass der Algorithmus für bestimmte Problemstellungen nicht immer zwangsläufig die beste Lösung findet. Es gibt aber Klassen von Problemen, bei denen dieses Verfahren erfolgreich arbeitet.

Als praktisches Beispiel nehmen wir die Geldrückgabe an der Kasse. Kassierer verfahren meistens nach dem Greedy-Algorithmus. Gebe zunächst den Schein oder die Münze mit dem größtmöglichen Wert heraus, der kleiner oder gleich der Restsumme ist. Mit dem Restbetrag wird gleichermaßen verfahren.

Beispiel: Rückgabe von 1 Euro und 68 Cent.

Die Lösung von links nach rechts:



Für dieses Beispiel liefert der Algorithmus immer die richtige Lösung. Hier sei angemerkt, dass es viele Problemlösungen der Graphentheorie [21, 22] gibt, die auf der Greedy-Strategie basieren.

#### 11.2.4 Dynamische Programmierung, Memoisierung

Bei der Dynamischen Programmierung wird die optimale Lösung aus optimalen Teillösungen zusammengesetzt. Teillösungen werden dabei in einer geeigneten Datenstruktur gespeichert, um kostspielige Rekursionen zu vermeiden. Rekursion kann kostspielig sein, wenn gleiche Teilprobleme mehrfach gelöst werden. Einmal berechnete Ergebnisse werden z. B. in Tabellen gespeichert und später wird gegebenenfalls darauf zugegriffen.

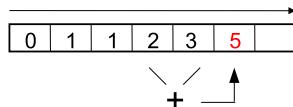
Memoisierung ist dem Konzept der Dynamischen Programmierung sehr ähnlich. Eine Datenstruktur wird beispielsweise in einen Rekursionsformel so eingearbeitet, dass auf bereits ermittelte Daten zurückgegriffen werden kann. Anhand der uns bereits gut bekannten Fibonacci-Zahlen wollen wir diese Verfahren untersuchen.

##### *Fibonacci-Zahlen mit Dynamischer Programmierung*

Die Erzeugung der Fibonacci-Zahlen lässt sich mit Dynamischer Programmierung gegenüber der Version aus Abschnitt 11.2.1 wesentlich effizienter realisieren. Wir müssen uns eine geeignete Datenstruktur wählen. In diesem Fall ist eine Liste sehr hilfreich und wir könnten das Programm in etwa so formulieren:

```
n-elementige, leere Liste fibi erzeugen
fibi[0] = 0
fibi[1] = 1
for (i=2 to n)
    fibi[i] = fibi[i-1] + fibi[i-2]
```

Die Funktionswerte werden in einer Schleife ermittelt und können anschließend ausgegeben werden.



### Fibonacci-Zahlen mit Memoisierung

Im folgenden Beispiel verwenden wir die Datenstruktur **fibi** im Rekursionsprozess, um bereits ermittelte Zwischenlösungen wiederzuverwenden:

```
m-elementige, leere Liste fibi erzeugen
fibi[0] = 0
fibi[1] = 1
```

Die Initialisierung muss einmal beim Programmstart ausgeführt werden und erzeugt eine  $m$ -elementige, leere Liste **fibi**. Die ersten zwei Elemente tragen wir ein. Bei der Anfrage  $fib(n)$  mit einem  $n$  das kleiner als  $m$  ist, wird die Teilfolge der Fibonacci-Zahlen bis  $n$  in die Datenstruktur eingetragen.

```
fib(n) = if (fibi[n] enthält einen Wert)
           return fibi[n]
       else {
           fibi[n] = fib(n-1) + fib(n-2)
           return fibi[n]
       }
```

Entweder ist der Funktionswert bereits einmal berechnet worden und kann über die Liste **fibi** zurückgegeben werden, oder er wird rekursiv ermittelt und gespeichert.

Ein Beispiel für Dynamische Programmierung aus der Bioinformatik zur Berechnung der Ähnlichkeit zweier Sequenzen wäre der *Needleman-Wunsch-Algorithmus*, den wir in Abschnitt 11.4.3 besprechen werden.

### 11.2.5 Divide and Conquer

Das Divide-and-Conquer Verfahren (Teile und Herrsche) arbeitet rekursiv. Ein Problem wird dabei im **Divide-Schritt** in zwei oder mehrere Teilprobleme aufgespalten. Das wird solange gemacht bis die entstandenen Teilprobleme klein genug sind, um direkt gelöst zu werden. Die Lösungen werden dann in geeigneter Weise, im **Conquer-Schritt**, kombiniert und liefern am Ende eine Lösung für das Originalproblem.

Viele effiziente Algorithmen basieren auf dieser Technik [20, 19, 17]. In Abschnitt 11.4.2.3 werden wir die Arbeitsweise von Divide-and-Conquer anhand des Sortieralgorithmus *QuickSort* unter die Lupe nehmen.

## 11.3 Algorithmen miteinander vergleichen

Um Algorithmen zu analysieren und miteinander vergleichen zu können, müssen wir ein geeignetes Maß finden. Zum Scheitern verurteilt wäre sicherlich die Idee, mit einer Stoppuhr die verwendete Zeit für die Lösung eines Problems auf einem Rechner zu messen oder den Speicheraufwand zu notieren. Computer unterscheiden sich zu stark und die benötigte Zeit zur Lösung eines Problems auf einem einzigen Computer sagt nicht sehr viel über das Problem aus.

Es benötigt ein größeres Abstraktionsniveau, das unabhängig vom Rechner, dem Compiler und der verwendeten Programmiersprache ist. Daher wird die Anzahl der Elementaroperationen auf einem abstrakten Rechenmodell geschätzt.

Bei der Laufzeitanalyse eines Algorithmus lässt sich das Verhalten für unterschiedliche Situationen ermitteln. Beispielsweise ist man daran interessiert zu erfahren, wie sich dieser Algorithmus bei der schlechtmöglichen Eingabe verhält (*worst-case-Analyse*). Es könnte auch interessant sein, eine Abschätzung für den Aufwand über alle möglichen Eingaben (*average-case-Analyse*) zu untersuchen. An dieser Stelle würde ich folgende, weiterführende Literatur empfehlen: [20, 17, 19, 16, 18].

Schauen wir uns dazu ein kleines Beispiel an.

### Laufzeitanalyse der rekursiven Fakultätsfunktion

Die Fakultätsfunktion verhält sich im *average-* und *worst-case* gleich, da der Algorithmus immer den gleichen Weg zur Lösung eines Problems geht.

$$\begin{aligned} fac(1) &= 1 \\ fac(n) &= n * fac(n-1) \end{aligned}$$

Um für einen gegebenen Wert  $n$  die Fakultät zu berechnen, bedarf es  $n - 1$  Multiplikationen<sup>5</sup>. Wird  $n$  sehr groß (und das ist der interessante Fall) kann man die  $-1$  vernachlässigen. Daher werden wir dem Algorithmus eine **lineare** Laufzeit zuschreiben und sagen, dass seine Komplexität  $O(n)$  ist.  $O(n)$  bedeutet nichts anderes, als dass die benötigte Zeit zum Lösen des Problems linear mit der Größe der Eingabe ( $n$ ) wächst.

## 11.4 Kleine algorithmische Probleme

Um die Techniken aus Abschnitt 11.2 besser zu verstehen, werden wir im folgenden Abschnitt kleine algorithmische Probleme lösen. Zu diesem Thema lassen sich sehr viele gute Bücher finden, aber diese beiden dürfen in keiner Bibliothek fehlen [15, 11].

### 11.4.1 Identifikation und Erzeugung von Primzahlen mit Brute Force

Primzahlen sind natürliche Zahlen, die nur durch 1 und sich selber ganzzahlig teilbar sind. 2 ist die kleinste Primzahl. Der im Folgenden vorgestellte Algorithmus ähnelt sehr dem „Sieb des Eratosthenes“.

Um für eine Zahl  $p$  zu testen, ob sie diese Eigenschaften erfüllt und somit eine Primzahl ist, können wir die **Brute-Force** Strategie anwenden und einfach alle Zahlen, die kleiner als  $p$  sind, durchtesten.  $p$  ist keine Primzahl, wenn es eine Zahl  $q$  mit  $2 \leq q \leq \sqrt{p}$  gibt, die  $p$  ganzzahlig teilt, also  $p \% q == 0$  ergibt (für die Funktionsweise des Modulo-Operators siehe Abschnitt 2.3.3).

Hier ein kleines Programm mit der Funktion `istPrimzahl`, das für eine Zahl  $p$  alle potentiellen Teiler von 2 bis  $\sqrt{p}$  auf ganzzahlige Teilbarkeit überprüft.

```

1 public static class PrimZahlen {
2     // Prüfe, ob p eine Primzahl ist
3     public boolean istPrimzahl(int p){
4         boolean istPrim = true;
5         if (p<2) return false;
6
7         for (int i=2; i<=Math.sqrt(p); i++){
8             if (p%i == 0){
9                 istPrim = false;
10                break;
11            }
12        }
13        return istPrim;
14    }
15    // Testet alle Zahlen von 0..1000 auf die Primzahleigenschaften

```

---

<sup>5</sup> Normalerweise spielt aber die Größe der Eingabe auch eine entscheidende Rolle. Das ist für unsere einfache Vorstellung der Laufzeitanalyse an dieser Stelle nicht wichtig.

```

17 public static void main(String [] args){
18     int pMax = 1000;
19     System.out.println("Primzahlen von 0 bis "+pMax);
20     for (int i=0; i<=pMax; i++)
21         if (PrimZahlen.istPrimzahl(i))
22             System.out.print(i+", ");
23     }
24 }
```

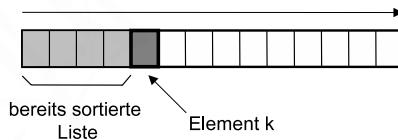
## 11.4.2 Sortieralgorithmen

Das Problem, unsortierte Daten in eine richtige Reihenfolge zu bringen eignet sich gut, um verschiedene Programmietechniken und Laufzeitanalysen zu veranschaulichen.

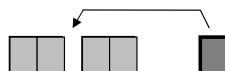
### 11.4.2.1 InsertionSort

Eine sehr einfache Methode Daten zu sortieren, ist das „Einfügen in den sortierten Rest“. Die Vorgehensweise des Algorithmus ist mit zwei kleinen Bildchen, die die beiden Arbeitsschritte zeigen, sehr leicht verständlich. Wir sortieren eine Liste, indem wir durch alle Positionen der Liste gehen und das aktuelle Element an dieser Position in die Teilliste davor einsortieren.

Wenn wir das Element  $k$  einsortieren möchten, können wir davon ausgehen, dass die Teilliste vor diesem Element, die Positionen 1 bis  $k - 1$ , bereits sortiert ist.



Um das Element  $k$  in die Liste einzufügen, prüfen wir alle Elemente der Teilliste, beginnend beim letzten Element, ob das Element  $k$  kleiner ist, als das gerade zu prüfende Element  $j$ . Sollte das der Fall sein, so rückt das Element  $j$  auf die Position  $j + 1$ . Das wird solange gemacht bis die richtige Position für das Element  $k$  gefunden wurde.



Als Beispielimplementierung schauen wir uns die Klasse **InsertionSort** an:

```

1 public class InsertionSort {
2     private static void insert(int[] a, int pos){
3         int value = a[pos];
4         int j      = pos-1;
5
6         // Alle Werte vom Ende zum Anfang der bereits sortierten Liste
7         // werden solange ein Feld nach rechts verschoben, bis die
8         // Position des Elements a[pos] gefunden ist. Dann wird das
9         // Element an diese Stelle kopiert.
10        while(j>0 && a[j]>value){
11            a[j+1] = a[j];
12            j--;
13        }
14        a[j+1] = value;
15    }
16
17    public static void sortiere(int[] x) {
18        // "Einfügen in den sortierten Rest"
19        for (int i=1; i<x.length; i++)
20            insert(x, i);
21    }
22
23    public static void main(String[] args) {
24        int[] liste = {0,9,4,6,2,8,5,1,7,3};
25        sortiere(liste);
26        for (int i=0; i<liste.length; i++)
27            System.out.print(liste[i]+ " ");
28    }
29 }
```

Nach der Ausführung ist die int-Liste sortiert und wird ausgegeben.

```
C:\JavaCode>java InsertionSort
0 1 2 3 4 5 6 7 8 9
```

Zur Laufzeit können wir sagen, dass die Komplexität des Algorithmus im average-case mit  $O(n^2)$  quadratisch ist. Je besser die Daten vorsortiert sind, desto schneller arbeitet das Verfahren. Im best-case ist sie sogar linear, also  $O(n)$ .

### 11.4.2.2 BubbleSort

Der BubbleSort-Algorithmus vergleicht der Reihe nach zwei benachbarte Elemente einer  $n$ -elementigen Liste  $x$  und vertauscht sie, falls sie nicht in der richtigen Reihenfolge vorliegen. Ist er am Ende der Liste angekommen, wird der Vorgang wiederholt. Der Algorithmus endet, wenn alle Elemente in der richtigen Reihenfolge vorliegen, im letzten Durchgang also keine Vertauschoperationen mehr stattgefunden haben. Dies geschieht nach maximal  $(n - 1) \cdot \frac{n}{2}$  Schritten. Folgender Algorithmus würde immer die maximale Anzahl an Schritten ausführen, selbst wenn die Liste bereits sortiert ist:

```
for (i=1 to n-1)
```

```

for (j=0 to n-i-1)
    if (x[j] > x[j+1])
        vertausche x[j] und x[j+1]

```

Aus theoretischer Sicht ist dieser Sortieralgorithmus sehr ineffizient. Die Komplexität ist quadratisch, also  $O(n^2)$  [20, 16, 17]. Aus praktischen Gesichtspunkten muss hier aber gesagt werden, dass zu sortierende Listen, die bereits schon eine gewisse Vorsortierung besitzen und nicht allzu groß sind, mit BubbleSort in der Praxis relativ schnell zu sortieren sind.

Beispielsweise ist das in der Schachprogrammierung so. Es werden Listen von legalen Zügen generiert und anschließend sortiert. Als Sortierungskriterium werden meistens Funktionen verwendet, die auf Heuristiken, wie z.B. „Wie oft hat sich dieser Zug in der Vergangenheit als sehr gut erwiesen?“, basieren. Da die Zuglisten relativ kurz sind (im Schnitt unter 50) und durch intelligente Zuggeneratoren meistens schon gut vorsortiert wurden, eignet sich BubbleSort ausgezeichnet [26].

Bei einer effizienten Implementierung bricht der Algorithmus bereits ab, wenn keine Tauschoperationen mehr durchgeführt wurden:

```

1 public class BubbleSort {
2     public static void sortiere(int[] x) {
3         boolean unsortiert=true;
4         int temp;
5
6         while (unsortiert){
7             unsortiert = false;
8             for (int i=0; i<x.length-1; i++)
9                 if (x[i] > x[i+1]) {
10                     temp      = x[i];
11                     x[i]      = x[i+1];
12                     x[i+1]   = temp;
13                     unsortiert = true;
14                 }
15             }
16         }
17
18     public static void main(String[] args) {
19         int[] liste = {0,9,4,6,2,8,5,1,7,3};
20         sortiere(liste);
21         for (int i=0; i<liste.length; i++)
22             System.out.print(liste[i]+ " ");
23     }
24 }

```

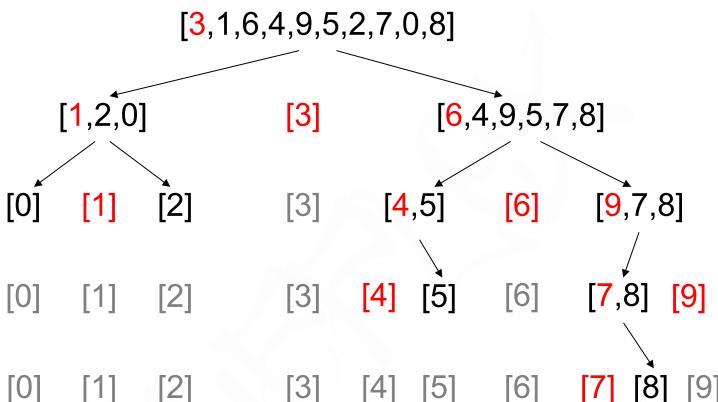
Die Liste wird sortiert ausgegeben:

```
C:\JavaCode>java BubbleSort
0 1 2 3 4 5 6 7 8 9
```

### 11.4.2.3 QuickSort

Der QuickSort-Algorithmus ist ein typisches Beispiel für die Entwurfstechnik Divide-and-Conquer. Um eine Liste zu sortieren, wird ein **Pivotelement**  $p$  ausgewählt und die Elemente der Liste in zwei neue Listen werden gespeichert, mit der Eigenschaft, dass in der ersten Liste die Elemente kleiner oder gleich  $p$  und in der zweiten Liste die Elemente größer  $p$  liegen. Mit den beiden Listen wird wieder genauso verfahren.

Machen wir uns das an einem Beispiel klar. Wir wollen in diesem Beispiel die Liste  $[3, 1, 6, 4, 9, 5, 2, 7, 0, 8]$  sortieren. Dazu wählen wir ein Pivotelement, z. B. das erste Element in der Liste, aus und teilen die Liste in zwei neuen Listen auf. Diese beiden Listen werden wieder aufgespalten usw.



Der Algorithmus fügt schließlich die einelementigen Listen nur noch zusammen, da deren Reihenfolge bereits richtig ist und liefert die sortierte Liste.

Es gibt verschiedene Möglichkeiten für die programmiertechnische Umsetzung.

```

1 import java.util.Random;
2
3 public class QuickSort {
4     public static void sortiere(int x[]) {
5         qSort(x, 0, x.length - 1);
6     }
7
8     public static void qSort(int x[], int links, int rechts) {
9         if (links < rechts) {
10             int i = partition(x, links, rechts);
11             qSort(x, links, i - 1);
12             qSort(x, i + 1, rechts);
13         }
14     }
15
16     public static int partition(int x[], int links, int rechts) {
17         int pivot, i, j, help;
18         pivot = x[rechts];
  
```

```

19     i      = links;
20     j      = rechts -1;
21     while(i <=j) {
22         if (x[i] > pivot) {
23             // tausche x[i] und x[j]
24             help = x[i];
25             x[i] = x[j];
26             x[j] = help;
27             j--;
28         } else i++;
29     }
30     // tausche x[i] und x[rechts]
31     help      = x[i];
32     x[i]      = x[rechts];
33     x[rechts] = help;
34
35     return i;
36 }
37
38 public static void main(String[] args) {
39     int[] liste = {0,9,4,6,2,8,5,1,7,3};
40     sortiere(liste);
41     for (int i=0; i<liste.length; i++)
42         System.out.print(liste[i]+ " ");
43 }
44 }
```

Auch der QuickSort-Algorithmus kann die int-Liste in die korrekte Reihenfolge bringen.

```
C:\JavaCode>java QuickSort
0 1 2 3 4 5 6 7 8 9
```

Für die Bestimmung der Laufzeit ist die Wahl des Pivotelements entscheidend. Im worst case wird immer genau das Element ausgesucht, welches die Liste so zerlegt, dass auf der einen Seite das Pivotelement vorliegt und auf der anderen Seite der Rest der Liste. Der Aufwand läge dann bei  $O(n^2)$ . Im besten Fall kann das Pivotelement die beiden Listen in zwei gleichgroße Listen zerlegen. In diesem Fall hat der Rechnungsbaum eine logarithmische Tiefe und die Laufzeit des Algorithmus ist nach oben mit  $O(n * \log(n))$  beschränkt. Das gilt auch für den average-case.

### 11.4.3 Needleman-Wunsch-Algorithmus

Mit dem *Needleman-Wunsch-Algorithmus* steigen wir etwas in die Bioinformatik ein. Im Jahre 1970 von Saul Needleman und Christian Wunsch formuliert [30], vergleicht dieser Algorithmus zwei Protein-Sequenzen<sup>6</sup> und gibt einen Ähnlichkeitsfaktor abhängig von einer Funktion zurück. Dieses Verfahren basiert auf Dynamischer Programmierung. Wir wollen in unserem Beispiel keine Aminosäuren vergleichen,

---

<sup>6</sup> Der menschliche Körper, sieht man mal vom Wasser ab, besteht zu über 70% aus Proteinen. Der Algorithmus ist allerdings auch in der Lage, beliebige Zeichensequenzen zu vergleichen.

sondern Zeichenketten und ein Maß für Ähnlichkeit berechnen. Deshalb sind einige Vereinfachungen vorgenommen worden.

Zunächst einmal zur vereinfachten Definition des Algorithmus. Gegeben sind zwei Zeichenketten  $x = (x_1, x_2, \dots, x_n)$  und  $y = (y_1, y_2, \dots, y_m)$ . Wie wir im Abschnitt 11.2.4 erfahren haben, benötigen wir eine Datenstruktur zum Speichern der Zwischenlösungen. In diesem Fall verwenden wir eine  $n \times m$ -Matrix  $E$ , die ganzzahlige Werte speichert. Wir interpretieren einen Eintrag in  $E(i, j)$  als besten Ähnlichkeitswert für die Teilequenzen  $(x_1, x_2, \dots, x_i)$  und  $(y_1, y_2, \dots, y_j)$ .

Als Startwerte setzen wir

$$E(0, 0) = 0$$

und

$$\begin{aligned} E(i, 0) &= 1, \quad \forall i = 0, 1, \dots, n \\ E(0, j) &= 1, \quad \forall j = 0, 1, \dots, m \end{aligned}$$

Die Rekursionsformel sieht dann wie folgt aus:

$$E(i, j) = \max \begin{cases} E(i - 1, j - 1) + \text{bonus}(x_i, y_j) \\ E(i - 1, j) \\ E(i, j - 1) \end{cases}$$

Die Bonusfunktion  $\text{bonus}(a, b)$  liefert eine 1, falls  $a$  und  $b$  gleiche Zeichen sind, andernfalls eine 0. So maximieren wir am Ende die größte Übereinstimmung.

Da wir jeden Eintrag der Matrix genau einmal berechnen und durchlaufen, haben wir eine Laufzeit von  $O(nm)$ . Dieser Algorithmus ist sehr schnell implementiert, aber für die Bioinformatik gibt es inzwischen bessere Algorithmen (z. B. *Smith-Waterman*, *Hirschberg*).

Hier ein Implementierungsbeispiel zu unserer Needleman-Wunsch-Variante:

```

1 public class NeedlemanWunsch{
2     private int[][] E;
3     private String n, m;
4
5     public NeedlemanWunsch(String a, String b){
6         n = a; m = b;
7         E = new int[n.length() + 1][m.length() + 1];
8         initialisiere();
9     }
10
11    public void initialisiere(){
12        // Starteintrag wird auf 0 gesetzt
13        E[0][0] = 0;
14
15        // füllt die erste Zeile und erste Spalte mit 0-en
16        for (int i=1; i<=n.length(); i++)
17            E[i][0] = 0;

```

```

18     for (int j=1; j<=m.length(); j++)
19         E[0][j] = 0;
20     }
21
22     private int cost(char a, char b){
23         if (a==b) return 1;
24         else return 0;
25     }
26
27     public int compare(){
28         for (int i=1; i<=n.length(); i++)
29             for (int j=1; j<=m.length(); j++)
30                 E[i][j] = Math.max(E[i-1][j-1]
31                             + cost(n.charAt(i-1), m.charAt(j-1)),
32                             Math.max(E[i-1][j], E[i][j-1]));
33         return E[n.length()][m.length()];
34     }
35
36     public static void main(String[] args){
37         String a = "Hallo Waldfee";
38         String b = "Hola fee";
39         NeedlemanWunsch NW = new NeedlemanWunsch(a, b);
40         System.out.println("Ergebnis: "+NW.compare());
41     }
42 }
```

Bei der Ausführung erhalten wir das Ergebnis 6, da die beiden Zeichenketten sechs gemeinsame Buchstaben in der gleichen Reihenfolge enthalten. Der Needleman-Wunsch Algorithmus ist auf vielfältige Weise erweiterbar, z. B. lassen sich mit kleinen Änderungen leicht Unterschiede in Textdateien finden. Dem interessierten Leser sei an dieser Stelle folgende weiterführende Literatur [23] empfohlen.

## 11.5 Zusammenfassung und Aufgaben

### Zusammenfassung

Wir haben verschiedene Entwurfstechniken für Algorithmen kennengelernt: Rekursion, Brute Force, Greedy, Dynamische Programmierung, Memoisierung und Divide and Conquer. Dabei wurden nicht nur die den Techniken zu Grunde liegenden Charakteristiken aufgezeigt, sondern auch praktische Beispiele gegeben. Eine kurze Einführung in die Laufzeitabschätzung von Algorithmen sollte einen kleinen Einblick in diese umfangreiche Thematik geben. Der Sortierung von Daten wurde eine besondere Aufmerksamkeit gewidmet.

### Aufgaben

Übung 1) Schreiben Sie eine rekursive Funktion fibonacci(int n), die die ersten  $n$  Zahlen der Fibonacci-Folge berechnet und ausgibt. Die Definition ist in Abschnitt 11.2.1 zu finden. Erweitern Sie anschließend Ihre Funktion derart, dass Teillösungen nur einmal berechnet werden.

Übung 2) (Collatz-Problem) Schreiben Sie eine Funktion, für die gilt:

$f(n+1) = f(n)/2$ , wenn  $f(n)$  gerade ist und  $f(n+1) = 3*f(n) + 1$  sonst.  $f(0)$  kann eine beliebige natürliche Zahl sein. Abbruchbedingung ist  $f(n) = 1$ .

Wenn bei irgendeiner Eingabe das Programm nicht terminiert, haben Sie vermutlich einen Fehler gemacht oder 1000\$ verdient, da Sie ein Gegenbeispiel für das Collatz-Problem gefunden haben.

siehe: <http://de.wikipedia.org/wiki/Collatz-Problem>

Übung 3) Erweitern Sie die in diesem Kapitel beschriebene Variante des Needleman-Wunsch-Algorithmus um die Ausgabe des Weges durch die Matrix. Dabei soll das Einfügen eines leeren Elements in  $x$  oder  $y$  mit dem Symbol '\_' dargestellt werden. Visualisieren Sie die Matrix während des Algorithmus.

Übung 4) Recherchieren Sie nach dem Knobelspiel *Türme von Hanoi* und formulieren Sie in Pseudocode einen rekursiven Algorithmus, der das Problem löst.

## Tag 12: Bildverarbeitung

In diesem Kapitel wollen wir uns mit der Bildverarbeitung in Java beschäftigen. Dabei werden wir lernen, wie Bilder im Rechner dargestellt und manipuliert werden können. Eine kleine Einführung in die Rechnung mit komplexen Zahlen hilft uns, schicke Fraktale zu zeichnen. Wir werden Farben invertieren, farbige Bilder in Grauwertbilder konvertieren und eine einfache Binarisierungsmethode kennenlernen.

### 12.1 Das RGB-Farbmodell

Bevor wir uns überlegen wie wir ein Bild darstellen, sollten wir uns klarmachen, was eigentlich eine Farbe ist. Das menschliche Auge kann hauptsächlich Licht von drei verschiedenen Wellenlängen wahrnehmen. Diese Wellenlängen werden vom Auge als die Farben Rot, Grün und Blau wahrgenommen. Die Mischung der Intensitäten dieser drei Wellenlängen ermöglicht es, viele verschiedene Farben zu sehen. Der Bau von Monitoren wurde dadurch relativ einfach, da man, um eine bestimmte Farbe zu erzeugen, nur Licht mit drei verschiedenen Wellenlängen zu erzeugen braucht. Die exakte Mischung macht dann eine spezifische Farbe aus.

Dieses Farbmodell wird RGB-Modell genannt, da es auf den drei Grundfarben Rot, Grün und Blau basiert. Schwarz erhält man, wenn keine der drei Farbkanäle Licht liefert. Weiß durch die gleichzeitige, maximale Aktivierung aller drei Grundfarben. Gelb zum Beispiel wird durch eine Mischung aus Rot und Grün erzeugt. Man spricht hierbei von additiver Farbmischung, da die wahrgenommene Farbe immer heller wird, je mehr Grundfarben man hinzunimmt. Die Farbmischung eines Tuschkastens dagegen wird als subtraktiv bezeichnet. Vermischt man – wie Kinder es gerne machen – viele Farben miteinander, erhält man etwas sehr dunkles, meist ein hässliches Braun.

An dieser Stelle wollen wir gleich etwas ausprobieren. Die Funktion `setColor` des **Graphics**-Objekts setzt die Farbe, die zum Zeichnen durch zukünftige Befehle

benutzt werden soll. Der folgende Code erzeugt ein Fenster mit fünf farbigen Rechtecken.

```

1 import java.awt.*;
2
3 public class Bunt extends FensterSchliesstSchickKurz{
4     public Bunt(String title1 , int w, int h){
5         super(title1 , w, h);
6     }
7     public void paint(Graphics g){
8         g.setColor(Color.RED);
9         g.fillRect(1,1,95,100);
10
11        g.setColor(new Color(255,0,0)); // Rot
12        g.fillRect(101,1,95,100);
13
14        g.setColor(new Color(0,255,0)); // Grün
15        g.fillRect(201,1,95,100);
16
17        g.setColor(new Color(0,0,255)); // Blau
18        g.fillRect(301,1,95,100);
19
20        g.setColor(new Color(255,255,0)); // Rot + Grün = ?
21        g.fillRect(401,1,95,100);
22
23        g.setColor(new Color(100,100,100)); // Rot + Grün + Blau = ?
24        g.fillRect(501,1,95,100);
25    }
26
27    public static void main(String[] args){
28        Bunt b = new Bunt ("Farbige Rechtecke", 500, 100);
29    }
30 }
```

Zeile 9 und 12 führen zum gleichen Ergebnis, einem roten Rechteck. Die beiden Anweisungen `g.setColor(Color.RED)` und `g.setColor(new Color(255,0,0))` erzeugen die Farbe Rot. Dem Konstruktor der Klasse **Color** können direkt die drei Farbwerte des RGB-Modells übergeben werden. Dabei steht 0 für keine Intensität und 255 für die maximale Intensität.

Mit diesem Modell können wir somit  $256^3 = 16.777.216$  verschiedene Farben definieren. Wie bereits zuvor angemerkt wurde, führt der Aufruf `g.setColor(new Color(255,255,0))` in Zeile 20 (also die Farben Rot und Grün) zur Mischfarbe Gelb.

Welche Farbmischung ergibt wohl Pink?

Interessant ist noch zu bemerken, dass der letzte Aufruf ein graues Rechteck erzeugt. Alle drei Grundfarben, in gleicher Intensität gemischt, ergeben immer einen Grauwert. Der dunkelste Grauwert ist Schwarz (0,0,0) und der hellste Weiß (255,255,255).

Jetzt sind wir in der Lage, verschiedenfarbige Bilder zu erzeugen. Bisher zeigten die Beispiele farbige Rechtecke. Stellen wir uns diese Rechtecke ganz klein vor, dann brauchen wir sie nur noch sinnvoll anzutragen und schon entsteht ein Bild. Das kleinste darstellbare Rechteck, das wir zeichnen können, ist ein einziger blauer Pixel, also ein Rechteck der Länge und Höhe 1.

```

1 import java.awt.*;
2
3 public class Farbverlauf extends FensterSchliesstSchickKurz{
4     static int fensterBreite = 600;
5     static int fensterHoehe = 300;
6
7     public Farbverlauf(String title1, int w, int h){
8         super(title1, w, h);
9     }
10
11    public void paint(Graphics g){
12        for(double x=1; x<fensterBreite; x++){
13            for(double y=1; y<fensterHoehe; y++){
14                int rot = (int) Math.floor(255*x/fensterBreite);
15                int blau = (int) Math.floor(255*y/fensterHoehe);
16                g.setColor(new Color(rot,0,blau));
17                g.fillRect(x,y,1,1);
18            }
19        }
20    }
21    public static void main(String[] args){
22        Farbverlauf b = new Farbverlauf ("Farbverlauf", fensterBreite,
23                                         fensterHoehe);
24        b.setVisible(true);
25    }
26 }
```

Hier sehen wir, wie einfach es ist, einen seichten Farbverlauf zwischen zwei Farben darzustellen. Die Schleifen in Zeile 11 und 12 gehen über jeden Pixel im Fenster und geben ihm eine Farbe abhängig von seiner Position.

Das führt zu folgender Ausgabe:



## 12.2 Grafische Spielerei: Apfelmännchen

Machen wir das Ganze noch etwas interessanter und erzeugen geometrische Muster, die eine hohe Selbstähnlichkeit aufweisen. Das bedeutet, dass ein Muster aus mehreren verkleinerten Kopien seiner selbst besteht. Wir werden sehen, wie einfach es ist, ein wahrlich künstlerisches Bild zu berechnen. Dazu ist allerdings ein kleiner Ausflug in die Mathematik notwendig. Im Speziellen wird uns eine kleine Einführung in die Komplexen Zahlen helfen, Fraktale zu verstehen und realisieren zu können.

### 12.2.1 Mathematischer Hintergrund

Die Komplexen Zahlen sind eine Menge von Zahlen. Während die uns beispielsweise bekannten Zahlenklassen  $\mathbb{N}$ ,  $\mathbb{Z}$ ,  $\mathbb{Q}$  oder  $\mathbb{R}$  auf einem Zahlenstrahl eingetragen werden können, geht dies mit den Komplexen Zahlen nicht. Eine komplexe Zahl lässt sich als ein Punkt in einem 2-dimensionalen Koordinatensystem vorstellen. Dabei stellt die  $x$ -Achse den bekannten Zahlenstrahl der reellen Zahlen dar. Auf der  $y$ -Achse hingegen können wir noch einmal die gesamten reellen Zahlen abtragen.

Eine komplexe Zahl besteht also aus einem Paar reeller Zahlen. Der Teil einer komplexen Zahl, der auf der  $x$ -Achse abgetragen wird, ist der **Realteil**. Der Wert auf der  $y$ -Achse wird **Imaginärteil** genannt. Dieser wird durch ein ' $i$ ' markiert. Ein Beispiel für eine komplexe Zahl ist:  $1 + 1i$ .

Mit Komplexen Zahlen können wir auch rechnen. Nehmen wir zum Beispiel die Addition

$$(-2 + 3i) + (1 - 2i),$$

so ist das Ergebnis mit einem Zwischenschritt einfach

$$(-2 + 3i) + (1 - 2i) = -2 + 1 + (3 - 2)i = -1 + 1i.$$

Etwas komplizierter ist die Multiplikation. Das Produkt  $A * B$  der beiden komplexen Zahlen  $A = a_1 + a_2 i$  und  $B = b_1 + b_2 i$  ist definiert als

$$a_1 * b_1 - a_2 * b_2 + (a_1 * b_2 + a_2 * b_1)i.$$

Eine Besonderheit sei noch angemerkt, denn es gilt:

$$(0 + 1i) * (0 + 1i) = 1i * 1i = i^2 = -1.$$

Damit hat die Gleichung  $x = \sqrt{-1}$  eine Lösung und zwar die komplexe Zahl  $(0 + 1i)$ .

Da eine komplexe Zahl als ein Punkt in einem 2-dimensionalen Koordinatensystem angesehen werden kann, hat eine solche Zahl auch eine Länge, ihren Abstand vom Ursprung. Anders als bei den reellen Zahlen, bei denen der Abstand vom Ursprung einfach der Betrag der Zahl selbst ist, muss der Abstand einer komplexen Zahl über den Satz des Pythagoras errechnet werden. Ist zum Beispiel  $K$  eine komplexe Zahl mit  $K = 2 + 3i$ , dann ist ihre Länge (auch Norm genannt)  $|K| = \sqrt{2^2 + 3^2} = \sqrt{13}$ .

Die Objektorientierung von Java bietet eine sehr einfache Möglichkeit, das Rechnen mittels komplexer Zahlen zu implementieren. Sehen wir uns ein Beispiel an:

```

1 class KomplexeZahl{
2     double re;      // Speichert den Realteil
3     double im;      // Speichert den Imaginärteil
4
5     public KomplexeZahl(double r, double i){ // Konstruktor
6         re = r;
7         im = i;
8     }
9
10    public KomplexeZahl plus(KomplexeZahl k){
11        return new KomplexeZahl(re + k.re,im + k.im);
12    }
13
14    public KomplexeZahl mal(KomplexeZahl k){
15        return new KomplexeZahl(re*k.re - im*k.im, re*k.im + im*k.re);
16    }
17
18    public double norm(){ // Berechnet den Abstand der Zahl vom Ursprung
19        return Math.sqrt(re*re + im*im);
20    }
21
22    public String text(){ // Gibt die Zahl als Text aus
23        return re+" "+im+" i";
24    }
25}

```

Die Rechnung mit Komplexen Zahlen erweist sich jetzt als genauso einfach, wie die Rechnung mit natürlichen Zahlen:

```

KomplexeZahl zahl1 = new KomplexeZahl(1, 1);
KomplexeZahl zahl2 = new KomplexeZahl(2, -2);
System.out.println("Ergebnis: "+(zahl1.plus(zahl2)).text());

```

Als Ergebnis erhalten wir:

```

C:\Java>java KomplexeZahl
Ergebnis: 3.0 + -1.0 i

```

## 12.2.2 Fraktale

Wir können jetzt mit Komplexen Zahlen rechnen, aber was hat das mit Bildern zu tun? Unsere Motivation ist der Einstieg in die Fraktale.

Komplexe Zahlen eignen sich zur Berechnung von Bildern, da jede Zahl direkt einem Punkt auf einer 2-dimensionalen Fläche zugeordnet werden kann. Berechnen wir also für eine Fläche von Komplexen Zahlen jeweils für jede dieser Zahlen einen Funktionswert, so füllen wir Stück für Stück eine besagte Fläche mit Werten. Jetzt müssen wir jedem Funktionswert nur noch eine Farbe zuordnen und fertig ist das Meisterwerk.

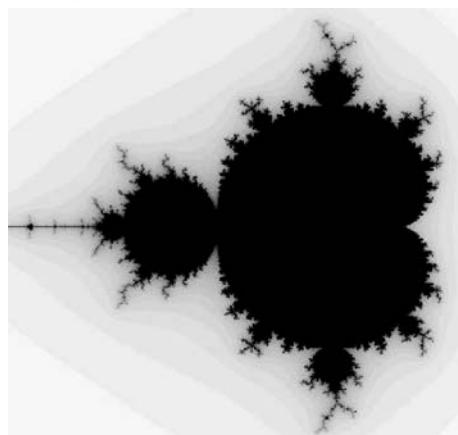
Betrachten wir zunächst folgendes Programm und seine Ausgabe.

### 12.2.2.1 Implementierung eines einfachen Apfelmännchens

```

1 import java.awt.*;
2
3 public class Fraktal extends FensterSchliesstSchickKurz{
4     static int resolution = 500;
5
6     public Fraktal ( String title1 , int w, int h){
7         super(title1 , w, h);
8     }
9
10    public int berechnePunkt(double x, double y){
11        KomplexeZahl c = new KomplexeZahl(x,y);
12        KomplexeZahl z = new KomplexeZahl(0,0);
13        double iter = 0;
14        for (; iter < 40; iter ++){
15            z = (z.mal(z)).plus(c);
16            if(z.norm() > 4) break;
17        }
18        return (int)Math.floor((255)* iter /40);
19    }
20
21    public void berechneBild(Graphics g){
22        for (double x=0; x<aufloesung ;x++){
23            for (double y=0; y<aufloesung ; y++){
24                int Fxy = 255 - berechnePunkt(2.5*x/aufloesung - 1.9 ,
25                                         2.5*y/aufloesung - 1.3);
26                g.setColor(new Color(Fxy, Fxy, Fxy));
27                g.fillRect((int)x, (int)y, 1, 1);
28            }
29        }
30    }
31
32    public void paint(Graphics g){
33        berechneBild(g);
34    }
35
36    public static void main(String[] args){
37        Fraktal f = new Fraktal ("Apfelmännchen", aufloesung , aufloesung );
38        f.setVisible(true);
39    }
40 }
```

Unser Programm erzeugt ein Fraktal mit dem berühmten Namen **Apfelmännchen** und das sieht wie folgt aus:



Der Aufbau ähnelt sehr dem Programm **Farbverlauf.java** aus dem Abschnitt 12.1. Die `paint`-Methode besteht aus zwei `for`-Schleifen, die über alle Pixel des Fensters gehen. Der einzige Unterschied besteht darin, dass die Farbe des Pixels mit der Funktion `berechnePunkt` gesetzt wird, anstelle direkt von `x` und `y` abzuhängen. Die Funktion `berechnePunkt` hat es in sich.

Es ist nicht wichtig, dass wir an dieser Stelle alle mathematischen Einzelheiten der Berechnung verstehen. Soviel sei jedoch gesagt, `berechnePunkt` errechnet die Anzahl an Iterationen, die gebraucht werden, damit die Länge der Komplexen Zahl  $z$  größer als 4 wird. Diese Anzahl an Iterationen hängt von  $x$  und  $y$ , also der Position im Bild, ab. Anschließend wird aus der Anzahl der Iterationen ein Farbintensitätswert berechnet.

Die Variable  $z$  wird mit  $(0 + 0i)$  initialisiert und in jedem Schritt durch  $z = z^2 + c$  ersetzt. Nach dem ersten Schritt ist  $z$  also immer gleich  $c$ , nach dem zweiten Schritt ist  $z = c^2 + c$  usw.  $c$  ist konstant gleich  $x + yi$ . Es ist daher klar, dass diese Iteration für jede Bildposition, die einer Komplexen Zahl mit einer Länge von über 4 entspricht, bereits nach der ersten Iteration abbricht. Diesen Positionen wird ein sehr heller Farbwert zugeordnet. Je länger es dauert bis der Schwellwert überschritten ist, desto dunkler wird der Bildpunkt. Bei schwarzen Bildpunkten wurde der Schwellwert nicht erreicht.

Hätten Sie erwartet, dass ein so kurzes Programm ein solch komplexes Bild erzeugen könnte?

Es handelt sich dabei um ein Apfelmännchen, ein erstmals 1980 von Benoît Mandelbrot berechnetes Fraktal. Wenn Sie mehr über Fraktale wissen wollen, stellt Wikipedia [44] sicher einen guten Ausgangspunkt für Recherchen dar.

Doch unsere Art der Darstellung eines Bildes hat einige Nachteile. Haben Sie das Fenster schon einmal minimiert und dann wieder maximiert oder es teilweise durch ein anderes Fenster verdeckt? Jedes Mal, wenn wir das Bild wieder in den Vordergrund holen, wird es neu berechnet. Das nimmt Zeit in Anspruch. Eine Möglichkeit zur Speicherung unserer Bilder wäre sicherlich auch von Vorteil.

### 12.2.3 Die Klasse `BufferedImage`

Bisher haben wir bunte Rechtecke gezeichnet. Das ist zwar ganz nett und hat uns bereits die ersten schönen Bilder beschert, aber wirklich weit führt uns das noch nicht. Um richtige Bilder mit Java laden, speichern und vor allem verändern zu können, müssen wir uns mit der Klasse **`BufferedImage`** beschäftigen. Erweitern wir die Klasse **`Fraktal`** ein wenig, um die oben angesprochenen Probleme zu beheben.

```

1 import java.awt.*;
2 import java.awt.image.*;
3
4 public class Fraktal extends FensterSchliesstSchickKurz{
5     static int auflösung = 500;
```

```

6  BufferedImage bild; // Hier speichern wir das einmal berechnete Bild
7
8  public Fraktal (String title1 , int w, int h){
9      super(title1 , w, h);
10     bild      = new BufferedImage(aufloesung , aufloesung ,
11                                     BufferedImage.TYPE_INT_RGB);
12     Graphics g = bild.createGraphics();
13     berechneBild(g);
14 }
15
16 // Das Bild muss noch angezeigt werden
17 public void paint(Graphics g){
18     g.drawImage(bild , 5, 31, this);
19 }
20
21 // Um Flackern zu vermeiden (siehe "Ghosttechnik" Tag 10)
22 public void update(Graphics g){
23     paint(g);
24 }
25
26 ...
27 }
```

Die entscheidende Erweiterung erreicht man durch die Verwendung der Klasse **BufferedImage**. Anstatt jedes Mal, wenn das Fenster neu gezeichnet werden muss, auch die Berechnung des Bildes zu wiederholen, berechnen wir das Bild nur ein einziges Mal, z. B. beim Start des Programms im Konstruktor. Sollte später ein Teil oder sogar das ganze Bild erneut gezeichnet werden, können wir einfach das in `bild` gespeicherte Bild anzeigen.

In Zeile 10 wird zuerst die Variable `bild` mit einem leeren Bild initialisiert. `BufferedImage.TYPE_INT_RGB` gibt dabei an, dass es sich um ein Bild im RGB-Format handelt<sup>1</sup>. Anschließend lassen wir uns durch `createGraphics` ein **Graphics**-Objekt von `bild` geben. Anstatt `berechneBild` in der `paint`-Routine mit diesem **Graphics**-Objekt des Fensters zu füttern, zeichnen wir das Bild in das **BufferedImage**. Um das Bild letztendlich anzuzeigen, rufen wir in der `paint`- und in der `update`-Methode (siehe "Ghosttechnik" in Kapitel 10) die Funktion `drawImage` des zum Fenster gehörigen **Graphics**-Objekt auf.

Probieren Sie es aus. Wird das Fenster überdeckt und wieder hervorgeholt, muss es nicht neu berechnet zu werden.

Der Funktion `drawImage` müssen wir zuerst das **BufferedImage**-Objekt übergeben. Danach kommen *x*- und *y*-Wert der Position in Pixeln. Die Koordinaten sind allerdings auf die linke obere Ecke des Fensters bezogen. Hierbei sind leider die Rahmen eingeschlossen, d. h. wir müssen diese beim Zeichnen berücksichtigen. In einer Standard WindowsXP-Applikation ist die Breite des linken Rahmens 5 Pixel und die Höhe des oberen, einschließlich der Titelleiste, 31 Pixel. Um den vierten nötigen Übergabeparameter kümmern wir uns später im Abschnitt 12.2.5. Soviel sei aber gesagt, es ist hier ein **ImageObserver**-Objekt nötig.

---

<sup>1</sup> Eigentlich ist unser Bild s/w, aber das würde die Sache nur unnötig verkomplizieren. Später wollen wir unserem Apfelmännchen noch ein schönes Farbkleid spendieren.

Die Klasse **FensterSchliesstSchickKurz**, von der **Fraktal** abgeleitet ist, ist wiederum von **Frame** abgeleitet. **Frame** selbst implementiert aber das Interface **ImageObserver**. Damit ist auch **Fraktal** ein **ImageObserver**. Der this-Pointer zeigt in diesem Fall auf eine Instanz von **Fraktal**, daher können wir ihn hier als **ImageObserver** benutzen.

Wenn dies nicht auf Anhieb klar ist, besteht kein Grund zu Verzweiflung. Lesen Sie gegebenenfalls den entsprechenden Abschnitt in Kapitel 7 "Erweitertes Klassenkonzept" noch einmal nach.

#### 12.2.4 Bilder laden und speichern

In Kapitel 9 wurde der Einfachheit halber festgelegt, dass wir nur mit AWT arbeiten wollen. Leider bietet aber AWT keine direkte Möglichkeit, ein Bild zu speichern.

Wie man selbst ein Klassenkonzept innerhalb vom AWT entwickelt, das Bilder z. B. im Bitmap-Format speichert, wurde bereits an anderer Stelle zu genüge gezeigt [9, 12]. Dem interessierten Leser sei hier ans Herz gelegt, auch in diese Richtung weiter zu studieren.

Innerhalb vom Package javax existiert allerdings eine sehr einfache Möglichkeit, mittels der Klasse **BufferedImage**, Bilder in den verschiedensten Formaten zu speichern. Wenn Sie ihren Javacompiler, wie anfangs beschrieben, direkt von Sun besorgt haben, sollte folgendes Programm problemlos funktionieren.

```

1 import java.awt.*;
2 import java.awt.image.*;
3 import java.io.File;
4 import java.io.IOException;
5 import javax.imageio.*;
6
7 public class Fraktal extends FensterSchliesstSchickKurz{
8     static int aufloesung = 500;
9     BufferedImage bild;
10
11    public Fraktal (String title1 , int w, int h){
12        super(title1 , w, h);
13        bild      = new BufferedImage(aufloesung , aufloesung ,
14                                     BufferedImage.TYPE_INT_RGB);
15        Graphics g = bild.createGraphics ();
16        berechneBild(g);
17
18        try {
19            ImageIO.write(bild , "JPEG" , new File("Apfelmann.jpg"));
20        } catch(IOException e) {
21            e.printStackTrace();
22        }
23    }
24
25    ...
26 }
```

Hier wird auf die Klasse **ImageIO** aus der Java-Swing-Bibliothek javax zurückgegriffen. Dazu müssen wir die Bibliothek importieren (Zeile 5). Diese bietet, neben

vielen anderen, die Funktion `write`. Dieser Funktion werden ein Format und der Name der Zielfile übergeben. Führen Sie das aktualisierte **Fraktal**-Programm einmal aus und öffnen Sie anschließend das sich nun im gleichen Ordner befindliche **Apfelmann.jpg** mit einem normalen Bildbetrachtungsprogramm.

Es hat funktioniert. Wir können Bilder erzeugen und speichern.

Natürlich kommen wir auch ohne ein externes Bildbetrachtungsprogramm aus, schreiben wir uns einfach schnell selbst eins:

```

1 import java.awt.*;
2 import java.awt.image.*;
3 import java.io.File;
4 import javax.imageio.*;
5
6 public class ZeigeBild extends FensterSchliesstSchickKurz{
7     BufferedImage bild;
8
9     public ZeigeBild (String title1 , String dateiname){
10         super(title1 , 700, 700);
11         try {
12             bild = ImageIO.read(new File(dateiname));
13         }catch(Exception e){
14             System.out.println("Beim Laden ist was schief gelaufen!");
15             System.exit(1);
16         }
17     }
18
19     // Das Bild muss auch angezeigt werden
20     public void paint(Graphics g){
21         g.drawImage(bild , 5, 31, this);
22     }
23
24     // Um Flackern zu vermeiden (siehe "Ghosttechnik" Tag 10)
25     public void update(Graphics g){
26         paint(g);
27     }
28
29     public static void main(String[] args){
30         if (args.length > 0){
31             ZeigeBild f = new ZeigeBild("Bild laden" , args[0]);
32             f.setVisible(true);
33         } else {
34             System.out.println("Bitte Dateinamen angeben!");
35             System.exit(1);
36         }
37     }
38 }
```

Hier wird klar, wie komfortabel die Klasse **ImageIO** ist. Die Programmzeile

```
bild = ImageIO.read(new File(dateiname));
```

genügt, um das Bild einzulesen. Wir müssen uns nicht einmal um das Format kümmern. Das Anzeigen des **BufferedImage** wird wie gewohnt erledigt. Dem Programm sollte natürlich ein gültiger Dateiname beim Aufruf übergeben werden.

Das Handwerkzeug im Umgang mit Bildern, also Laden, Speichern und Erzeugen von Bildern, haben wir erworben. Bevor wir jedoch einen Schritt weiter gehen und Bilder bearbeiten, färben wir zunächst unser Apfelmännchen bunt ein. Das ist leider nicht ganz einfach, dafür werden wir aber anschließend mit noch schöneren Bildern belohnt.

Erinnern wir uns noch einmal an den grundlegenden Mechanismus. Für jedes Bildpixel erzeugen wir eine komplexe Zahl  $c$ . Eine solche komplexe Zahl kann als Punkt in einer 2-dimensionalen Ebene gesehen werden. Nun starten wir eine Iteration:

$$z_{\text{neu}} = z_{\text{alt}}^2 + c,$$

die nach einer bestimmten Anzahl von Iterationen (abhängig von  $c$  und damit von der Pixelposition) eine komplexe Zahl  $z$  erzeugt, deren Länge (Abstand vom Nullpunkt) einen gewissen Schwellwert überschreitet.

Wir wollen die Anzahl an benötigten Iterationsschritten dazu benutzen, den entsprechenden Bildpunkt einzufärben und nicht nur seine Helligkeit zu bestimmen. Dazu definieren wir Intervalle von Iterationszahlen, die in einer bestimmten Farbe eingefärbt werden sollen.

```

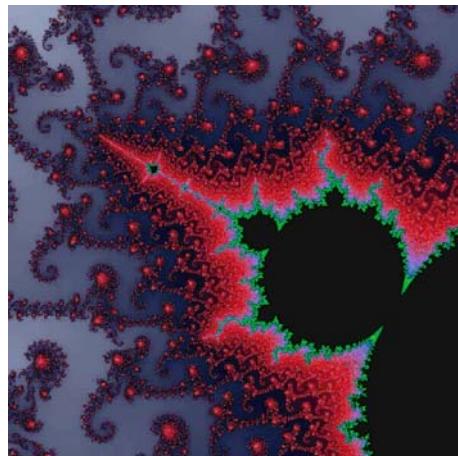
1 public class FraktalBuntFinal extends FensterSchliesstSchickKurz {
2     static int aufloesung      = 100;
3     static int fensterRandLRU = 5; // Fensterrand links , rechts , unten
4     static int fensterRandO   = 31; // Fensterrand oben
5     static int itermax        = 2000;// Maximale Anzahl von Iterationen
6     static int schwellenwert  = 35; // bis Erreichen des Schwellwerts .
7     static int [][] farben   = {
8         { 1, 255,255,255}, // Hohe Iterationszahlen sollen hell ,
9         { 300, 10, 10, 40}, // die etwas niedrigeren dunkel ,
10        { 500, 205, 60, 40}, // die "Spiralen" rot
11        { 850, 120,140,255}, // und die "Arme" hellblau werden .
12        {1000, 50, 30,255}, // Innen kommt ein dunkleres Blau ,
13        {1100, 0, 0,255}, // dann grelles Grün
14        {1997, 20, 70, 20}, // und ein dunkleres Grün .
15        {itermax , 0, 0, 0}}; // Der Apfelmännchen wird schwarz .
16
17     static double bildBreite  = 0.000003628;
18     // Der Ausschnitt wird auf 3:4 verzerrt
19     static double bildHoehe   = bildBreite *3.f/4.f;
20     // Die Position in der Komplexen-Zahlen-Ebene
21     static double [] bildPos  = {-0.743643135-(2*bildBreite/2),
22                               0.131825963-(2*bildBreite*3.f/8.f)} ;
23
24     BufferedImage bild;
25
26     public FraktalBuntFinal (String title1 , int w, int h){
27         super(title1 , w, h);
28
29         bild = new BufferedImage(aufloesung , aufloesung ,
30                                BufferedImage.TYPE_INT_RGB);
31         Graphics gimg = bild.createGraphics ();
32
33         berechneBild(gimg);
34         try {
35             ImageIO.write(bild , "BMP" , new File ("ApfelmannBunt.bmp"));

```

```

36         } catch(IOException e){
37             System.out.println("Fehler beim Speichern!");
38         }
39     }
40
41     public Color berechneFarbe(int iter){
42         int F[] = new int[3];
43         for (int i=1; i<farben.length-1; i++){
44             if (iter < farben[i][0]){
45                 int iterationsInterval = farben[i-1][0]-farben[i][0];
46                 double gewichtetesMittel = (iter-farben[i][0])/
47                     (double)iterationsInterval;
48
49                 for (int f=0; f<3; f++){
50                     int farbInterval = farben[i-1][f+1]-farben[i][f+1];
51                     F[f] = (int)(gewichtetesMittel*farbInterval)
52                         +farben[i][f+1];
53                 }
54
55             return new Color(F[0], F[1], F[2]);
56         }
57     }
58     return Color.BLACK;
59 }
60
61     public int berechnePunkt(KomplexeZahl c){
62         KomplexeZahl z = new KomplexeZahl(0,0);
63         int iter = 0;
64         for (; (iter <= itermax) && (z.norm() < schwellenwert); iter++)
65             z = (z.mal(z)).plus(c);
66         return iter;
67     }
68
69     public void berechneBild(Graphics g){
70         for (int x=0; x<aufloesung; x++){
71             for (int y=0; y<aufloesung; y++){
72                 KomplexeZahl c = new KomplexeZahl(bildBreite*(double)(x)/
73                     aufloesung + bildPos[0], bildBreite*(double)(y)/
74                     aufloesung + bildPos[1]);
75
76                 g.setColor(berechneFarbe(berechnePunkt(c)));
77                 g.fillRect(x, y, 1, 1);
78             }
79         }
80     }
81
82     public void paint(Graphics g){
83         g.drawImage(bild, fensterRandLRU, fensterRandO, this);
84     }
85
86     public void update(Graphics g){
87         paint(g);
88     }
89
90     public static void main(String[] args){
91         FraktalBuntFinal f = new FraktalBuntFinal("Apfelmännchen - Bunt",
92             aufloesung+2*fensterRandLRU,
93             aufloesung+fensterRandLRU+fensterRandO);
94         f.setVisible(true);
95     }
96 }
```

Wir erhalten jetzt das Apfelmännchen in voller Farbenpracht:



Diese Version des Programms hat zwei Vorteile gegenüber der vorherigen.

Erstens sind die Konstanten, die den zu berechnenden Bereich in der Komplexen Zahlen-Ebene festlegen, nicht mehr irgendwo im Programm versteckt, sondern leicht am Anfang der Klasse durch `bildBreite`, `bildHoehe` und `bildPos` zu sehen. Das Gleiche gilt für die maximale Iterationsanzahl, den Schwellwert zum Abbruch der Iteration und die Breite des Fensterrands. Es ist ein guter Programmierstil keine "magic numbers" – also unbenannte Konstanten, die einfach irgendwo auftauchen – zu verwenden.

Der zweite Vorteil ist die einfache Möglichkeit, das Fraktal nach Lust und Laune einzufärben. Dies bewerkstelligt die Funktion `berechneFarbe`. Für eine gegebene Iterationszahl `iter` sucht sie die oberste Zeile in der Matrix `farben`, deren erster Wert größer als `iter` ist. Die drei weiteren Werte der gerade gefundenen Zeile entsprechen dann den RGB-Werten. Diese Art der Einfärbung würde zu sehr großen gleichfarbigen Zonen führen, da viele Iterationswerte die gleiche Farbe bekämen. Daher ermitteln wir ein gewichteten Mittelwert der Farben aus der aktuellen und der vorherigen Zeile.

Es ist theoretisch möglich, unendlich "tief" in das Apfelmännchen hineinzuzoomen. Irgendwann spielt allerdings der Darstellungsbereich eines `double` nicht mehr mit. Die Zahlen werden dann so ähnlich, da der Ausschnitt in der Komplexen Zahlen Ebene so extrem klein wird, dass ein `double` den Unterschied nicht mehr darstellen kann. Man kann sich hier retten, indem man Datentypen mit größerem Speicher verwendet, aber irgendwann stößt man immer an die Grenzen der Berechenbarkeit.

Wenn Sie etwas mit dem Apfelmännchen herumgespielt haben, dann werden Sie bemerkt haben, wie viel Zeit man darauf verwenden kann, schöne Bilder zu erstellen. Zu diesem Thema sei empfohlen, einmal eine eigene Internetrecherche durchzuführen. Es gibt bereits ganze Galerien wahrlich künstlerischer Fraktalbilder.

Lassen Sie sich von folgender Webseite einen Einblick in die Vielfältigkeit geben:

<http://fraktalbild.de/>

### 12.2.5 Bilder bearbeiten

Bisher haben wir die Klasse **BufferedImage** als eine Art Zwischenspeicher für Bilder verwendet. Das ist sie auch. Es eröffnen sich allerdings neue Möglichkeiten, wenn man sie in einem größeren Verbund von zusammenspielenden Klassen sieht. Dieser Verbund wird gebildet aus **ImageProducer** (Bilderzeuger), **ImageFilter** (Bildfilter), **ImageObserver** (Bildbeobachter) und **ImageConsumer** (Bildverbraucher). Er bietet eine Umgebung, mit Bildern umzugehen und verschiedene Operationen auf ihnen auszuführen. Wir verzichten hier bewusst darauf, dies weiter zu erklären oder zu benutzen. Möchte man mit Bildern wissenschaftlich arbeiten, z. B. für Handschrift- oder Gesichtererkennung, ist es allemal besser, ein Bild als Matrix oder Vektor (siehe auch Kapitel 13) zu repräsentieren und es nur für Anzeigezwecke in ein **Image**-Objekt umzuwandeln.

#### 12.2.5.1 Ein Bild invertieren

Als erstes wollen wir ein Bild invertieren. Dies bedeutet, jeden Bildpixel mit seinem farblichen Gegenspieler einzufärben. Benutzen wir die gewohnte RGB-Darstellung, kann eine Invertierung leicht durchgeführt werden. Ist der Helligkeitswert einer der Grundfarben  $x$ , so ist sein invertierter Helligkeitswert  $255 - x$ . Aus 0 wird also 255 und aus 255 wird 0. 127 wird zu 128, ein mittelhelles Grau bleibt demnach fast unverändert.

Die Sache wird ein wenig komplizierter, da die drei Farbkanäle gemeinsam in einem Integer gespeichert werden. Dabei geht man wie folgt vor: Sind  $r$ ,  $g$  und  $b$  die Helligkeitswerte der drei Grundfarben, jeweils im Bereich von 0 bis 255, dann ergibt sich der gemeinsame RGB-Wert aus:

$$rgb = 256 * 256 * r + 256 * g + b.$$

Das Schöne an dieser Art der Darstellung ist, dass alle drei Farben speicherplatzgünstig in einem einzigen int gespeichert und aus diesem wieder eindeutig rekonstruiert werden können. Ein int in Java verfügt über acht Byte. Hier sind die ersten beiden Bytes für den Alpha-Wert (Transparenz) der Zahl reserviert, der in der reinen RGB-Darstellung keine Rolle spielt. Die restlichen drei Byte kodieren jeweils einen Farbkanal. Um an den konkreten Wert nur eines Bytes aus dem int zu kommen, setzen wir die anderen auf 0 und verschieben das entsprechende Byte an die niederwertigste Stelle. Dazu benutzen wir das bitweise UND, das die UND-Funktion elementweise auf zwei Bitworte anwendet. Die einzelnen Farbkanäle lassen sich dann durch

```
int rot    = (farbe & 256*256*255) / (256*256);
int gruen = (farbe & 256*255) / 256
int blau  = (farbe & 255)
```

extrahieren.

```

public BufferedImage invertiere(BufferedImage b){
    int x = b.getWidth();
    int y = b.getHeight();
    BufferedImage ib = new BufferedImage(x,y,BufferedImage.TYPE_INT_RGB);
    for (int i=0; i<x; i++){
        for (int k=0; k<y; k++){
            // 255 + 256*255 + 256*256*255
            int neu = 255 + 256*255 + 256*256*255 - b.getRGB(i,k);
            ib.setRGB(i,k,neu);
        }
    }
    return ib;
}

```

Die Funktion `invertiere` kann leicht in das `ZeigeBild`-Programm eingebaut werden. Fügen Sie als letzte Zeile des Konstruktors einfach `bild = invertiere(bild)` an und laden Sie ein Bild. `invertiere` erzeugt ein neues Bild und lässt das alte unverändert. Die beiden Schleifen laufen über jeden Pixel des alten Bildes und schreiben den invertierten RGB-Wert in das neue Bild. Wir greifen hier nicht mehr auf die Funktion `fillrect` des **Graphics**-Objekts zurück, sondern setzen gezielt einen Pixel. Hierbei können wir aber nur einen `int` als Farbwert und nicht ein **Color**-Objekt angeben.



### 12.2.5.2 Erstellung eines Grauwertbildes

Das gleiche Prinzip können wir nutzen, um aus einem Farbbild ein Graubild zu erstellen. Wir laufen wieder über jeden Bildpunkt, berechnen einen durchschnittlichen Helligkeitswert aus den drei Kanälen und schreiben diesen in die drei Kanäle des Pixels im neuen Bild. Dazu ist es allerdings nötig, zuerst den Farbwert jedes einzelnen Kanals zu bestimmen. Das lässt sich realisieren, indem wir die beiden Zeilen innerhalb der `for`-Schleife von `invertiere` durch folgende ersetzen:

```

int alt      = b.getRGB(i,k);
int rot      = (alt & 256*256*255)/(256*256);
int gruen    = (alt & 256*255)/256;
int blau     = (alt & 255);
int grauwert = (int) Math.floor(0.299*rot + 0.587*gruen + 0.114*blau);
int neu      = 256*256*grauwert + 256*grauwert + grauwert;
ib.setRGB(i,k,neu);

```

Eine Besonderheit ist hierbei, dass wir anstelle des einfachen arithmetischen Mittelwerts der Helligkeiten einen gewichteten Mittelwert berechnen. Dies ist durch die Biologie des menschlichen Auges motiviert. Grüne Farbanteile im Licht werden stärker wahrgenommen als rote und diese wiederum stärker als blaue.



### 12.2.5.3 Binarisierung eines Grauwertbildes

Den Grauwert kann man auch verwenden, um das Bild zu binarisieren. Dies bedeutet, man entscheidet für jeden Pixel, ob er entweder schwarz oder weiß sein soll, abhängig von der Intensität des Grauwerts.

```

int alt = b.getRGB(i,k);
int rot = (alt & 256*256*255)/(256*256);
int gruen = (alt & 256*255)/256;
int blau = (alt & 255);
int grauwert = (int) Math.floor(0.299*rot + 0.587*gruen + 0.114*blau);
if (grauwert > 125)
    ib.setRGB(i,k,256*256*255 + 256*255 + 255);
else
    ib.setRGB(i,k,0);

```

Unsere Binarisierungsmethode setzt Pixel mit einem Grauwert grau größer als 125 auf Weiss und alle anderen auf Schwarz. Da alle Bildpunkte mit dem gleichen Schwellwert binarisiert werden, nennt man dieses Verfahren auch **globale Binarisierung**. Bessere Verfahren (z. B. [31]) ermitteln die Schwellwerte für kleinere Flächen, diese bezeichnet man als **lokale Binarisierung**.



## 12.3 Zusammenfassung und Aufgaben

### Zusammenfassung

Das RGB-Farbmodell ist ein hilfreiches Konzept für die Darstellung von Farben im Rechner. Wir können jetzt Farben gezielt erzeugen und anzeigen. Mit dem Apfelmännchen sind wir, nach einer kurzen Wiederholung der Komplexen Zahlen, in die Berechnung von Fraktalen eingestiegen und können diese jetzt sogar speichern und wieder laden. Die Bildverarbeitungstechniken Invertierung, Grauwertbilderzeugung und Binarisierung haben wir ebenfalls kennengelernt.

### Aufgaben

Übung 1) Erweitern Sie das Programm ZeigeBild so, dass das Fenster seine Größe dem geladenen Bild anpasst (die Methoden `getWidth` und `getHeight` der Klasse **BufferedImage** könnten sich als nützlich erweisen...).

Übung 2) Erweitern Sie das Fraktal-Programm so, dass es dem Benutzer gestattet ist, den Bildausschnitt selbst zu wählen. Hier wäre auch eine Zoomfunktion denkbar, die auf Mauseingaben reagiert.

# 13

---

## Tag 13: Methoden der Künstlichen Intelligenz

Das Gebiet der Künstlichen Intelligenz ist zu umfassend, als auch nur einen Überblick in diesem Kapitel geben zu können. Daher wurden ein paar interessante Aspekte – wie ich finde – herausgepickt und exemplarisch erläutert. Wir werden sehen, dass unser bisheriges Wissen über Java ausreicht, um in diese Materie einzusteigen und Probleme der Künstlichen Intelligenz lösen zu können.

Zunächst werden wir besprechen, wie Bilder von handgeschriebenen Ziffern mittels zweier Algorithmen von einem Programm richtig erkannt werden können. Wir wollen also die Bilder den richtigen Zahlen zuordnen, d. h. sie klassifizieren. Zu guter Letzt spielen wir ein bisschen TicTacToe und werden sehen, wie ein Programm in der Lage ist, den besten Zug in einem Spiel zu finden.

### 13.1 Mustererkennung

Mustererkennung ist eines von vielen Gebieten der Künstlichen Intelligenz. Wie es der Name andeutet, beschäftigt man sich hier mit der Erkennung und Klassifizierung von Mustern. Seien es Bilder (Gesichter, Buchstaben, Straßenschilder, ...) oder beispielsweise Sprachaufnahmen. Es kann die Aufgabe gestellt sein, ein Muster zu identifizieren, das in einer etwas veränderter Form vorliegt oder verrauscht<sup>1</sup> ist. Für den Menschen ist die Erkennung von Mustern relativ leicht, für Computer hingegen ist es ein schwieriges Problem.

In den meisten Fällen gibt es viele Vorverarbeitungsschritte, um die Daten für eine Klassifizierung vorzubereiten. Wir wollen uns das hier ersparen und verwenden einfach vorverarbeitete  $12 \times 16$  Pixel große Bilder von Ziffern. Es ist bekannt, welche Zahl jedes Bild repräsentiert. In diesem Fall sprechen wir von einem Etikett oder

---

<sup>1</sup> Mit Verrauschen sind beispielsweise falsche Bildpunkte in einem Bild gemeint. Da man Bilder einfach als eine Menge von Daten ansehen kann, bedeutet Rauschen also eine gewisse Menge von Datenelementen, die falsch gesetzt bzw. gestört sind.

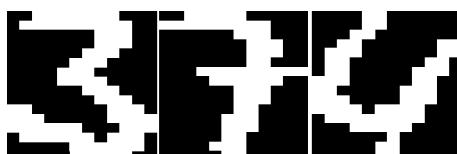
einem Label auf dem z. B. steht „ich bin eine 4“. Diese Daten sind unsere Trainingsdaten, wir vertrauen ihnen und versuchen anhand dieser Informationen unbekannte Zahlen (ohne Label) zu identifizieren.

## *Einlesen der Trainingsdaten*

1000 dieser Bilder liegen in einer Datei (Kodierung siehe folgende Abbildung) als  $12 \times 16$  Matrizen vor, jeder Bildpunkt liegt im Bereich von  $[0.0, 0.1, \dots, 1.0]$ . Unter den Bildmatrizen befindet sich das kodierte Label, wobei jeweils nur einer der zehn Parameter auf 1.0 gesetzt ist (Index  $k$ ) und alle anderen auf 0.0. Der Index  $k$  verrät, dass es sich dabei um die Zahl  $k - 1$  handelt. Ist also der 4te Eintrag mit einer 1.0 versehen, so handelt es sich um die Zahl 3.

Die Bilddaten liegen als Pixelwerte im Format  $12 \times 16$  vor. Unter jedem Bild ist die Etikette kodiert.

Wenn man sich für die verschiedenen Zahlen jetzt Grauwerte ausgeben lässt und diese etwas vergrößert, dann können wir die Zahl sofort erkennen und klassifizieren:



Drei Beispiele für die Zahlenbilder in unserer Trainingsdatei. Ein Mensch hat keinerlei Schwierigkeiten, sie dem richtigen Zahlenwert zuzuordnen.

Unsere Aufgabe wird es nun sein, das dem Computer beizubringen.

Die hier verwendeten Daten sind auf der Homepage verlinkt. Informationen dazu sind im Vorwort zu finden.

Nachdem die Daten eingelesen wurden, werden wir zwei Methoden zur Klassifizierung dieser Daten kennenlernen. Vorher müssen wir die Daten nur noch geeignet repräsentieren und das geht am besten durch einen Vektor. Dabei werden die Zeilen der Matrix einfach aneinandergehängt, so dass wir einen 192-dimensionalen Vektor ( $12 * 16 = 192$ ) für ein Bild erhalten.

Als Datentypen verwenden wir jetzt die beiden folgenden:

```
public double[][] trainingsdaten;
public int[] label;
```

In die Liste `trainingsdaten` werden die 192-dimensionalen Vektoren eingelesen, in `label` die entsprechenden Label. Die Einlesemethode könnte dann wie folgt aussehen:

```
public boolean readData(String filename){
    String filenameIn = filename, sLine;
    StringTokenizer st;

    try {
        FileInputStream fis      = new FileInputStream(filenameIn) ;
        InputStreamReader isr     = new InputStreamReader(fis);
        BufferedReader bur       = new BufferedReader(isr);

        int zaehler = 0;
        while((sLine = bur.readLine()) != null) {
            // lese das Ziffernbild ein
            int count = 0;
            for (int i=0; i < 16; i++) {
                // Wir zerlegen die Zeile in Ihre Bestandteile:
                st = new StringTokenizer(sLine, " ");
                for (int j=0; j < 12; j++)
                    trainingsdaten[zaehler][count++] =
                        Double.parseDouble(st.nextToken());
                sLine = bur.readLine();
            }

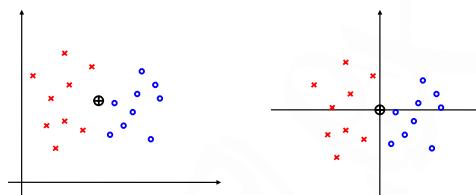
            // dekodierte das Label
            int lab = -1; // es könnte sich um einen Ausreißer handeln
            st = new StringTokenizer(sLine, " ");
            for (int i=0; i<10; i++) {
                if(Double.parseDouble(st.nextToken()) == 1.0) {
                    lab = i;
                    break;
                }
            }
            label[zaehler] = lab;
            sLine = bur.readLine();
            zaehler++;
        }
    } catch(ArrayIndexOutOfBoundsException eAIOOB) {
        System.out.println("Es gab einen Indexfehler.");
    } catch(IOException eIO) {
        System.out.println("Konnte Datei "+filenameIn+" nicht öffnen!");
    }
    return true;
}
```

Da die Zifferndatenbank von Comay [32] auch Ausreißer (Outlier) beinhaltet, also Bilder, die keine Zahlen darstellen, müssen wir diese besonders behandeln. Ausreißer erhalten im obigen Programm (Zeilen 24–32) als Label eine  $-1$ , da bei ihnen alle Elemente der Zeile unter der Bildmatrix nur Nullen enthalten.

Bevor wir mit den Klassifizierungsalgorithmen beginnen, werden wir die Daten zentrieren. Das hat den Vorteil, dass die Daten in der Nachbarschaft des Ursprungs liegen und wir diese Eigenschaft später ausnutzen können. Das bedeutet, dass wir den Schwerpunkt<sup>2</sup>  $\vec{\mu}$  der Trainingsdaten ermitteln und ihn anschließend so verschieben, dass er dem Ursprung der Trainingsvektoren  $\{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m\}$  entspricht:

$$\vec{x}_i := (\vec{x}_i - \vec{\mu}), \forall i, \text{ mit } i = 1, \dots, m$$

Bildlich kann man sich das in etwa so vorstellen, dass der Datenschwerpunkt auf den Ursprung gezogen wird:



Die programmiertechnische Umsetzung ist sehr einfach:

```
public void zentrierung(){
    // Ermittle das Zentrum der Daten
    int m[] = new int[192];
    for (int j=0; j<192; j++){
        for (int i=0; i<1000; i++)
            m[j]+=trainingsdaten[i][j];
        m[j]/=1000;
    }

    // Zentriere die Daten
    for (int i=0; i<1000; i++)
        for (int j=0; j<192; j++)
            trainingsdaten[i][j]-=m[j];
}
```

### 13.1.1 k-nn

Der naheliegendste Klassifizierungs-Algorithmus und gleichzeitig auch der mit der besten Erkennungsrate ist der *k-nächste Nachbarn* Algorithmus oder kurz *k-nn*. Wir haben in unserem Beispiel die  $m = 1000$  verschiedenen 192-dimensionalen Da-

---

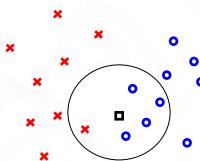
<sup>2</sup> In diesem Fall entspricht der Schwerpunkt dem Mittelwert.

ten  $\{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m\}$  mit ihren entsprechenden Etiketten  $\{l_1, l_2, \dots, l_m\}$  der Trainingsmenge erfasst. Die Label zusammen mit den Datenvektoren ergeben eine Menge von Tupeln  $\{(\vec{x}_1, l_1), (\vec{x}_2, l_2), \dots, (\vec{x}_m, l_m)\}$ . Diese Menge entspricht unserer Referenzmenge, diesen Daten vertrauen wir<sup>3</sup>.

Nun kommt ein neuer Vektor  $\vec{y}$ , z. B. eine Ziffer einer Postleitzahl oder eine Eingabe des Nutzers mit der Maus, und wir wollen feststellen, mit welchem Label er versehen werden soll, also welche Ziffer dargestellt ist. Zunächst berechnen wir die Abstände, z. B. den Euklidischen Abstand zwischen  $\vec{y}$  und allen  $\vec{x}_i$ , mit  $i = 1, \dots, m$  aus der Trainingsmenge und wählen das Label, welches am häufigsten unter den  $k$  nächsten Nachbarn von  $\vec{y}$  vorkommt. Um vielen Pattsituationen aus dem Weg zu gehen, sollte  $k$  eine kleine, ungerade Zahl sein, z. B. 3 oder 5.

### 13.1.1.1 Visualisierung

Da eine anschauliche Visualisierung eines 192-dimensionalen Raumes kaum möglich ist, wählen wir einfacheheitshalber zwei Dimensionen. Die Vorgehensweise des Algorithmus lässt sich sehr schön in einem Bild zeigen:



Das *Viereck* entspricht dem unbekannten Datenvektor  $\vec{y}$ . Der Abstand zu allen Trainingsvektoren mit den Labels *Kreuz* und *Kreis* wurde ermittelt. Unter den  $k = 5$  nächsten Nachbarn sind vier mal *Kreis* und einmal *Kreuz* vertreten. Der neue Vektor  $\vec{y}$  wird also als *Kreis* klassifiziert.

So könnte die Klassifizierungsmethode für  $k$ -nn aussehen:

```
public int classify(double[] y, int k){
    int returnLabel;
    int freq[] = new int[10];
    int lab[] = new int[k];
    double dist[] = new double[k];
    double d;

    // Abstände auf unendlich setzen
    for (int i=0; i<k; i++)
        dist[i] = Double.POSITIVE_INFINITY;

    // Abstand von y zu allen Vektoren der Trainingsmenge berechnen
    // und die k-nächsten Nachbarn speichern
```

<sup>3</sup> In diesem Fall ist das besonders wichtig, da das verwendete Klassifizierungsverfahren anfällig gegenüber Rauschen ist und die Trainingsdaten deshalb möglichst gut sein sollten.

```

for (int i=0; i<1000; i++) {
    d = 0;
    // Abstand berechnen
    for (int j=0; j<192; j++)
        d += (y[j]-trainingsdaten[i][j]) * (y[j]-trainingsdaten[i][j]);
    if (d > dist[k-1])
        continue;
    // die Entfernungen werden mit InsertionSort sortiert
    for (int j=0; j<k; j++) {
        if (d<dist[j]) {
            for (int h=k-1; h>j; h--) {
                dist[h] = dist[h-1];
                lab[h] = lab[h-1];
            }
            dist[j] = d;
            lab[j] = label[i];
            break;
        }
    }
}

// nun wird demokratisch entschieden
for (int i=0; i<k; i++)
    if (lab[i]!=-1)
        freq[lab[i]]++;
returnLabel = -1;
int f = -1;
for (int i=0; i<10; i++) {
    if (f<freq[i]) {
        f = freq[i];
        returnLabel = i;
    }
}
return returnLabel;
}

```

Nehmen wir an, die Klasse **Knn** enthält die Einlese- und Klassifizierungsmethoden. Als kleine Übung können wir jetzt feststellen, wie hoch die Erkennungsrate ist, wenn wir unsere ursprünglichen Trainingsdaten mit sich selbst vergleichen. Die Klassifikation wird in einer Testmethode ausgeführt.

```

public static void main(String [] args){
    Knn classifier = new Knn();
    classifier.readData("digits-training.txt");
    classifier.zentrierung();

    // Teste Trainingsdaten auf sich selbst und ermittle
    // die Erkennungsrate
    double er = 0;
    for (int i=0; i<1000; i++)
        if (classifier.label[i] ==
            classifier.classify(classifier.trainingsdaten[i], 1))
            er++;
    er/=1000;
    System.out.println("Erkennungsrate = "+er);
}

```

Bei der Ausführung erhalten wir:

```
C:\JavaCode>java Knn
Erkennungsrate = 0.993
```

Je mehr Datenpunkte (also richtig gelabelte Bilder von handgeschriebenen Ziffern) in der Trainingsdatenbank vorliegen, je dichter also der zu untersuchende Raum gefüllt ist, desto besser arbeitet der Algorithmus. Problematisch ist dabei aber, dass mit jedem weiteren Datenpunkt der Algorithmus langsamer wird. Die meisten Anwendungen mit integrierter Ziffernerkennung sind zeitkritisch. Beispielsweise braucht die Post ein sehr schnelles Verfahren für die automatische Erkennung von Postleitzahlen. Daher werden in der Praxis oft andere Algorithmen eingesetzt.

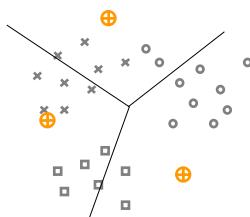
### 13.1.2 $k$ -means

Anders als der  *$k$ -nächste Nachbarn*-Algorithmus wählt  $k$ -means in einem Lernprozess  $k$  Vektoren (Prototypen) als Repräsentanten der Trainingsdaten aus. Diese Vektoren  $\vec{p}_1, \vec{p}_2, \dots, \vec{p}_k$  sitzen im besten Fall in der Mitte von Punktanhäufungen mit dem gleichen Label. Eine solche Punktwolke wird nun durch diesen einen Vektor repräsentiert. Das Verfahren führt also eine Umkodierung und eine Komprimierung der Daten durch. Dabei spielen die Label der Daten keine Rolle, es wird lediglich die Verteilung der Daten im Raum untersucht und in  $k$  verschiedene Cluster  $C_i$  mit  $i = 1, \dots, k$  unterteilt. Lokale Anhäufungen werden durch wenige Prototypen repräsentiert. Da wir in diesem Fall die ursprünglichen Label der Datenpunkte nicht direkt zur Klassifizierung verwenden, gehört dieses Verfahren zur Klasse der **unüberwachten Lernalgorithmen** und ist ein klassisches Clusteringverfahren.

Für unser Ziffernbeispiel können wir davon ausgehen, dass sich gleiche Zahlen im 192-dimensionalen Raum in Gruppen zusammenfinden. Daher werden wir zunächst die Daten mit  $k$ -means umrepräsentieren und anschließend die Label der Prototypen mittels Mehrheitsentscheid der umliegenden Datenpunkte wählen.

Wenn ein Vektor  $\vec{y}$  klassifiziert werden soll, greifen wir auf den Algorithmus  $k$ -nn mit nur einem Nachbarn zurück. Es gewinnt also der Vektor  $\vec{p}_k$ , der  $\vec{y}$  am nächsten liegt. Das hat den großen Vorteil, dass wir zur späteren Klassifizierung unbekannter Vektoren nicht mehr auf die viel umfangreichere Trainingsmenge zurückgreifen müssen.

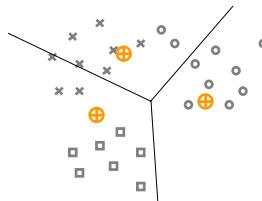
Bleibt nur noch das Problem, die  $k$  richtigen Vektoren für die Prototypen auszuwählen. Der Lernprozess arbeitet wie folgt:



$k$  zufällige Prototypen, in diesem Beispiel sind es drei, werden ausgewählt. Die Trainingsvektoren werden jeweils dem Prototypen zugeordnet, der ihnen am nächsten

liegt. Als Abstandsfunktion zwischen zwei Vektoren kann hier der Euklidische Abstand berechnet werden. Die Linien stellen die Trennung der Daten dar. Diese Phase der Zuordnung nennen wir **Expectation-Schritt**.

Wenn alle Trainingsvektoren den Prototypen zugeordnet wurden, setzen wir die Prototypen neu. Sie sollen ihre Trainingsvektoren besser repräsentieren. Diesen Schritt nennen wir **Maximization-Schritt**.

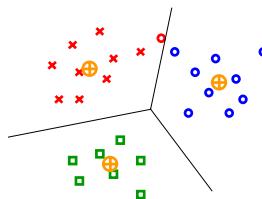


Die Prototypen haben einen Sprung in Richtung der Clusterzentren gemacht. Das ganze wird solange wiederholt, Expectation und Maximization im Wechsel, bis die Prototypen sich nicht mehr viel bewegen, oder eine vorher bestimmte Anzahl von Iterationsschritten erreicht wurde.

Am Ende des Lernprozesses erwarten wir für unser Beispiel z. B. folgende Prototypen:



Jetzt können wir entscheiden, welche Klasse der Daten die jeweiligen Prototypen repräsentieren sollen. Dazu führen wir wieder einen Mehrheitsentscheid durch.



In diesem Beispiel konnten alle Trainingsvektoren bis auf einen, korrekt durch die gewählte Verteilung der Prototypen repräsentiert werden.

### 13.1.2.1 Expectation-Maximization als Optimierungsverfahren

Wir können diese Optimierungsmethode etwas präziser wie folgt beschreiben.  $k$ -means versucht die Funktion  $f(\vec{p}_1, \dots, \vec{p}_k)$  mit

$$f(\vec{p}_1, \dots, \vec{p}_k) = \sum_{i=1}^k \sum_{j=1}^m z_{ij} dist(\vec{p}_i, \vec{x}_j)$$

zu minimieren, wobei

$$z_{ij} = \begin{cases} 1, & \text{falls } x_j \text{ zu } C_i \text{ gehört} \\ 0, & \text{sonst} \end{cases}$$

ist. Um dieses Problem zu optimieren, werden immer im Wechsel die beiden Schritte  
a) Prototypen festhalten (**Maximization-Schritt**) und die Zugehörigkeit aktualisieren und b) Zugehörigkeiten festhalten und die Positionen der Prototypen optimieren (**Expectation-Schritt**) durchgeführt, bis ein Abbruchkriterium erfüllt ist.

Für einen professionellen Einstieg in die Künstliche Intelligenz empfehle ich „den Duda“ [24].

### 13.1.2.2 Allgemeine Formulierung des $k$ -means Algorithmus

initialisiere die  $k$  Prototypen zufällig im Zieldatenraum

Expectation: ordne alle Vektoren der Trainingsmenge  $\{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m\}$  ihren nächstgelegenen Prototypen  $\vec{p}_i$  zu, mit  
 $\vec{x}_i \in C_j$ , wenn  $d(\vec{x}_i, \vec{p}_j) \leq d(\vec{x}_i, \vec{p}_r)$  für  $r = 1, \dots, k$

Maximization: setze die Prototypen auf das jeweilige Zentrum der  $n_i$  Trainingsvektoren, die zu diesem Prototypen, also zum Cluster  $C_i$ , zugeordnet wurden  
 $\vec{p}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \vec{x}_j$

Gehe zum Expectation-Schritt bis Abbruchkriterium erfüllt ist

Das einfachste Abbruchkriterium wäre ein Zähler. Wir stoppen nach einer bestimmten Anzahl von Expectation-Maximization-Schritten. Wenn die Daten gut in Cluster (Datenwolken) zu unterteilen sind, dann könnten wir als Abbruchbedingung auch die Summe der Differenzen der entsprechenden Prototypen zweier aufeinanderfolgender Zeitschritte nehmen und aufhören, wenn sie sich nicht oder kaum noch ändern.

Ein weiteres Problem ist die Wahl von  $k$ . Sicherlich scheint es auf den ersten Blick für unser Ziffernbeispiel sinnvoll zu sein,  $k = 10$  zu wählen, da wir es mit zehn unterschiedlichen Klassen zu tun haben. Das Problem dabei ist die Aufteilung im Raum. Es gibt Zahlen die sehr ähnlich sind und sich nicht wie runde Punktwolken im Raum trennen lassen. Sie umschlingen sich teilweise oder eine Ziffer befindet sich

in mehreren Punktwolken. Das bedeutet, dass wir ein größeres  $k$  verwenden müssen, um eine akzeptable Erkennungsrate zu erreichen.

Sollten wir  $k$  so groß machen, wie unsere Trainingsmenge Elemente hat, und bei der zufälligen Initialisierung nur Vektoren aus der Trainingsmenge erlauben, würden die Prototypen auf jeweils einem Element sitzen und wir würden im Endeffekt wieder  $k$ -nn mit einem Nachbarn durchführen.

### 13.1.2.3 Implementierung des k-means

Das Implementierungsbeispiel lehnt sich stark an die Klasse **Knn** aus dem vorhergehenden Abschnitt an. Die Daten sind eingelesen und zentriert.

Wir fügen den Listen der Trainingsvektoren und Label einige weitere zum Verwalten der Prototypen hinzu.

```
// wir setzen beide Daten public, damit wir in einen
// einfachen Zugriff erhalten
public double [][] trainingsdaten;
public int [] label;

private int [] myprototype;
private double prototypen [][][];
private int protolabel [];
private int k;
```

Im Konstruktor werden die Listen erzeugt und die Anzahl der Prototypen wird festgelegt.

```
// Konstruktor
public KMeans(int anzprototypen) {
    k = anzprototypen;
    trainingsdaten = new double[1000][192];
    label = new int[1000];
    myprototype = new int[1000];
    prototypen = new double[k][192];
    protolabel = new int[k];
}
```

Im Gegensatz zur **Knn**-Klasse wollen wir an dieser Stelle eine Funktion für die Abstandsberechnung einführen. Als Eingabe werden zwei gleichdimensionale Vektoren erwartet und der Euklidische Abstand beider zurückgeliefert.

```
// Euklidischer Abstand zwischen zwei Vektoren
private double distance(double [] v1, double [] v2) {
    double dist = 0, diff = 0;
    for (int i = 0; i < v1.length; i++) {
        diff = v1[i] - v2[i];
        dist += diff*diff;
    }
    return Math.sqrt(dist);
}
```

Die Trainingsroutine ist, aufgrund der vielen Arbeitsschritte, etwas umfangreich. Wir können zwischen den angebotenen Abbruchkriterien auswählen und Parameter für die maximale Anzahl der Iterationen `anzIter` und eine ausreichende Schranke für Veränderungen der Prototypen `epsilon` angeben.

Die Prototypen werden zufällig aus den Daten ausgewählt. Der Fall, dass zweimal oder sogar mehrmals derselbe Vektor als Prototyp gewählt wird, ist zu vernachlässigen. Zum einen ist es sehr unwahrscheinlich und zum anderen divergieren beide spätestens nach einer Iteration auseinander, wenn einem der beiden Vektoren einige Trainingsvektoren zugeordnet werden und er sich daraufhin in Richtung des Zentrums bewegt.

```
public void train(int abbruchkrit, int anzIter, double epsilon) {
    // Prototypen zufällig aus der Trainingsmenge wählen
    double d = Double.MAX_VALUE;
    int nearestprototype = 0;
    Random random = new Random();
    double sum[][] = new double[k][192];

    for (int i = 0; i < k; i++) {
        int rand = random.nextInt(trainingsdaten.length);
        prototypen[i] = trainingsdaten[rand];
        protolabel[i] = label[rand];
    }

    while ((0 < anzIter && 1 == abbruchkrit) ||
            (d > epsilon && 2 == abbruchkrit)) {
```

Die Prototypen sind ausgewählt und jetzt werden wir die `while`-Schleife erst wieder verlassen, wenn das gewählte Abbruchkriterium erfüllt ist. In dieser bescheidenen *k*-means-Version stehen zwei Kriterien zur Verfügung: a) Anzahl der Iterationen und b) Änderung der Prototypen.

Es folgt der Expectation-Schritt, also die Zuordnung der Trainingsvektoren zu den gewählten Prototypen.

```
// ****
// Expectationschritt:
//      - Trainingsdaten zu den nächstgelegenen Prototypen zuordnen
for (int i=0; i<1000; i++) {
    d = Double.MAX_VALUE;
    for (int j=0; j<k; j++) {
        if (d > distance(trainingsdaten[i], prototypen[j])) {
            nearestprototype = j;
            d = distance(trainingsdaten[i], prototypen[j]);
        }
    }
    myprototype[i] = nearestprototype;
}
// ****
```

Nach der Trennung des Raumes durch die Prototypen werden die Prototypen auf ihre neuen Zentren gesetzt. Falls ein Prototyp leer ausgehen sollte, gibt es bei dem Normierungsschritt des Vektors ein Problem. Daher wurden die Prototypen zur Schwerpunkttermittlung dazugenommen.

```

// *****
// Maximizationschritt:
//   - Prototypen werden zu den jeweiligen Daten zentriert
int xicount[] = new int[k];
double[][] prototypenref = new double[k][192];

for (int i=0; i<k; i++)
    for (int j=0; j<192; j++)
        sum[i][j] = 0;

// Schwerpunkte der Prototypen neu berechen
for (int i=0; i<1000; i++) {
    // Wieviele Vektoren gehören zu diesem Prototypen
    xicount[myprototype[i]]++;

    // Addition der Vektoren zu jedem Prototypen
    for (int j = 0; j < 192; j++)
        sum[myprototype[i]][j] += trainingsdaten[i][j];
}

for (int i=0; i<k; i++) {
    for (int j=0; j<192; j++) {
        prototypenref[i][j] = prototypen[i][j];
        prototypen[i][j] = (prototypen[i][j] + sum[i][j])
                           / (xicount[i]+1);
    }
}
// *****

```

Da als Abbruchkriterium die Änderung der Prototypen interessant sein kann, ermitteln wir die Differenzen.

```

// Abstandsberechnung der alten gegenüber den neuen Prototypen
d = 0;
for (int i=0; i<k; i++)
    d += distance(prototypen[i], prototypenref[i]);
anzIter--;
}

```

Die while-Schleife ist jetzt beendet und die neuen Label für die Prototypen werden demokratisch, nach Mehrheitsentscheid, festgelegt.

```

// *****
// Korrektur des Labels an Hand der Zugehörigkeiten
for (int i=0; i<k; i++){
    int ziffern[] = new int[10];

    for (int j=0; j<1000; j++)
        if (myprototype[j]==i)
            if (label[j]!=-1)
                ziffern[label[j]]++;

    int max=0;
    for (int l=1; l<10; l++)
        if (ziffern[l]>ziffern[max])
            max=l;

    protolabel[i]=max;
}
// *****

```

Um die gelernten Prototypen jetzt testen zu können, bietet die **KMeans**-Klasse eine `classify`-Methode an.

```
public int classify(double[] y) {
    int returnLabel;
    double d;

    returnLabel = -1;
    d = Double.MAX_VALUE;
    for (int i = 0; i < k; i++) {
        if(d > distance(prototypen[i], y)) {
            returnLabel = protolabel[i];
            d = distance(prototypen[i], y);
        }
    }
    return returnLabel;
}
```

Eine kleine Testfunktion zur Ermittlung der Erkennungsrate soll nicht fehlen. Wir justieren testweise 20 Prototypen im Ziffernraum und beenden die Lernphase, wenn sich die Prototypen fast nicht mehr ändern, mit  $\text{epsilon}=0.000001$ .

```
public static void main(String[] args) {
    // Abbruchkriterien
    // (1) wird die Anzahl Iterationen (anzIterationen) gewählt
    // (2) Veränderung der Positionen der Prototypen
    int abbruchKriterium      = 2;
    double epsilon             = 0.000001;
    int anzPrototypen         = 20;
    int anzIterationen        = 3;

    KMeans classifier          = new KMeans(anzPrototypen);
    classifier.readData("digits-training.txt");
    classifier.zentrierung();
    classifier.train(abbruchKriterium, anzIterationen, epsilon);

    // Teste Trainingsdaten auf sich selber und ermittle
    // die Erkennungsrate
    double er = 0;
    for (int i = 0; i < 1000; i++) {
        if (classifier.label[i] ==
            classifier.classify(classifier.trainingsdaten[i]))
            er++;
    }
    er /= 1000;
    System.out.println("Erkennungsrate = " + er);
}
```

Ein Testlauf zeigt eine Erkennungsrate von fast 80%. Das ist zunächst etwas unbefriedigend, aber mit verschiedenen Techniken weiter zu verbessern.

```
C:\JavaCode>java KMeans
Erkennungsrate = 0.782
```

Es sei dem interessierten Leser an dieser Stelle überlassen, sich noch intensiver mit Klassifizierungsalgorithmen und ihren Optimierungen auseinander zu setzen (z.B. in [25, 24]). Spannung verspricht dieses Thema allemal.

## 13.2 Spieltheorie

Wir betrachten Spiele als Suchprobleme. Das mögliche Verhalten eines Gegners muss dabei in Betracht gezogen werden. Im folgenden wollen wir dem Computer beibringen, perfekt **TicTacToe** zu spielen und immer den für sich besten Zug auszuwählen. Die Herausforderung dabei ist, dass das Verhalten des Gegners ungewiss ist und uns gegenüber meistens unproduktiv. Wir müssen also davon ausgehen, dass der Gegner immer den für sich besten Zug auswählt.

Wie schon bei der Einführung in die Algorithmik erwähnt, war Claude Elwood Shannon derjenige, der einen einfachen Algorithmus zur Lösung dieses Problems formulierte [27]. Eine Bewertungsfunktion spielt dabei eine sehr wichtige Rolle.

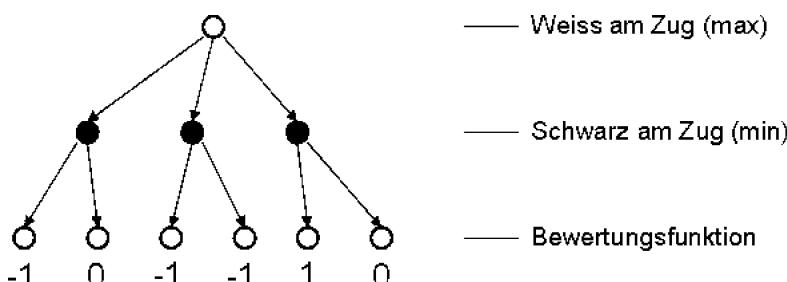
Für unser TicTacToe-Spiel liefert die Bewertungsfunktion, angewandt auf eine bestimmte Brettstellung, folgende Werte:

- 1** *X hat gewonnen*
- 0** *unentschieden*
- 1** *O hat gewonnen*

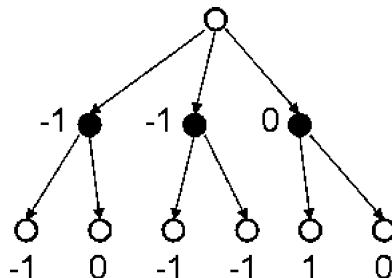
Zu Beginn des Spiels haben wir zunächst keine Informationen über dessen Ausgang. Wir müssen also einen Blick in die Zukunft werfen und rekursiv, via Brute Force, alle Züge ausprobieren und die Zugfolgen bis zur letzten Stellung untersuchen. Erst am Ende können wir eine Aussage darüber treffen, welcher der Wege am vielversprechendsten ist.

### 13.2.1 MinMax-Algorithmus

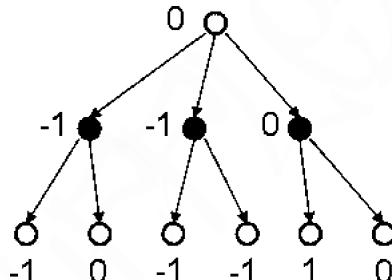
Das MinMax-Prinzip ist relativ einfach. Schauen wir es uns anhand eines Beispiels einmal an. Angenommen wir haben zwei Spieler **Schwarz** und **Weiss**. Es liegt eine Stellung vor, in der Weiss drei mögliche Züge hat. Anschließend ist Schwarz an der Reihe und hat jeweils zwei mögliche Züge. Eine Bewertungsfunktion liefert uns für jede der ermittelten Stellungen einen der Werte  $\{-1, 0, 1\}$ . Dabei soll  $-1$  für einen Sieg von Schwarz,  $0$  für ein Unentschieden und  $1$  für einen Sieg von Weiß stehen.



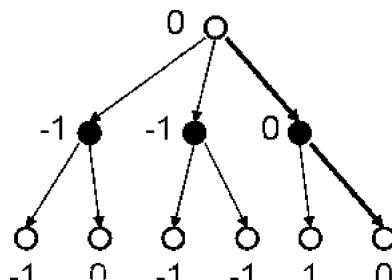
Der Spieler Schwarz gewinnt, wenn er zu einer Stellung mit der Bewertung  $-1$  gelangt. Darum wird er sich immer für den kleineren Wert entscheiden, er **minimiert** seine Stellung. Schwarz wählt also immer den kleinsten Wert aus. Somit ergeben sich folgende Bewertungen an den mittleren Knoten:



Weiss versucht seinerseits die Stellung zu **maximieren** und wählt immer den größten Wert aus. In diesem Fall entscheidet er sich für den Weg mit der Bewertung  $0$ , der ihm ein Unentschieden sichert.



Bei dem rekursiven Durchlaufen der Stellungen können wir uns nicht nur die Bewertungen speichern, sondern auch den Zug, der zu der jeweils besten Stellung führt. Am Ende der Berechnung können wir neben einer Bewertung sogar die beste Zugfolge, die sogenannte **Hauptvariante**, ausgeben.



Das im Wechsel stattfindende Maximieren und Minimieren gibt dem Algorithmus **MinMax** seinen Namen.

### 13.2.1.1 MinMax-Algorithmus mit unbegrenzter Suchtiefe

Wir können den MinMax-Algorithmus für den Fall formulieren, dass wir wie bei TicTacToe bis zu einer Stellung rechnen wollen und können, bei der eine Entscheidung über Sieg, Niederlage und Unentschieden getroffen wird und kein weiterer Zug möglich ist. Diese Stellungen nennen wir **terminale Stellungen**. Stellungen, in denen Weiss, also die Partei am Zug ist, die den Wert maximieren möchte, nennen wir *MaxKnoten* und analog dazu *MinKnoten*, diejenigen, bei denen Schwarz am Zug ist.

sei  $n$  die aktuelle Spielstellung  
 $S$  sei die Menge der Nachfolger von  $n$

$$\minmax(n) = \begin{cases} \text{Wert}(n), & \text{wenn } n \text{ terminale Stellung} \\ \max_{s \in S} \minmax(s), & \text{wenn } n \text{ ist MaxKnoten} \\ \min_{s \in S} \minmax(s), & \text{wenn } n \text{ ist MinKnoten} \end{cases}$$

Es stellt sich hier natürlich die Frage: *Wie können wir MinMax, z. B. für Schach, einsetzen?*

Es gibt im Schach schätzungsweise  $2,28 * 10^{46}$  legale Schachpositionen und im Durchschnitt ist eine Partie 50 Züge lang. Eine weiße oder schwarze Aktion wird als **Halbzug** definiert und zwei Halbzüge ergeben einen **Zug**. Das ergebe  $10^{120}$  Partieverläufe, die wir untersuchen müssten. Diese Zahl ist unvorstellbar groß, nur zum Vergleich: Die Anzahl der Atome im Weltall wird auf  $10^{80}$  geschätzt. Es ist uns also nicht möglich, von Beginn des Spiels bis zu allen terminalen Stellungen zu rechnen.

Aber wie können wir Programme trotzdem dazu bringen, Schach zu spielen?

### 13.2.1.2 MinMax-Algorithmus mit begrenzter Suchtiefe und Bewertungsfunktion

Die Idee besteht darin, nach einer bestimmten Suchtiefe abzubrechen und eine Bewertung dieser Stellung vorzunehmen. Diese Bewertung wird dann im Suchbaum zurückpropagiert [27].

Wie nun diese Bewertungsfunktion auszusehen hat, ist damit nicht gesagt. Sie sollte positive Werte für Weiss liefern, wobei größere Werte eine bessere Stellung versprechen und analog dazu negative Werte für favorisierte Stellungen von Schwarz. In der Schachprogrammierung wird meistens eine Funktion  $f$  verwendet, die  $n$  verschiedene Teilbewertungen  $f_i$  einer Stellung  $S$ , gewichtet aufaddiert und zurückgibt:

$$f(S) = \sum_{i=1}^n \alpha_i f_i(S)$$

Die Gewichte  $\alpha_1, \dots, \alpha_n$  müssen erst noch gefunden werden. Hier zwei kleine Beispiele für Teilbewertungen beim Schachspiel:

$$f_1(S) = materialWeiss(S) - materialSchwarz(S)$$

und

$$f_2(S) = mobilitaetWeiss(S) - mobilitaetSchwarz(S)$$

Mit Hilfe der Bewertungsfunktion  $f$ , der wir ab jetzt den Namen *evaluate* geben wollen, und einer Variable  $t$ , die die aktuelle Tiefe speichert, lässt sich nun der MinMax-Algorithmus wie folgt formulieren:

```
maxKnoten (Stellung X, Tiefe t) -> Integer
  if (t == 0)
    return evaluate(X)
  else
    w := -∞
    for all Kinder X1, ..., Xn von X
      v=minKnoten(Xi, t - 1)
      if (v > w)
        w = v
    return w
```

Analog dazu lässt sich die Funktion *minKnoten* formulieren. Das wird aber Teil der Übungsaufgaben sein. Gestartet wird der Algorithmus mit einem  $t > 0$ . Unser Algorithmus berechnet leider immer alle Zugmöglichkeiten. Eine Verbesserung ist der Alpha-Beta-Algorithmus, der Schranken für Spieler mitspeichert und deshalb an vielen Stellen Äste im Suchbaum einfach abschneiden kann.

Dieser Alpha-Beta-Algorithmus arbeitet in Kombination mit Transpositionstabellen ganz hervorragend. Das sind Tabellen in denen bereits berechnete Suchwerte gespeichert und später wieder verwendet werden. Aktuelle Schachprogramme schaffen mit weiteren Optimierungen eine Suchtiefe von 18 Halbzügen und mehr.

### 13.2.1.3 Beispiel TicTacToe

Für unser TicTacToe-Spiel definieren wir aber erstmal ein Spielbrett

```
public int [][] brett = new int [3][3];
```

und eine kleine Ausgabefunktion.

```

public void zeigeBrett(int [][] b){
    System.out.println("*****");
    for (int i=0; i<3; i++){
        System.out.print("* ");
        for (int j=0; j<3; j++){
            if (b[i][j]==-1)
                System.out.print("0 ");
            else if (b[i][j]==1)
                System.out.print("X ");
            else
                System.out.print("- ");
        }
        System.out.println("*");
    }
    System.out.println("*****");
}

```

Um den MinMax-Algorithmus verwenden zu können, brauchen wir für jede Stellung alle möglichen legalen Züge des Spielers, der an der Reihe ist. Dies ist bei unserem TicTacToe-Spiel sehr einfach, da wir einen legalen Zug einfach durch eine 0 in der Spielbrettmatrix identifizieren können. Eine 0 entspricht also einem leeren Feld. Nun müssen wir uns nur noch überlegen, in welcher Weise wir diesen Zug benennen wollen. Hier könnte man z. B. die x- und y-Koordinate aus der Matrix verwenden. Aus objektorientierter Sicht wäre es hier angebracht, eine Klasse **Move** zu schreiben und in genMoves eine **Move**-Liste zu erzeugen und zurückzugeben. Wir wollen das Programm aber möglichst klein und einfach halten und kodieren einen Zug, bestehend aus x- und y-Komponente, wie folgt:

$$\text{zug}(x,y) = 10 * x + y$$

Damit haben wir eine eindeutige Zahl als Repräsentanten für einen Zug. Mit folgenden beiden Funktionen lassen sich die x- und y-Komponenten wieder leicht ermitteln:

$$x = \text{zug}(x,y)/10 \text{ und } y = \text{zug}(x,y)\%10$$

```

public int [] genZuege(int [][] b){
    // wieviele Züge gibt es?
    int anzZuege=0;
    for (int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            if (b[i][j]==0)
                anzZuege++;

    // speichere die Züge
    int [] zuege = new int [anzZuege];
    anzZuege = 0;
    for (int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            if (b[i][j]==0){
                zuege[anzZuege] = i*10+j;
                anzZuege++;
            }

    return zuege;
}

```

Nun benötigen wir noch die Bewertungsfunktion evaluate und schon können wir uns daran machen, den MinMax-Algorithmus zu implementieren.

```
public int evaluate(int [][] b){
    // prüfe Zeilen
    int sum=0;
    for (int i=0; i<3; i++)
        sum += b[i][0] + b[i][1] + b[i][2];
    if (sum == -3)
        return -1;
    else if (sum == 3)
        return 1;

    // prüfe Spalten
    sum=0;
    for (int j=0; j<3; j++)
        sum += b[0][j] + b[1][j] + b[2][j];
    if (sum == -3)
        return -1;
    else if (sum == 3)
        return 1;

    // prüfe die Diagonale von links oben nach rechts unten
    sum = b[0][0] + b[1][1] + b[2][2];
    if (sum == -3)
        return -1;
    else if (sum == 3)
        return 1;

    // prüfe die Diagonale von links unten nach rechts oben
    sum = b[0][2] + b[1][1] + b[2][0];
    if (sum == -3)
        return -1;
    else if (sum == 3)
        return 1;

    // unentschieden
    return 0;
}
```

Die Bewertungsfunktion geht davon aus, dass einer der beiden Spieler gewonnen hat oder kein Zug mehr möglich ist, also neun Steine auf dem Brett stehen. Wir brauchen noch eine Funktion, die feststellt, ob alle Spielfelder besetzt sind.

```
public int zaehleZeichen(int [][] b){
    int count=0;
    for (int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            if (b[i][j]!=0)
                count++;
    return count;
}
```

Jetzt können wir uns daran machen, den MinMax-Algorithmus zu schreiben. Dazu implementieren wir zwei Funktionen, die sich gegenseitig im Wechsel aufrufen. Zunächst die MinMax-Funktion aus Sicht vom Spieler X, für den wir die Stellung maximieren wollen.

```

public int minmaxX(int [][] b){
    // Blatt erreicht
    if (zahleZeichen(b)==9)
        return evaluate(b);

    int max = -5;
    int [] zuege = genZuege(b);

    for (int i=0; i<zuege.length; i++) {
        // führe X-Zug aus
        b[zuege[i]/10][zuege[i]%10] = 1;

        int wert = minmaxO(b);
        if (wert>max)
            max = wert;

        // nimm Zug zurück
        b[zuege[i]/10][zuege[i]%10] = 0;
    }

    return max;
}

```

Die MinMax-Funktion für Spieler *O* wird Teil der Übungsaufgaben sein. Wir können an dieser Stelle aber vorweg schauen, ob es bei genauem Spiel einen Gewinner geben kann oder ob das Spiel theoretisch immer Remis endet.

Dazu erzeugen wir eine Instanz der Klasse TicTacToe und starten das Spiel.

```

public static void main(String [] args){
    TicTacToe ttt = new TicTacToe();
    ttt.zeigeBrett(ttt.brett);
    System.out.println("MinMax liefert den Wert: "
                        + ttt.minmaxX(ttt.brett));
}

```

Geben wir mal die Startstellung in MinMax ein und schauen uns die Bewertung an:

```

C:\JavaCode>java TicTacToe
*****
* - - -
* - - -
* - - -
*****
MinMax liefert den Wert: 0

```

Daraus schlussfolgern wir, dass es bei korrektem Spiel beider Spieler keinen Gewinner geben kann.

### 13.3 Zusammenfassung und Aufgaben

#### Zusammenfassung

Dieses Kapitel hat uns mit Beispielen aus der Mustererkennung und Spieltheorie verschiedene Ansätze der Künstlichen Intelligenz gezeigt. Mit Hilfe der beiden Methoden *k*-nn und *k*-means konnte der Computer Bilder von Ziffern richtig klassifizieren.

$k$ -nn hatte zwar eine bessere Erkennungsrate, war aber in der Erkennungsphase zeitlich sehr viel aufwändiger als  $k$ -means.

Mit einem perfekt spielenden TicTacToe-Programm sind wir in die Spieltheorie eingestiegen. Das MinMax-Prinzip in Kombination mit einer Bewertungsfunktion war ein Durchbruch bei der Entwicklung spielender Maschinen. Heutzutage basieren alle Suchalgorithmen bei Schachprogrammen auf dem MinMax-Prinzip.

## Aufgaben

Übung 1) Implementieren Sie die Funktion minmax0 analog zu minmaxX aus Abschnitt 13.2.1. Jetzt vervollständigen Sie die Klasse **TicTacToe**, so dass folgende main-Funktion den MinMax-Algorithmus startet:

```
public static void main(String[] args){  
    TicTacToe ttt = new TicTacToe();  
    ttt.showBoard(ttt.board);  
    System.out.println("MinMax liefert den Wert: "  
                       + ttt.minmaxX(ttt.board));  
}
```

Übung 2) Erweitern Sie den MinMax-Algorithmus auf die Speicherung des letzten Zuges. Schreiben Sie eine Eingabefunktion für das TicTacToe-Spiel und spielen Sie gegen den Computer.

Übung 3) Suchen Sie im Internet nach Datenbanken von Gesichtern. Passen Sie die Einlesefunktion an und verwenden Sie Ihre beiden Methoden  $k$ -nn und  $k$ -means für die Erkennung von Gesichtern.

## Tag 14: Entwicklung einer größeren Anwendung

Anhand kleiner Projekte haben wir wichtige Konzepte und Methoden der Softwareentwicklung kennengelernt. Jetzt ist es an der Zeit, ein großes Projekt von der Entwurfssphase bis zur fertigen Präsentation durchzuführen. Als Beispiel haben wir Tetris gewählt und unsere Variante mit dem Namen **TeeTrist** versehen, da sie über einen überschaubaren Funktionsumfang verfügt und in ihrer Darstellung sicherlich noch ausbaufähig ist.

### 14.1 Entwurf eines Konzepts

Um mit diesem Projekt beginnen zu können, müssen wir zunächst ein paar grundlegende Fragen<sup>1</sup> beantworten. Da es sich um eine bereits existierende Anwendung handelt, können wir das Regelwerk und die entsprechenden Anforderungen zusammentragen. In der Softwaretechnik wird diese Phase als **Anforderungsanalyse** bezeichnet. Wir verwenden eine Drei-Schichten-Architektur (siehe z. B. [35]), in der GUI, konzeptuelles Schema (Spiellogik) und Datenhaltung getrennt entwickelt werden. Dazu müssen wir die Schnittstellen zwischen Spieldaten und -logik sowie Spiellogik und GUI definieren. Die Antworten auf die folgenden Fragen ergeben eine **Systembeschreibung**.

#### Wo lassen sich detaillierte Informationen zu Tetris finden?

Eine gute Informationsquelle stellt Wikipedia dar. Auf der folgenden Seite <http://de.wikipedia.org/wiki/Tetris> können wir uns mit dem Spielgeschehen und den Feinheiten vertraut machen.

#### Welche Regeln gelten für das Spiel und wie sieht ein typischer Spielverlauf aus?

Es gibt ein Spielfeld, in das von oben nacheinander Spielsteine gelangen. Diese besitzen eine bestimmte Fallgeschwindigkeit. Der Benutzer kann den Fall der Steine

---

<sup>1</sup> Diese Fragen sind speziell auf unser Projekt abgestimmt und sollen uns helfen, die Arbeitsschritte und Dynamik des Projekts zu verstehen.

durch Drehung um 90, Verschiebung nach links oder rechts und Erhöhung der Fallgeschwindigkeit manipulieren. Ein Spielstein wird durch den Rand des Spielfelds oder andere bereits auf dem Spielfeldgrund befindliche Steine gestoppt, er bleibt an der Position liegen und der nächste Stein (wird zufällig gewählt) gelangt von oben in das Spielfeld. Sollten die Steine, nachdem ein Stein eine feste Position erhalten hat, eine komplett gefüllte Reihe ergeben, so verschwinden diese aus dem Spielfeld und der Benutzer erhält einen Punktebonus (dieser ist abhängig von der Anzahl der kompletten Reihen, möglich sind 1 – 4). Das Spiel endet, wenn die Höhe der im Spielfeld positionierten Spielsteine zu groß geworden ist und kein neuer Spielstein mehr auf das Feld passt. Ziel ist es, möglichst viele Punkte zu erreichen.

### Wie soll die Interaktion zwischen Benutzer und Anwendung stattfinden?

Die Anwendung läuft in einer Schleife und produziert Spielsteine, die dann automatisch fallen. Der Benutzer soll mit den auf der Tastatur befindlichen Pfeil-Tasten →, ←, ↓, ↑ diese Steine manipulieren können. Zusätzlich gibt es die Möglichkeit, mit **q** das Spiel zu beenden und die Anwendung zu schließen, mit **n** ein neues Spiel zu starten oder mit der **Leertaste** den Stein sofort nach unten fallen zu lassen.

### Welche Spieldaten müssen gespeichert werden?

Es müssen die bereits positionierten Spielsteine und die Position des aktuell fallenen Spielsteins innerhalb der Spielfeldmatrix festgehalten werden. Die Spielsteinarten werden mit unterschiedlichen Zahlen repräsentiert, um bei der späteren Darstellung in der GUI eine farbliche Unterscheidung vornehmen zu können. Zu den gehaltenen Daten gehören auch die Baupläne der aus einzelnen Kästchen zusammengesetzten Spieltypen.

### Wie soll die Spiellogik realisiert werden?

Da wir eine Anwendung realisieren wollen, die zeitabhängig ist (Fallgeschwindigkeit des Spielsteins), müssen wir diesen Teil eigenständig laufen lassen. Das führt dazu, dass wir uns mit dem Konzept der Threads in Java vertraut machen werden. Die Spiellogik verwaltet die Kommunikation mit dem Benutzer und den aktuellen Zustand des Spiels. Nach einer bestimmten Zeit wird der aktuelle Stein ein Feld nach unten bewegt, falls das möglich ist. Sollte der Stein nicht weiter fallen können, wird untersucht, ob vollständige Zeilen vorliegen und diese aus dem Spielfeld genommen werden können. Anschließend wird überprüft, ob das Spielfeld bereits voll und demnach das Spiel beendet ist oder ein neuer zufälliger Stein ins Spiel gebracht werden kann.

### Wie definieren wir die Schnittstelle zwischen Spiellogik und Spieldaten?

Die Spieldaten repräsentieren alle wichtigen Spielzustandsinformationen, wie z. B. die aktuelle Belegung des Spielbretts und Methoden für die Änderung dieser Zustände. Zu den Daten gehören Informationen über die Spielbrettgröße, aktuelle Punktzahl und Inhalt einer bestimmten Position der Spielmatrix. Für die Spiellogik werden Methoden bereitgestellt, die Einfluss auf den aktuellen Zustand nehmen können, wie z. B. „gib nächsten Stein“, „verschiebe Stein nach links oder rechts“, „drehe Stein“,

„lass Stein schneller oder ganz nach unten fallen“ und „überprüfe auf vollständige Zeilen“. In die andere Richtung, also von den Daten zur Spiellogik gibt es keine Kommunikation.

### Wie sieht die Schnittstelle zwischen GUI und Spiellogik aus?

Die Spiellogik liefert der GUI die folgenden Informationen: Breite und Höhe des Spielbretts, aktuelle Punktezahl und die Information über den Spielsteintyp an einer bestimmten Position innerhalb des Spielfelds. Des Weiteren veranlasst die Spiellogik, nach einer Änderung des Systemzustands, die Aktualisierung der Anzeige und liefert die Information, ob das Spiel zu Ende ist.

### Welche Komponenten werden für eine grafische Benutzeroberfläche (GUI) benötigt?

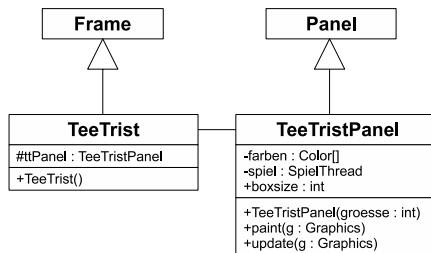
Als Oberfläche genügt ein Fenster in dem wir ein Panel (GUI-Element, das ähnlich wie ein Fenster mehrere GUI-Elemente aufnehmen kann) einfügen. Das Panel repräsentiert das Spielfeld und die darauf befindlichen Steine.

#### 14.1.1 Klassendiagramm

Um die Systembeschreibung zu vervollständigen, erarbeiten wir noch ein Klassendiagramm. Schauen wir uns die Klassen der drei Komponenten GUI, Spiellogik und Datenverwaltung zunächst einzeln und anschließend im Kontext an.

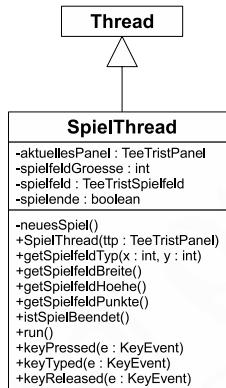
##### 14.1.1.1 GUI Klassen

Als äußeren Rahmen nehmen wir ein Fenster und betten dort ein Panel ein. Auf diesem Panel werden alle Spielinformationen und die Dynamik der Spielsteine innerhalb des Spielfelds angezeigt. Das Fenster **TeeTrist** erbt von **Frame** und **TeeTristPanel** von **Panel**.



### 14.1.1.2 Spiellogik

Da wir die Spiellogik als unabhängigen Prozess laufen lassen wollen, erbt die Klasse **SpielThread** von der Klasse **Thread**. Die beiden wichtigsten Methoden sind `run` und `keyPressed`. Die Methode `run` wird ein einziges Mal zu Beginn gestartet und lässt sich als unendliche Schleife verstehen. Die Kommunikation mit dem Benutzer wird durch die Methode `keyPressed` realisiert. Die GUI-Klasse **TeeTristPanel** greift auf die Methoden `getSpielfeldBreite`, `getSpielfeldHoehe`, `getSpielfeldPunkte`, `getSpielfeldTyp` und `istSpielBeendet` zu.



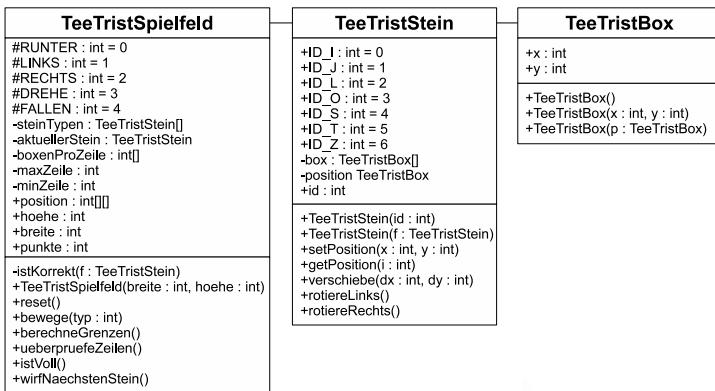
### 14.1.1.3 Spieldatenverwaltung

Die Klasse **TeeTristBox** repräsentiert lediglich die Position eines Kästchens relativ zum Ursprung des Spielsteins.

Um die Kästchen zu einem Spielstein zusammenzufassen, repräsentiert die Klasse **TeeTristStein** die relativen Positionen der 4 Steine und die aktuelle Position des Steins innerhalb der Spielmatrix. Die Funktionen `getPositionX(int i)` und `getPositionY(int i)` liefern die  $x$ - und  $y$ -Koordinaten des  $i$ -ten Kästchens für die Spielfeldmatrix und die Funktion `setPosition(int x, int y)` setzt den Ursprung neu. Des Weiteren bietet diese Klasse Funktionen für die Manipulation eines Spielsteins, wie `rotiereLinks`, `rotiereRechts` und `verschiebe`.

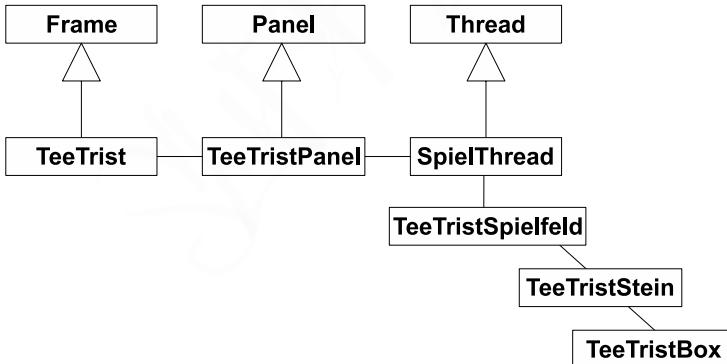
Die Hauptverwaltung der Spielinformationen befindet sich in der Klasse **TeeTristSpielfeld**. Informationen über den aktuellen Stein `aktuellerStein`, das Spielfeld `position[][][]`, die Spielfeldgröße `hoehe` und `breite` sowie die bisher erreichten Punkte `punkte` werden gespeichert. Zu den Methoden zählen die Reinitialisierung des Spielfelds `reset`, die Bewegung des aktuellen Spielsteins `bewege` (verwendet die Methode `istKorrekt`, die `true` liefert, falls sich die Position wirklich verändert hat), die Erzeugung eines neuen zufälligen Steins `wirfNaechstenStein`, die Berechnung der neuen Grenzen für eine optimierte Suche der vollen Zeilen `berechneGrenzen`, die Überprüfung und Löschung voller Zeilen `ueberpruefen`.

Zeilen und die Methode `istVoll`, die prüft, ob die maximale Zeilenhöhe erreicht wurde.



#### 14.1.1.4 Komplettes Klassendiagramm

Um die Darstellung nicht unnötig unübersichtlich zu gestalten, genügt es uns, die Klassen und die Verbindungen der Klassen untereinander im gesamten Klassendiagramm darzustellen.



## 14.2 Implementierung

### 14.2.1 Klasse TeeTristBox

Die Klasse `TeeTristBox` ist eine Containerklasse für die beiden Variablen `x` und `y`, die die aktuelle Position einer Box speichern. Wir stellen für spätere Zwecke mehrere Konstruktoren zur Verfügung. Für unbekannte Positionen erzeugt der Konstruktor eine Box, die sich im Ursprung befindet.

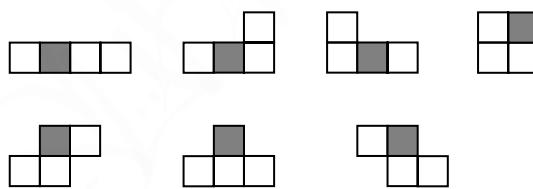
```

1 public class TeeTristBox {
2     public int x;
3     public int y;
4
5     public TeeTristBox() {
6         this(0,0);
7     }
8
9     public TeeTristBox(int x, int y) {
10        this.x = x;
11        this.y = y;
12    }
13
14     public TeeTristBox(TeeTristBox p) {
15        this.x = p.x;
16        this.y = p.y;
17    }
18 }
```

Der letzte Konstruktor entspricht einem Copy-Konstruktor. Die Eingabe einer Instanz vom selben Typ, führt zu einer identischen Kopie (siehe dazu 7.2.5.1).

### 14.2.2 Klasse TeeTristStein

Die Aufgabe besteht nun darin, die sieben verschiedenen Spielsteine zu erzeugen. Die grau dargestellten Boxen entsprechen dem Ursprung mit der relativen Position (0,0):



Die Steinarten werden durch die Integer-Konstanten ID\_I, ID\_J,..., ID\_Z assoziiert. Die Liste box verwaltet die zu dem Spielstein gehörenden vier Boxen und in position wird die aktuelle Position, relativ zum Spielfeld, gespeichert.

```

1 public class TeeTristStein {
2     // 0#00
3     static final int ID_I = 0;
4     // 0
5     // 0#0
6     static final int ID_J = 1;
7     // 0
8     // 0#0
9     static final int ID_L = 2;
10    // 0#
11    // 00
12    static final int ID_O = 3;
```

```

14 // #0
15 // 00
16 static final int ID_S = 4;
17 // 000
18 // #
19 static final int ID_T = 5;
20 // 0#
21 // 00
22 static final int ID_Z = 6;
23
24 private TeeTristBox box[];
25 private TeeTristBox position;
26
27 public int id;

```

Bei der Erzeugung einer Instanz der Klasse, werden je nach Typ, die entsprechenden Boxen hinzugefügt. An dieser Stelle wird ebenfalls wieder ein Copy-Konstruktor bereitgestellt.

```

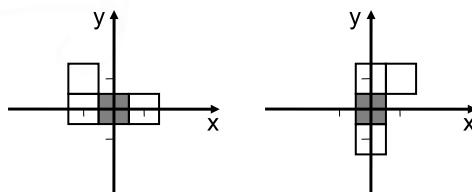
29 // Konstruktor
30 public TeeTristStein(int id) {
31     box = new TeeTristBox[4];
32     for(int i=0; i<box.length; i++)
33         box[i] = new TeeTristBox();
34     position = new TeeTristBox();
35     this.id = id;
36
37     int index = 0;
38     switch (id) {
39         // 0#000
40         case ID_I:
41             box[index++]=new TeeTristBox(-1,0);
42             box[index++]=new TeeTristBox(0,0);
43             box[index++]=new TeeTristBox(1,0);
44             box[index++]=new TeeTristBox(2,0);
45             break;
46         // 0
47         // 0#0
48         case ID_J:
49             box[index++]=new TeeTristBox(1,1);
50             box[index++]=new TeeTristBox(-1,0);
51             box[index++]=new TeeTristBox(0,0);
52             box[index++]=new TeeTristBox(1,0);
53             break;
54         // 0
55         // 0#0
56         case ID_L:
57             box[index++]=new TeeTristBox(-1,1);
58             box[index++]=new TeeTristBox(-1,0);
59             box[index++]=new TeeTristBox(0,0);
60             box[index++]=new TeeTristBox(1,0);
61             break;
62         // 0#
63         // 00
64         case ID_0:
65             box[index++]=new TeeTristBox(-1,0);
66             box[index++]=new TeeTristBox(0,0);
67             box[index++]=new TeeTristBox(-1,-1);
68             box[index++]=new TeeTristBox(0,-1);
69             break;
70         // #0
71         // 00
72         case ID_S:

```

```

73     box [ index ++]=new TeeTristBox ( 0 , 0 );
74     box [ index ++]=new TeeTristBox ( 1 , 0 );
75     box [ index ++]=new TeeTristBox ( - 1 , - 1 );
76     box [ index ++]=new TeeTristBox ( 0 , - 1 );
77     break ;
78     // 000
79     // #
80 case ID_T:
81     box [ index ++]=new TeeTristBox ( 0 , 0 );
82     box [ index ++]=new TeeTristBox ( - 1 , 1 );
83     box [ index ++]=new TeeTristBox ( 0 , 1 );
84     box [ index ++]=new TeeTristBox ( 1 , 1 );
85     break ;
86     // 0#
87     // 00
88 case ID_Z:
89     box [ index ++]=new TeeTristBox ( - 1 , 0 );
90     box [ index ++]=new TeeTristBox ( 0 , 0 );
91     box [ index ++]=new TeeTristBox ( 0 , - 1 );
92     box [ index ++]=new TeeTristBox ( 1 , - 1 );
93     break ;
94 }
95 }
96
97 public TeeTristStein ( TeeTristStein f ) {
98     box = new TeeTristBox [ 4 ];
99     for ( int i=0; i<box.length; i++)
100         box [ i ] = new TeeTristBox ( f .box [ i ] );
101     position = new TeeTristBox ( f .position );
102     id = f .id;
103 }
```

Mit `setPosition` wird die Position des Spielsteins im Spielfeld festgelegt. Die Methoden `getPositionX` und `getPositionY` liefern die jeweiligen  $x$ - und  $y$ -Koordinaten. Der Spielstein kann aber auch mit Hilfe der drei folgenden Funktionen bewegt werden: `verschiebe`, `rotiereLinks` und `rotiereRechts`. An dieser Stelle wollen wir die Rechtsrotation um  $90^\circ$  des Spielsteins `ID_L` exemplarisch erläutern:



Wir interpretieren jede Box als Vektor. Deshalb können wir die Koordinaten jeder Box, mittels einer einfachen linearen Transformation, um  $90^\circ$  nach rechts drehen:

$$\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -x \end{pmatrix}$$

Da die  $x$ - und  $y$ -Werte in `box` gespeichert sind, können wir die Koordinaten wie folgt neu setzen:

```

hilfe = box[i].y;
box[i].y = -box[i].x;
box[i].x = hilfe;

```

Wollen wir an dieser Stelle nach links drehen, verwenden wir die folgende Rotationsgleichung:

$$\begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}$$

```

105 // public Methoden
106 public void setPosition(int x, int y) {
107     position.x = x;
108     position.y = y;
109 }
110
111 public int getPositionX(int i) {
112     return box[i].x + position.x;
113 }
114
115 public int getPositionY(int i) {
116     return box[i].y + position.y;
117 }
118
119 public void verschiebe(int dx, int dy) {
120     position.x += dx;
121     position.y += dy;
122 }
123
124 public void rotiereLinks() {
125     if (id==ID_O) return;
126     int hilfe;
127     for (int i=0; i<box.length; i++) {
128         hilfe = box[i].x;
129         box[i].x = -box[i].y;
130         box[i].y = hilfe;
131     }
132 }
133
134 public void rotiereRechts() {
135     if (id==ID_O)
136         return;
137     int hilfe;
138     for (int i=0; i<box.length; i++) {
139         hilfe = box[i].y;
140         box[i].y = -box[i].x;
141         box[i].x = hilfe;
142     }
143 }
144 }
```

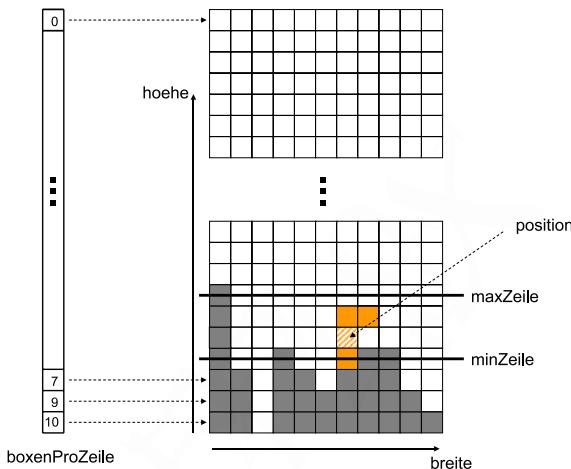
### 14.2.3 Klasse TeeTristSpielfeld

Die Klasse **TeeTristSpielfeld** verdient eine intensive Besprechung, da in ihr der größte Verwaltungsaufwand steckt. Die Integer-Konstanten RUNTER, LINKS, RECHTS, DREHE und FALLEN, repräsentieren die unterschiedlichen Bewegungen des

aktuellen Spielsteins aktuellerStein im Feld. Da die Klasse auch die Aufgabe hat, den nächsten Stein zu erzeugen, sind Ihr alle möglichen sieben Spielsteintypen mit steinTypen bekannt.

Die Matrix position speichert alle Boxtypen der bereits gespielten Spielsteine. boxenProZeile speichert die Anzahl der Boxen in jeder Zeile. Damit überprüfen wir später, ob eine Zeile gefüllt ist, zu Punkten führt und entsprechend gelöscht werden kann.

Veranschaulichen wir uns die verschiedenen Parameter anhand folgender Darstellung:



```

1 import java.util.Random;
2
3 public class TeeTristSpielfeld {
4     static final int RUNTER = 0;
5     static final int LINKS = 1;
6     static final int RECHTS = 2;
7     static final int DREHEN = 3;
8     static final int FALLEN = 4;
9
10    private TeeTristStein steinTypen[] = new TeeTristStein[7];
11    private TeeTristStein aktuellerStein;
12    public int position[][];
13    private int[] boxenProZeile;
14    private int maxZeile, minZeile;
15    public int hoehe, breite, punkte;

```

Im Konstruktor wird die Spielfeldgröße durch die Parameter breite und hoehe festgelegt und die Spielsteine initialisiert. Im sichtbaren Bereich der Matrix fehlen vier Zeilen, die für die Startposition eines Spielsteins gedacht ist.

```

17 // Konstruktor
18 public TeeTristSpielfeld(int breite, int hoehe) {
19     position      = new int[breite][hoehe+4];
20     boxenProZeile = new int[hoehe+4];
21
22     this.breite = breite;
23     this.hoehe = hoehe;
24
25     for (int id = TeeTristStein.ID_I; id<=TeeTristStein.ID_Z; id++)
26         steinTypen[id] = new TeeTristStein(id);
27
28     reset();
29 }
```

Die im Konstruktor aufgerufene Methode `reset` löscht die Matrixeinträge und setzt alle auf -1, wie auch die `maxZeile`. Die Anzahl der Boxen pro Zeile und die Punkte werden auf Null gesetzt.

```

31 // Reset-Methode
32 public void reset() {
33     for (int y=0; y<hoehe+4; y++) {
34         boxenProZeile[y] = 0;
35         for (int x=0; x<breite; x++)
36             position[x][y] = -1;
37     }
38
39     maxZeile = -1;
40     punkte = 0;
41 }
```

Die Spielsteine können verschiedene Bewegungen ausführen. Diese werden in der Methode `bewege`, durch die Löschung der aktuellen Steinposition und einer anschließenden Verschiebung oder Rotation, realisiert. Falls die gewünschte Bewegung nicht möglich ist, wird ein `false` zurückgeliefert, ansonsten ein `true`.

```

43 public boolean bewege(int typ) {
44     TeeTristStein neuerStein;
45     boolean bewegt;
46     int x,y;
47     for (int i=0; i<4; i++) {
48         x = aktuellerStein.getPositionX(i);
49         y = aktuellerStein.getPositionY(i);
50         position[x][y] = -1;
51     }
52
53     neuerStein = new TeeTristStein(aktuellerStein);
54     switch(typ) {
55         case RUNTER:
56             neuerStein.verschiebe(0, -1);
57             break;
58         case LINKS:
59             neuerStein.verschiebe(-1, 0);
60             break;
61         case RECHTS:
62             neuerStein.verschiebe(1, 0);
63             break;
64         case DREHE:
```

```

65      // Alternative hier ist die Linkssrotation rotateLeft()
66      neuerStein.rotiereRechts();
67      break;
68  case FALLEN:
69      while (istKorrekt(neuerStein))
70          neuerStein.verschiebe(0, -1);
71      neuerStein.verschiebe(0, 1);
72      break;
73  }
74 if (bewegt==istKorrekt(neuerStein))
75     aktuellerStein = neuerStein;
76
77 for (int i=0; i<4; i++) {
78     x = aktuellerStein.getPositionX(i);
79     y = aktuellerStein.getPositionY(i);
80     position[x][y] = aktuellerStein.id;
81 }
82 return bewegt;
83 }
```

Die Methode `istKorrekt` überprüft, ob der sich bewegte Stein innerhalb des Feldes liegt und keine der im Feld befindlichen Boxen überlagert.

```

85 private boolean istKorrekt(TeeTristeStein f) {
86     int x, y;
87     for(int i=0; i<4; i++) {
88         x = f.getPositionX(i);
89         y = f.getPositionY(i);
90
91         if (x<0 || x>=this.breite || y<0)
92             return false;
93
94         if (position[x][y]!=-1)
95             return false;
96     }
97     return true;
98 }
```

Nachdem festgestellt wurde, dass der aktuelle Stein nicht mehr ziehen kann, müssen einige Parameter aktualisiert werden.

```

100 public void berechneGrenzen() {
101     minZeile = Integer.MAX_VALUE;
102     int y;
103     for(int i=0; i<4; i++) {
104         y = aktuellerStein.getPositionY(i);
105         boxenProZeile[y]++;
106         maxZeile = Math.max(maxZeile, y);
107         minZeile = Math.min(minZeile, y);
108     }
109 }
```

Der Stein kann nicht mehr ziehen und die Grenzen `minZeile` und `maxZeile` wurden aktualisiert. Innerhalb dieser Region wird nun überprüft, ob Zeilen vollständig sind und entsprechend zu Punkten führen und entfernt werden können.

```

111 public void ueberpruefeZeilen() {
112     for (int y=minZeile+3, inc=1; y>=minZeile; y--) {
113         if (boxenProZeile[y]==this.breite) {
114             for (int i=y; i<maxZeile; i++) {
115                 for (int j=0; j<breite; j++)
116                     position[j][i] = position[j][i+1];
117                     boxenProZeile[i] = boxenProZeile[i+1];
118             }
119
120             for (int i=0; i<breite; i++)
121                 position[i][maxZeile] = -1;
122
123             boxenProZeile[maxZeile] = 0;
124             maxZeile--;
125             punkte+=(inc++)*50;
126         }
127     }
128 }
```

Das Spiel endet, wenn die maximale Höhe erreicht wurde.

```

130 public boolean istVoll() {
131     return maxZeile>=hoehe;
132 }
```

Falls das Spiel nicht zu Ende ist, muss ein neuer Stein ermittelt werden.

```

134 public void wirfNaechstenStein() {
135     aktuellerStein = new TeeTristStein(steinTypen[(int)(Math.random()*
136                                         steinTypen.length)]);
137
138     switch(aktuellerStein.id) {
139         case TeeTristStein.ID_I:
140         case TeeTristStein.ID_J:
141         case TeeTristStein.ID_L:
142         case TeeTristStein.ID_T:
143             aktuellerStein.setPosition(breite/2, hoehe);
144             break;
145         default:
146             aktuellerStein.setPosition(breite/2, hoehe+1);
147     }
148 }
149 }
```

#### 14.2.4 Klasse SpielThread

Für unsere Klasse **SpielThread** verwenden wir erstmalig das Konzept der Parallelisierung. Bei den Methoden der Programmerstellung (Abschnitt 2.5) haben wir bereits erfahren, dass Parallelisierung eine der drei Schlüsselmethoden ist.

Wir wollen die Animation laufen lassen und diese soll nach einer gewissen Zeit Änderungen im Spielzustand vornehmen, z. B. den Spielstein fallen lassen oder Punkte

vergeben. Dabei werden Tastatureingaben zur Manipulation des Spielgeschehens erwartet. Zu einer vernünftigen Realisierung verwenden wir einen **Thread**. Ein Thread ist ein Prozess oder ein Programmteil, der sich im Prinzip autonom verhält aber nur so lange funktionieren kann, bis er gestoppt wird oder das Hauptprogramm, das diesen Thread erzeugt und gestartet hat, beendet wird.

Die Handhabung in Java ist relativ einfach<sup>2</sup>. In unserem Beispiel genügt es, zunächst von der Klasse **Thread** abzuleiten und nur die Methode `run` zu überschreiben. Eine ausführliche Einführung zu diesem sehr wichtigen Thema findet sich z. B. in [7].

Schauen wir uns die Implementierung der Klasse **SpielThread** an, die die komplette Spiellogik beinhaltet:

```

1 import java.awt.event.KeyEvent;
2 import java.awt.event.KeyListener;
3
4 public class SpielThread extends Thread implements KeyListener{
5     private TeeTristPanel aktuellesPanel;
6     private int spielfeldGroesse;
7     private TeeTristSpielfeld spielfeld;
8     private boolean spielende;
```

Neben der Vererbung von **Thread** implementieren wir das Interface **KeyListener**, um auf die Tastatureingaben reagieren zu können. Das GUI-Element **TeeTristPanel** kümmert sich, wie wir anschließend sehen werden, um die Visualisierung des Spielfelds und stellt die Schnittstelle zwischen der Spiellogik und GUI dar. Eine Instanz von **TeeTristSpielfeld** speichert den aktuellen Spielzustand.

```

10 // Konstruktor
11 public SpielThread(TeeTristPanel ttp) {
12     super();
13     aktuellesPanel = ttp;
14     spielfeldGroesse = ttp.boxSize;
15
16     // Spielfeld wird angelegt
17     spielfeld = new TeeTristSpielfeld(11, 20);
18     spielfeld.wirfNaechstenStein();
19
20     spielende = false;
21 }
22
23 // get-Methoden
24 public int getSpielfeldTyp(int x, int y){
25     return spielfeld.position[x][y];
26 }
27
28 public int getSpielfeldBreite(){
29     return spielfeld.breite;
30 }
31
32 public int getSpielfeldHoehe(){
33     return spielfeld.hoehe;
34 }
```

<sup>2</sup> Die Handhabung mehrerer Threads dagegen ist sicherlich nicht als einfach einzurichten, da beispielsweise Probleme bei der Synchronisation zu lösen sind [13, 14].

```

35  public int getSpielfeldPunkte(){
36      return spielfeld.punkte;
37  }
38
39  public boolean istSpielBeendet(){
40      return spielende;
41  }
42

```

Im Konstruktor wird die Spielfeldgröße festgelegt und es werden einige get- und set-Methoden bereitgestellt.

Die Methode `neuesSpiel` startet den **Thread**, falls er bisher noch nicht gestartet wurde, initialisiert ihn und das Spielfeld neu und aktualisiert anschließend das Panel.

```

44  private void neuesSpiel() {
45      if (!isAlive())
46          start();
47      spielende = false;
48      spielfeld.reset();
49      spielfeld.wirfNaechstenStein();
50      aktuellesPanel.repaint();
51      resume();
52  }

```

Die ganze Spiellogik von **TeeTrist** ist nun in der folgenden Methode `run` versteckt. Die Methode `run` wird von **Thread** überschrieben, schauen wir sie uns erst einmal an:

```

54  // Threadmethode
55  public void run() {
56      while (true) {
57          if (!spielfeld.bewege(TeeTristSpielfeld.RUNTER)) {
58              spielfeld.berechneGrenzen();
59              spielfeld.ueberpruefeZeilen();
60              if (spielfeld.istVoll()) {
61                  spielende = true;
62                  aktuellesPanel.repaint();
63                  suspend();
64              }
65              spielfeld.wirfNaechstenStein();
66          }
67          aktuellesPanel.repaint();
68          try {
69              Thread.sleep(500);
70          } catch (InterruptedException ex) {
71              ex.printStackTrace();
72          }
73      }
74  }

```

Der Spielverlauf ist dadurch charakterisiert, dass wir im Prinzip eine endlose Schleife laufen lassen. Die Schleife kann lediglich durch zwei Ereignisse beendet werden. Eines ist dadurch gegeben, dass ein Stein, der permanent fällt, keine legale weitere Fallposition besitzt und kein neuer Stein legal nachrücken kann. Wenn ein Stein nicht

mehr fallen kann, dann werden die Grenzen mit `spielfeld.berechneGrenzen` neu berechnet und auf mögliche vollständige Zeilen in `spielfeld.ueberpruefeZeilen` geprüft und entsprechend gelöscht.

Das zweite Ereignis, um die Schleife beenden zu können, sehen wir im nächsten Programmabschnitt, wenn es um die Kommunikationsschnittstelle mit dem Benutzer geht.

```

76 // Methoden des KeyListener
77 public void keyPressed(KeyEvent e) {
78     switch(e.getKeyCode()) {
79         case KeyEvent.VK_DOWN:
80             spielfeld.bewege(TeeTristSpielfeld.RUNTER);
81             break;
82         case KeyEvent.VK_LEFT:
83             spielfeld.bewege(TeeTristSpielfeld.LINKS);
84             break;
85         case KeyEvent.VK_RIGHT:
86             spielfeld.bewege(TeeTristSpielfeld.RECHTS);
87             break;
88         case KeyEvent.VK_UP:
89             spielfeld.bewege(TeeTristSpielfeld.DREHE);
90             break;
91         case KeyEvent.VK_SPACE:
92             spielfeld.bewege(TeeTristSpielfeld.FALLEN);
93             break;
94         case KeyEvent.VK_N:
95             // Spielneustart
96             neuesSpiel();
97             break;
98         case KeyEvent.VK_Q:
99             suspend();
100            System.exit(0);
101            break;
102     }
103     aktuellesPanel.repaint();
104 }
105
106 // Das Interface KeyListener erfordert noch folgende Methoden
107 public void keyTyped(KeyEvent e) {}
108 public void keyReleased(KeyEvent e) {}
109 }
```

Die Methoden `keyPressed`, `keyTyped` und `keyReleased` müssen für das Interface **KeyListener** implementiert werden. Für unseren Fall ist aber nur eine interessant. Neben den üblichen Bewegungen des Steins (links, rechts, rotiere und ganz runter) gibt es noch die folgenden Eingaben:

**n** für starte neues Spiel

**q** beendet das Spiel und schließt die Anwendung

Hier sehen wir gleich die zweite Möglichkeit, die Endlosschleife zu beenden. Diesmal extern über eine Tastatureingabe und die Funktion `suspend`, die den **Thread** augenblicklich beendet.

### 14.2.5 Klasse TeeTristPanel

Die Visualisierung des Spielfelds übernimmt das **TeeTristPanel**. Des Weiteren gibt es eine Verbindung zur Spiellogik. Der Thread wird zu Beginn gestartet und bei der Darstellung die Information überprüft, ob das aktuelle Spiel bereits beendet wurde. Falls das Spiel zu Ende ist, wird der Text „GAME OVER!“ über das Spielfeld geblendet.

```
1 import java.awt.Color;
2 import java.awt.Dimension;
3 import java.awt.Frame;
4 import java.awt.Graphics;
5 import java.awt.Panel;
6
7 public class TeeTristPanel extends Panel {
8     private Color[] farben = { Color.RED, Color.YELLOW, Color.MAGENTA,
9                               Color.BLUE, Color.CYAN, Color.GREEN,
10                             Color.ORANGE};
11
12     private SpielThread spiel;
13     public int boxSize;
14
15     // Konstruktor
16     public TeeTristPanel(int groesse) {
17         boxSize = groesse;
18
19         // Spiellogik
20         spiel = new SpielThread(this);
21
22         setSize(groesse * spiel.getSpielfeldBreite(),
23                  groesse * (spiel.getSpielfeldHoehe() + 1));
24         setPreferredSize(this.getSize());
25         setBackground(Color.GRAY);
26         addKeyListener(spiel);
27     }
28 }
```

Die dargestellte Spielfeldgröße wird berechnet. Der Spiellogik wird mit addKeyListerner(spiel) die Kontrolle und Verwaltung der KeyEvents übertragen.

```

49     }
50 }
51
52 public void update(Graphics g) {
53     paint(g);
54 }
55 }
```

Eine entsprechende Darstellung in Abhängigkeit zu der gewählten Spielfeld- und Spielsteingröße wird in der `paint`-Methode geliefert.

#### 14.2.6 Klasse TeeTrist

Es fehlt nur noch das Fenster, indem das **TeeTristPanel** angezeigt wird.

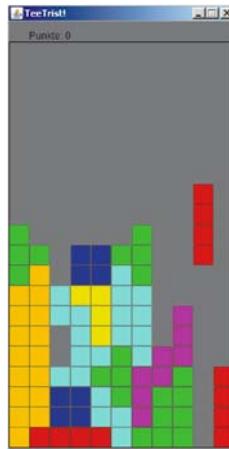
```

1 import java.awt.Frame;
2 import java.awt.event.WindowEvent;
3 import java.awt.event.WindowAdapter;
4 import java.awt.event.WindowListener;
5
6 public class TeeTrist extends Frame{
7     TeeTristPanel ttPanel;
8
9     // Konstruktor
10    public TeeTrist() {
11        addWindowListener(new WindowAdapter() {
12            public void windowClosing(WindowEvent e) {
13                System.exit(0);
14            }
15        });
16        setTitle("TeeTrist!");
17        ttPanel = new TeeTristPanel(25);
18        add(ttPanel);
19        pack();
20        setResizable(false);
21        setVisible(true);
22        ttPanel.requestFocus();
23    }
24
25    public static void main(String[] args) {
26        TeeTrist teetrist = new TeeTrist();
27    }
28 }
```

Die Methode `pack` setzt die Fenstergröße abhängig zu den verwendeten Komponenten. In diesem Fall orientiert sich die Größe ausschließlich an der Größe des Panels.

### 14.3 Spielen wir ein Spiel TeeTrist

Der Leser kann sich die Quellen von der Buchwebseite herunterladen und starten oder gleich die Appletversion ausprobieren (siehe [www.marco-block.de](http://www.marco-block.de)).



Da eine ausreichend lange **Testphase** bei der Entwicklung einer Software wichtig ist, ist ein schlechtes Gewissen beim Brechen neuer Rekorde überflüssig.

## 14.4 Dokumentation mit javadoc

Um Programme vernünftig zu kommentieren, bietet Java das Programm javadoc an. Liegen Kommentare im Programm nach einem bestimmten Schema und einer festgelegten Syntax vor, so erzeugt das Programm javadoc eine Dokumentation in HTML.

Anhand der Klasse **TeeTristPanel** wollen wir die einfache Verwendung zeigen. Kommentieren wir die Klasse:

```
...
/**
 * Die GUI-Klasse TeeTristPanel erzeugt das Spielfeld und eine
 * Instanz der Klasse Spielthread, die für die Spiellogik zuständig ist.
 * Dabei liest TeeTristPanel die aktuellen Spielfelddaten des
 * SpielfeldThread aus und erzeugt eine visuelle Darstellung der
 * aktuellen Spielsituation.
 */
public class TeeTristPanel extends Panel {
...
```

und die Methode update:

```
...
/**
 * Durch die Überschreibung der update-Methode wird ein Flackern
 * verhindert. Die überschriebene Methode hatte zuvor den Bildschirm
 * vor einem erneuten paint-Aufruf gelöscht.
 * Bei der Überlagerung ähnlicher Bilder ist kein Flackern mehr
 * festzustellen.
 * @param g Das Graphicsargument.
 */
public void update(Graphics g) {
...
```

Wenn wir die im gleichen Ordner befindliche HTML-Seite öffnen, können wir durch die Projektstruktur laufen.



## 14.5 Zusammenfassung und Aufgaben

### Zusammenfassung

Anhand eines Tetris-Projekts konnten wir die wichtigen Phasen einer Softwareentwicklung üben. Vor der eigentlichen Implementierung stehen Entwurf und Planung sowie eine umfangreiche Systembeschreibung. Die konkreten Probleme und Details wurden während der Vorstellung des Programms erläutert. Neben der Einführung von Threads wurden Rotationsmatrizen wiederholt. Da die „Testphase“ unseres Projekts selbstverständlich noch lange nicht abgeschlossen ist, wird der Leser motiviert sein, Erweiterungen und Verbesserungen vorzunehmen. Um die Softwareentwicklung zu vervollständigen wurde gezeigt, wie mit javadoc Programme professionell dokumentiert werden können.

### Aufgaben

Übung 1) Geben Sie TeeTrist einen neuen Namen und erweitern Sie Ihre Version um folgende Funktionen:

- Zweispielermodus
- Visualisierung des nachfolgenden Steins
- Änderung der Fallgeschwindigkeit der Steine

- Führen Sie verschiedene Level ein mit unterschiedlichen Feldkonfigurationen
- Verwaltung und Speicherung der HighScore
- Machen Sie ein Applet aus dieser Anwendung

Übung 2) Spielen Sie TeeTrist oder Ihre verbesserte Version, bis es Ihnen keinen Spaß mehr macht und erarbeiten Sie anschließend eine Lösung für Übung 3.

Übung 3) Entwerfen Sie ein eigenes Projektkonzept und setzen Sie es um. Vorschläge dazu wären:

- Vokabeltrainer
- Bildverarbeitungsprogramm
- Schach, Go, Sudoku
- Adress- oder allgemeine Datenverwaltung

Übung 4) Arbeiten Sie das Konzept der Threads in Java noch einmal nach. Die im TeeTrist-Beispiel verwendete Methode suspend gilt als deprecated. Der Zusatz deprecated bedeutet, dass diese Programmiertechnik bereits überholt ist und nicht mehr verwendet werden sollte. Es kommt oft vor, dass diese Techniken in späteren Versionen fehlen.

## Java – Weiterführende Konzepte

Der aufmerksame Leser ist jetzt bereits schon in der Lage, Softwareprojekte erfolgreich zu meistern. Die Kapitel und Übungsaufgaben waren so aufgebaut, dass mit einem minimalen roten Faden bereits in kürzester Zeit, kleine Softwareprojekte zu realisieren waren. Ich würde dem Leser hier nahe legen in ähnlicher Weise fortzufahren. Fangen Sie nicht gleich mit einem sehr großen Projekt an, sondern versuchen Sie sich an neuen Methoden in kleinen Projekten, das bewahrt die Übersicht.

Wenn Sie sich mit Java sicher fühlen und Ihnen Programmieren Spaß bereitet, dann suchen Sie sich Gleichgesinnte und realisieren Sie gemeinsam eigene Projekte.

Um alle Funktionen und Möglichkeiten aufzuzeigen, die Java bietet, müsste dieses Buch mehr als nur 14-Tage umfassen. In diesem Kapitel wollen wir noch weitere wichtige Aspekte ansprechen, kurz beschreiben und Literaturhinweise dazu geben.

### 15.1 Professionelle Entwicklungsumgebungen

Das Arbeiten mit einem Texteditor ist etwas rückständig, da professionelle Entwicklungsumgebungen dem Entwickler viel Arbeit abnehmen und somit die Herstellung von Software beschleunigen. Teilweise sind Konzepte wie CVS und automatische Erzeugung von Klassen- oder UML-Diagrammen integriert. Die meisten Umgebungen bieten auch automatische Ersetzungen an und überprüfen während des Eintippen die Korrektheit der Syntax.

Dem Leser sei es hier überlassen, mit welchem der zahlreichen Oberflächen er sich vertraut macht und für seine Softwareentwicklung verwendet. Vielleicht sollte zunächst einmal mit den frei erhältlichen Produkten gearbeitet werden, wie z. B. *Eclipse* [52] oder *Java NetBeans* [54].

## 15.2 Das Klassendiagramm als Konzept einer Software

Für die Planung eines Softwareprojekts ist es notwendig, Skizzen für die internen Strukturen zu erstellen. Aufgrund dieser Skizzen können Zuständigkeitsbereiche, Prozessabläufe und Datenströme festgelegt werden. Während der Konzepterstellung werden meistens viele Veränderungen vorgenommen. Um eine gute Qualität einer Software zu gewährleisten ist es notwendig, vor Beginn der Programmierung ein gut durchdachtes Konzept zu entwerfen. Für die Implementierung und das spätere Aufspüren von Fehlern ist eine statische, bildliche Darstellung der Komponenten der Software und deren Zusammenwirken hilfreich.

Es gibt verschiedene Darstellungsmöglichkeiten. Uns interessiert an dieser Stelle das Klassendiagramm, eine statische Darstellung der Software, die uns hilft, eine Implementierung vorzunehmen.

### 15.2.1 Klassendiagramm mit UML

Machen wir uns mit den Komponenten eines Klassendiagramms vertraut. Einige Darstellungen haben wir bereits in den vorhergehenden Kapiteln verwendet, ohne näher darauf einzugehen. Wir wollen in diesem Abschnitt die Konzepte so weit vorstellen, dass der Leser in der Lage ist, diese in eigenen Projekten nutzbringend zu verwenden.

Java ist eine objektorientierte Sprache. Demzufolge werden wir auch die für die Modellierung dieses Konzept umsetzen. Dem Namen Klassendiagramm können wir entnehmen, dass es sich dabei um eine Darstellung der Klassen und ihrer Beziehungen untereinander handelt.

#### 15.2.1.1 Klasse

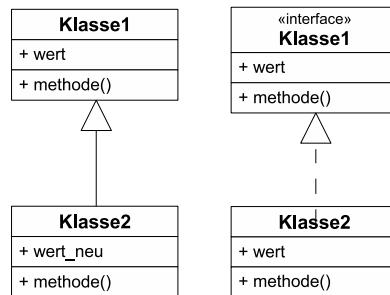
Eine Klasse wird eindeutig durch ihren Namen definiert und kann Attribute und Methoden besitzen.

Klasse
+ attributPublic # attributProtected - attributPrivate
+ methodePublic() # methodeProtected() - methodePrivate()

Zur Kennzeichnung der Modifier innerhalb dieser Darstellung werden die Symbole '+' für public, '#' für protected und '-' für privat verwendet.

### 15.2.1.2 Vererbung

Die Vererbung wird, wie wir sie in Kapitel 7 kennengelernt haben, wie folgt modelliert.



In der linken Abbildung erbt **Klasse2** von **Klasse1** das Attribut `wert`, überschreibt die Methode `methode()` und besitzt ein weiteres Attribut. Im zweiten Beispiel implementiert **Klasse2** das Interface **Schnittstelle**.

### 15.2.1.3 Beziehungen zwischen Klassen

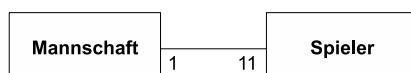
Da Klassen gekapselte Programmteile sind, besteht die Notwendigkeit darin, die Interaktionen und Abhängigkeiten der Klassen untereinander zu modellieren. Um Zusammenhänge und Beziehungen von Klassen deutlich zu machen, verbinden wir diese und ergänzen die Linien gegebenenfalls mit Symbolen oder Zahlen.



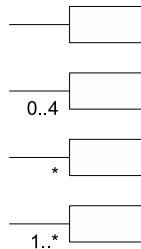
In diesem Beispiel stehen **KlasseA** und **KlasseB** in Beziehung.

#### Kardinalitäten

Wir hatten in unserem einfachen Modell eines Fußballmanagers festgelegt, dass eine Mannschaft genau aus 11 Spielern besteht. Das könnten wir wie folgt ausdrücken:

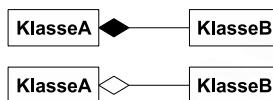


Es gibt noch weitere Möglichkeiten Kardinalitäten (den Grad einer Beziehung) anzugeben. Hier eine kleine Auswahl:

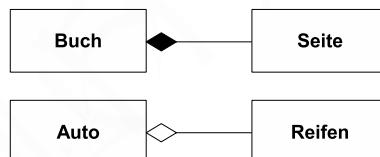


### *Aggregation und Komposition*

Neben der quantitativen Beziehung lässt sich noch eine „IstTeilvon“-Beziehung ausdrücken. Mit der Komposition, im oberen Beispiel durch die ausgefüllte Raute gekennzeichnet, wird ausgedrückt, dass **KlasseA** nur in Verbindung mit **KlasseB** existieren kann<sup>1</sup>.



Anders ist es bei der Aggregation, durch die nicht ausgefüllte Raute symbolisiert. In diesem Fall ist **KlasseB** zwar wieder Bestandteil von **KlasseA**, aber im Gegensatz zur Komposition, kann **KlasseA** auch ohne **KlasseB** existieren.



Zwei Beispiele dazu: Ein Buch besteht aus Seiten und kann ohne diese nicht existieren, aber ein Auto bleibt formal gesehen ein Auto auch ohne seine Reifen.

Der interessierte Leser sollte sich hier [42] weiter informieren oder die für das Informatikstudium wichtigen Arbeiten von Helmut und Heide Balzert durcharbeiten [33, 34].

## 15.3 Verwendung externer Bibliotheken

Es existieren viele interessante und nützliche Bibliotheken für Java. Ein paar interessante, die aufgezeigt sollen, wie vielseitig die Entwicklung mit Java sein kann, sind hier zusammengestellt.

---

<sup>1</sup> Die einzelnen Bestandteile existieren zwar unabhängig, die Komposition aber nur in Abhängigkeit ihrer Bestandteile.

### **JavaMail**

Umfangreiche Bibliothek zum Versenden und Empfangen von E-Mails via SMTP, POP3 und IMAP sowie deren sichere Vertreter POP3S und IMAPS [58].

### **SNMP4J**

Mit dieser Bibliothek lassen sich Programme entwickeln, die Netzwerkgeräte mittels des SNM-Protokolls (MPv1, MPv2c, MPv3 und gängige Verschlüsselungen) überwachen und steuern lassen können [59].

### **JInk**

Für die Verwaltung, Bearbeitung und Erkennung von handgeschriebenen, elektronischen Dokumenten [60].

### **ObjectDB**

ObjectDB ermöglicht das Speichern und Verwalten von Java-Objekten in einer Datenbank. Dabei können Objekte von wenigen Kilobytes bis hin zu mehreren Gigabytes verarbeiten werden [61].

### **Java3D**

Java muss nicht immer 2D sein. Mit dieser Bibliothek lassen sich dreidimensionale Grafiken erzeugen, manipulieren und darstellen [62].

Im Internet lassen sich dutzende solcher freien Bibliotheken finden.

## **15.4 Zusammenarbeit in großen Projekten**

Für die Zusammenarbeit in größeren Softwareprojekten ist es unerlässlich, Programme zu verwenden, die den Projektcode verwalten.

Es gibt zwei wichtige Programme in diesem Zusammenhang, CVS<sup>2</sup> und SVN. CVS (Concurrent Versions System) ist ein älteres Versionsverwaltungsprogramm und wird zunehmend von Subversion (SVN) ersetzt (hier [56] wird TortoiseSVN kostenlos angeboten).

---

<sup>2</sup> WinCVS wurde für das Forschungsprojekt FUSc# an der Freien Universität Berlin verwendet. Ein Tutorial zu CVS wurde dabei von André Rauschenbach entworfen und befindet sich hier [41].

Die Idee besteht darin, dass es einen Server (Rechner) gibt, auf dem der komplette Projektcode vorhanden ist. Jeder Programmierer kann diesen Programmcode „auschecken“, also herunterladen und verändern. Die Veränderungen und zusätzliche Informationen kann er nach getaner Arbeit anschließend wieder an den Server senden. Wenn die Programmierer an verschiedenen Programmteilen arbeiten, gibt es keine Probleme. Sollten aber zwei an demselben Programmabschnitt Veränderungen vorgenommen haben, so erkennt das Programm dieses Problem, gibt Hinweise auf die entsprechenden Abschnitte und hilft dabei, den Konflikt zu lösen.

---

# Glossar

## **Algorithmus**

Mit Algorithmen bezeichnen wir genau definierte Handlungsvorschriften zur Lösung eines Problems oder einer bestimmten Art von Problemen.

## **Applet**

Ein in Java geschriebenes Programm, das in eine Webseite integriert und in einem Webbrowser ausgeführt werden kann.

## **Applikation (Anwendungsprogramm)**

Eine allgemeine Bezeichnung für Computerprogramme, die dem Anwender eine nützliche Funktion anbieten. Oft auch als Synonym für ausführbare Programme verwendet.

## **BIT**

Der Begriff BIT ist eine Wortkreuzung aus „binary digit“ und repräsentiert die beiden binären Zustände true und false.

## **BreakPoint (Haltepunkt)**

Eine in Entwicklungsumgebungen gängige Möglichkeit, Fehler aufzuspüren. Die Haltepunkte werden innerhalb eines Programms festgelegt. Bei der Ausführung hält das Programm an dieser Stelle an und die aktuell im Speicher gültigen Variablen können ausgelesen und überprüft werden.

## **Byte-Code**

Der Java-Compiler erstellt nicht direkt den Maschinencode, der direkt auf einem Prozessor ausgeführt werden kann, sondern einen Zwischencode, den sogenannten Byte-Code. Dieser Zwischencode wird während der Laufzeit des Programms von der virtuellen Maschine übersetzt und ausgeführt.

**Casting (Typumwandlungen)**

Unter *impliziter* Typumwandlung versteht man, dass der Inhalt eines Datentyps, der weniger Speicher benötigt, ohne weitere Befehle in einen entsprechend größeren kopiert werden kann. Sollte dem Zieldatentyp weniger Speicher zur Verfügung stehen, können die Daten *explizit* umgewandelt werden.

**Compiler**

Als Compiler wird ein Programm bezeichnet, das ein Computerprogramm in eine andere Sprache übersetzt. In den meisten Fällen ist dabei die Übersetzung eines Quelltextes in eine ausführbare Datei gemeint.

**Constraints (Einschränkungen)**

Darunter versteht man Einschränkungen oder Vorgaben, z. B. für die Zuweisung von Variablen. Für Ein- und Ausgaben von Funktionen können ebenfalls Einschränkungen formuliert werden, die eingehalten werden müssen.

**Datenstruktur**

Zusammenfassung von Daten in einer geeigneten Struktur, die auch Methoden zur Verwaltung der Daten zur Verfügung stellt. Die Eigenschaften und Methoden einer Datenstruktur wirken sich auf die Geschwindigkeit des Zugriffs und die Haltung eines Datums aus.

**Debugger**

Ein Programm, das die Syntax eines Programms vor der Compilierung prüft und bei der Suche nach Fehlern behilflich ist. Gute Entwicklungsumgebungen besitzen einen integrierten Debugger.

**Deklaration**

Bezeichnet die Festlegung auf einen bestimmten Datentyp für eine Variable.

**deprecated**

Im Zuge der Weiterentwicklung einer Programmiersprache ändert sich deren Funktionalität. Dabei werden Sprachelemente, die nicht mehr benötigt werden, als deprecated bezeichnet. Sie werden nicht entfernt, damit ältere Programme funktionsfähig bleiben.

**Endlosschleife**

Ein Programm, das in einen Zyklus gerät, aus dem es ohne einen externen Programmabbruch nicht mehr herauskommt.

**Entwicklungsumgebung**

Unter Entwicklungsumgebungen versteht man grafische Oberflächen, die für eine professionelle Softwareentwicklung in Java oder anderen Programmiersprachen unerlässlich sind.

**Funktionen oder Methoden**

Gruppe von Anweisungen, die Eingaben verarbeiten können und einen Rückgabewert berechnen.

**Garbage Collector**

Ein Programm, das sich während der Laufzeit einer Java-Anwendung um die automatische Freigabe nicht mehr benötigten Speichers kümmert.

**Ghosttechnik**

Um rechenintensive Grafikausgaben zu beschleunigen, werden Ausgaben zunächst im Hintergrund berechnet und anschließend ausgegeben. Bei der eigentlichen Ausgabe auf einem Monitor wird so das System entlastet.

**GUI**

Abkürzung für graphical user interface. Bezeichnet eine grafische Oberfläche, die als Schnittstelle zwischen Programm und Benutzer fungiert.

**Implementierung**

Umsetzung einer abstrakten Beschreibung eines Algorithmus in eine konkrete Programmiersprache.

**Klasse**

Zusammenfassung von Eigenschaften und Methoden bzw. Bauplan eines Objekts. Wichtigster Grundbaustein der objektorientierten Programmierung.

**Konsole, Kommandozeile oder Eingabeaufforderung**

Die Kommandozeile wird bei den meisten Betriebssystemen im Textmodus angeboten und dient der Steuerung von Programmen und der Navigation innerhalb des physischen Speichers.

**Konstanten**

Konstanten sind Variablen, deren zugewiesener Inhalt nicht mehr geändert werden kann.

**Konventionen**

Konventionen (im Bereich Programmierung) sind keine formal festgelegten Regeln, sondern Vereinbarungen, an denen sich ein Großteil der Entwickler hält. Es dient der verbesserten Lesbarkeit eines Programms.

**Modularisierung**

Ein Programm in Programmabschnitte zu unterteilen, fördert die Übersicht und vereinfacht das Aufspüren von Fehlern.

**Objektorientierung**

Ein Programmierkonzept, bei dem Daten nach Eigenschaften und möglichen Methoden oder Operationen klassifiziert werden. Diese Eigenschaften und Methoden können vererbt werden. Eine Klasse stellt dabei den Bauplan eines Objekts fest.

**Package**

Bibliotheken in Java werden in Paketen organisiert. Diese Pakete definieren sinnvolle Gruppen von Klassen und Funktionen, die eine logische Struktur zur Lösung spezifischer Aufgaben liefern.

**plattformunabhängig**

Softwaresysteme, wie z. B. Java, die nach der Compilierung auf verschiedenen Betriebssystemen (Plattformen) laufen, nennt man plattformunabhängig.

**Programm**

Eine Abfolge von Anweisungen, die auf einem Computer zum Laufen gebracht werden können, um eine bestimmte Funktionalität zu liefern.

**Pseudocode**

Um unabhängig von Programmiersprachen einen Algorithmus zu formulieren, wird dieser im sogenannten Pseudocode entworfen. Pseudocode ähnelt der natürlichen Sprache mehr als einer formalen Programmiersprache, es können sogar ganze Sätze verwendet werden. Dabei gibt es keine formalen Vorgaben.

**Schlüsselwörter**

Schlüsselwörter sind reservierte Befehle in einer Programmiersprache, die für Variablenbezeichnungen nicht mehr zur Verfügung stehen.

**Syntax**

Unter der Syntax versteht man die formale Struktur einer Sprache. Sie beschreibt die gültigen Regeln und Muster, mit denen ein Programm entworfen werden kann.

**Syntaxhervorhebung**

Professionelle Entwicklungsumgebungen können Schlüsselwörter einer Sprache farblich hervorheben und damit die Lesbarkeit eines Programms deutlich erhöhen.

**terminieren**

Ziel bei der Programmierung ist es immer, ein Programm zu schreiben, das in keine Endlosschleife gelangt. Ein Programm sollte immer terminieren.

**textsensitive Sprache**

Der Begriff textsensitiv verdeutlicht, dass ein Unterschied zwischen Groß- und Kleinschreibung innerhalb einer Sprache gemacht wird. Bei der Deklaration von Variablen ist beispielsweise darauf zu achten.

**Thread**

Ein Prozess oder Programmteil, der sich im Prinzip autonom verhält, aber nur so lange funktionieren kann, bis er gestoppt wird oder das Hauptprogramm, das diesen erzeugt und gestartet hat, beendet wird.

**Variablen**

Der zur Verfügung stehende Speicher einer Variablen wird bei der Deklaration durch einen Datentyp definiert. Die Variable kann anschließend als Platzhalter für Werte dienen.

**virtuelle Maschine**

Die Java Virtual Machine (JVM) führt den vom Java-Compiler erstellten Byte-Code aus und ist die Verbindung zwischen Betriebssystem und Java-Programm.

**Zuweisung**

Nachdem der Datentyp einer Variablen festgelegt wurde, kann ihr ein Wert zugewiesen werden. Diesen Vorgang nennt man Zuweisung.

---

## Literatur

1. Abts D (1999) Grundkurs Java. Vieweg, Braunschweig/Wiesbaden
2. Solymosi A, Schmiedecke I (1999) Programmieren mit Java. Vieweg, Braunschweig/Wiesbaden
3. Rauh O (1999) Objektorientierte Programmierung in JAVA. Vieweg, Braunschweig/Wiesbaden
4. Schiedermeier R (2005) Programmieren in Java. Pearson Studium, München
5. Barnes DJ, Kölking M (2006) Java lernen mit BlueJ. Pearson Studium, München
6. Bishop J (2003) Java lernen. Pearson Studium, München
7. Schader M, Schmidt-Thieme L (2003) JAVA – Eine Einführung. Springer, Berlin Heidelberg New York
8. Steyer R (2001) Java 2 – Professionelle Programmierung mit J2SE Version 1.3. Markt+Technik, München
9. Willms R (2000) Java – Programmierung Praxisbuch. Franzis, Poing
10. Lewis J, Loftus W (2000) Java Software Solutions – Foundations of Program Design. Addison-Wesley, Boston
11. Sedgewick R (2003) Algorithmen in Java, Teile 1-4. Pearson Studium, München
12. Burger W, Burge MJ (2006) Digitale Bildverarbeitung – Eine Einführung mit Java und ImageJ. Springer, Berlin Heidelberg New York
13. Lea D (2000) Concurrent Programming in Java Second Edition – Design Principles and Patterns. Addison-Wesley, Boston
14. Magee J, Kramer J (1999) CONCURRENCY – State Models & Java Programs. Wiley, New York
15. Weiss MA (1999) Data Structures & Algorithm Analysis in JAVA. Addison-Wesley, Boston
16. Ottmann T, Widmayer P (2002) Algorithmen und Datenstrukturen. Spektrum Akademischer Verlag, Heidelberg Berlin
17. Schöning U (2001) Algorithmitk. Spektrum Akademischer Verlag, Heidelberg Berlin
18. Heun V (2000) Grundlegende Algorithmen. Vieweg, Braunschweig/Wiesbaden
19. Schöning U (1997) Algorithmen – kurz gefasst. Spektrum Akademischer Verlag, Heidelberg Berlin
20. Cormen TT, Leiserson CE, Rivest RL (2000) Introductions To Algorithms. MIT Press, Cambridge
21. Diestel R (1996) Graphentheorie. Springer, Berlin Heidelberg New York

22. Clark J, Holton DA (1994) Graphentheorie – Grundlagen und Anwendungen. Spektrum Akademischer Verlag, Heidelberg Berlin
23. Selzer PM, Marhöfer RJ, Rohwer A (2004) Angewandte Bioinformatik – Eine Einführung. Springer, Berlin Heidelberg New York
24. Duda RO, Hart PE, Stork DG (2001) Pattern Recognition. Wiley, New York
25. Webb A (1999) Statistical Pattern Recognition. Arnold, London Sydney Auckland
26. Block M, Rauschenbach A, Buchner J, Jeschke F, Rojas R (2005) Das Schachprojekt FUSc#. Technical Report B-05-21, Freie Universität Berlin
27. Shannon CE (1950) Programming a Computer for Playing Chess. Philosophical Magazine, Ser.7, Vol.41, No.314
28. Newell A, Shaw JC, Simon HA (1958) Chess playing programs and the problem of complexity. IBM Journal of Research and Development, 4(2), 320-335, 1958
29. Plaat A (1996) Research Re: search & Re-search. PhD thesis, Universität Rotterdam
30. Needleman S, Wunsch C (1970) A general method applicable to the search for similarities in the amino sequence of two proteins. J Mol Biol. 48(3):443-453
31. Ramírez M, Block M, Rojas R (2006) New Robust Binarization Approach in Letters. CONCIBE, Guadalajara/Mexico
32. Comay O, Intrator N (1993) Ensemble Training: some Recent Experiments with Postal Zip Data. Proceedings of the 10th Israeli Conference on AICV
33. Balzert Helmut (1997) Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, Heidelberg Berlin
34. Balzert Heide (2005) Lehrbuch der Objektmodellierung. Spektrum Akademischer Verlag, Heidelberg Berlin
35. Zuser W, Biffl S, Grechenig T, Köhle M (2001) Software Engineering mit UML und dem Unified Process. Pearson Studium, München
36. Münz S (2005) Professionelle Websites. Addison-Wesley, Boston
37. Ullensboom C (2006) Java ist auch eine Insel. Galileo Computing, Onlineversion, siehe: <http://www.galileocomputing.de/1082?GPP=opjV>
38. Java Sun-Webseite: <http://java.sun.com/docs/books/tutorial/java/nutsandbolts>
39. Jama-Webseite: <http://math.nist.gov/javanumerics/jama/>
40. Conway's Game of Life: [http://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway's_Game_of_Life)
41. CVS-Tutorial des FUSc#-Projekts: <http://page.mi.fu-berlin.de/~fusch>
42. UML: <http://www.uml.org/>
43. SelfHTML-Homepage: <http://de.selfhtml.org/>
44. Wikipedia Webseite: <http://de.wikipedia.org/wiki/Hauptseite>
45. Wikipedia BlackJack-Regeln: [http://de.wikipedia.org/wiki/Black\\_Jack](http://de.wikipedia.org/wiki/Black_Jack)
46. Wikipedia Algorithmus: <http://de.wikipedia.org/wiki/Algorithmus>
47. Wikipedia Brute Force: <http://de.wikipedia.org/wiki/Brute-Force-Methode>
48. FUSc#-Schachspielserver: <http://www.fuschmotor.de.vu/>
49. Wikipedia Definition Thread: [http://de.wikipedia.org/wiki/Thread\\_%28Informatik%29](http://de.wikipedia.org/wiki/Thread_%28Informatik%29)
50. Java-Sun Internetseite: <http://java.sun.com/>
51. Java-Tutor Internetseite: [http://www.java-tutor.com/java/links/java\\_ides.htm](http://www.java-tutor.com/java/links/java_ides.htm)
52. Eclipse Internetseite: <http://www.eclipse.org/>
53. JCreator Internetseite: <http://www.jcreator.com/>
54. Java-NetBeans Internetseite: <http://www.netbeans.org/>
55. Borland JBuilder Internetseite: <http://www.borland.com/>
56. Tortoise-SVN Internetseite: <http://tortoisesvn.tigris.org/>
57. Java API: <http://java.sun.com/javase/6/docs/api/>
58. JavaMail Internetseite: <http://java.sun.com/products/javamail/>

59. SNMP4J Internetseite: <http://www.snmp4j.org/>
60. JInk: <http://mathfor.mi.fu-berlin.de>
61. ObjectDB: <http://www.objectdb.com/>
62. Java3D: <https://java3d.dev.java.net/>

---

## Sachverzeichnis

- Abbruchkriterium 197
- Abhangigkeiten 65
- abs 104
- abstrakte Klasse 88
- abstrakte Methode 88
- abstraktes Rechnermodell 160
- Abstraktionsniveau 160
- Absturz 68
- acos 104
- Adressverweise 90
- Aggregation 236
- Algorithmik 153
- Algorithmus 153
  - Alpha-Beta 205
  - Glossar 239
  - k-means 195
  - k-nachste Nachbarn 192
  - MinMax 202
  - Needleman-Wunsch-Algorithmus 166
  - Sortieralgorithmen 162
- Alpha-Beta-Algorithmus 156, 205
- Alpha-Wert 184
- Anforderungsanalyse 211
- Anweisungen
  - markierte 38
- Apfelmannchen 173, 176
- Applet 2, 139, 141
  - destroy 141
  - flackerndes 146
  - Glossar 239
  - init 141
  - Initialisierung 141
  - start 141
- stop 141
- Appletprogrammierung 139
- Appletviewer 141
- Applikation 2
  - Glossar 239
  - in Applet 143
- ArithmeticeException 65
- arithmetisches Mittel 186
- Array 53, 54, 108
  - multidimensionale 55
  - zweidimensional 53
- ArrayIndexOutOfBoundsException 47, 54
- asin 104
- atan 104
- Attribute 90
- Ausgabe 41
- Ausgabefunktion 58
- Ausreier 192
- average-case-Analyse 160
- AWT 123, 179
- Bakterienkulturen 57
- Bedingung 34
- Benoit Mandelbrot 177
- Berechnungen
  - fehlerhafte 68
- Bewertungsfunktion 156, 202, 204
- Bibliothek
  - erstellen 120
  - JAMA 118
  - java.awt 123
  - java.lang 105
  - java.util 104

- Java3D 237
- JavaMail 237
- javax 179
- JInk 237
- ObjectDB 237
- SNMP4J 237
- verwenden 101
- Bildbetrachtungsprogramm 180
- Bilder
  - bearbeiten 184
  - binarisieren 186
  - Grauwertbild 185
  - invertieren 184
  - laden und anzeigen 128
  - laden und speichern 179
- Bildmatrix 192
- Bildschirmauflösung 124
- Bildverarbeitung 171
- Binarisierung
  - global 186
  - lokal 186
- Bioinformatik 166
- BIT
  - Glossar 239
- Bitmap 179
- Boolean 96
- boolean 11
- Brasilien 85
- break 37
- Breakpoints 71
  - Glossar 239
  - konditionale 71
- Brute Force 156, 161, 202
- BubbleSort 163
- Buffer 102
- BufferedReader 49
- Button 133, 136
- Byte 96
- byte 15
- Byte-Code
  - Glossar 239
- call by reference 91
- call by value 91
- Casting 16, 18
  - Glossar 239
- ceil 104
- char 14
- Character 96
- Christian Wunsch 166
- CLASSPATH 4, 5
- Claude Elwood Shannon 156, 202
- Clusteringverfahren 195
- Collatz-Problem 169
- Color 126, 127
- Comays Zifferndatenbank 192
- Compiler 160
  - Glossar 240
- Computer 160
- Conquer-Schritt 160
- Constraints 63
  - Glossar 240
- continue 39
- Conway's Game of Life 53, 56
- Copy-Konstruktur 93, 216, 217
- cos 104
- CVS 233, 237
- Daten
  - laden 45
  - lesen 47
  - speichern 45, 49
  - von Konsole einlesen 49
- Datenbank 47
- Datenhaltung 211
- Datenstrom 234
- Datenstruktur 53
  - Array 53
  - einfach 53
  - Glossar 240
  - Matrix 53
  - Vector 108
- Datentypen
  - ganzzahlige 15
  - primitive 7, 16, 53, 95
  - Größenverhältnisse 18
- Datenverlust 17
- Datum 53
- Debuggen 63
  - zeilenweise 71
- Debugger 70
  - Glossar 240
- Defaultkonstruktur 93
- Deklaration 9
  - Glossar 240
- deprecated 231
  - Glossar 240
- Deutschland 85

- Diashow 129
- Divide-and-Conquer 160, 165
- Divide-Schritt 160
- Division
  - ganzzahlig 14, 70
  - gebrochen 16
- do-while 36
- Dokumentation 229
- Double 96
- double 16
- Drei-Schichten-Architektur 211
- Dynamische Programmierung 158, 166
  - Beispiel 158
- Effizienz 153, 155
- Eingabeaufforderung 4
  - Glossar 241
- Eingabeparameter 42, 64
- Einlesemethode 191
- Einschränkungen 63
- Elementaroperationen 160
- Endlosschleife 36
  - Glossar 240
- Entwicklungsumgebungen 4, 24, 71, 233
  - Glossar 240
- Entwurfs-Techniken 154
- Entwurfssphase 211
- Ereignisbehandlung 144
- Erkennungsrate 192, 194, 198, 201
- Etikett 189
- Euklidischer Abstand 193
- euklidischer Abstand 104
- Eventtypen 131
- evolutionäre Stammbäume 74
- Exceptions 65
- exp 104
- Expectation-Maximization 196, 197
- Expectation-Schritt 196
- Fakultät 64, 155, 160
- false 11
- Farbmischung
  - additiv 171
  - subtraktiv 171
- Fehler 63
  - Division durch 0 65
- Fehlerbehandlungen 63
- Fehlerklasse 65
- Fehlerkonzept 68
- Fehlermeldung 25, 47
- Fehlerquelle 65
- Fenster
  - erzeugen 123
  - zentrieren 124
- Fenstereigenschaften 124
- Fensterereignisse 130
- Fenstermanagement 123
- Fibonacci-Zahlen 156, 158
- final 11
- Flackern 146, 149
- Float 96
- float 16
- floor 104
- for 34, 37
- Fraktale 175
- Frame 123, 139
- Freie Universität Berlin 140
- Freundschaftsspiel 85
- Fußballmanager 73
- Funktionen 41, 92, 101
  - Eingabeparameter 42
  - Glossar 240
- Ganzzahlen 14
- Garbage Collector 94
  - Glossar 241
- Gauß-Verteilung 105
- Generalisierung 74
- Generation 57
- Gesichtererkennung 209
- get-set-Funktionen 76
- Ghosttechnik 147
  - Glossar 241
- Gleiter 61
- Gleitkommazahlen 16
- Grafikausgabe 123
- Graphentheorie 158
- Graphics 125
- Grauwert 172
- Greedy 157
  - Beispiel 158
- Grundfarben 171, 184
- GUI 211
  - Glossar 241
- GUI-Elemente 132
- Halbzug 204
- handgeschriebene Ziffern 195

- Hauptvariante 203
- Helligkeitswert 184, 185
- Hexadezimaldarstellung 102
- Hintergrund 126
- HTML 139
- if 31, 95
- Image 129
- Imaginärteil 174
- Implementierung 57
  - Glossar 241
- Importierung 118
- Index 47, 54
- Initialisierung 55
- InputStreamReader 49
- InsertionSort 154, 162
- Instanz 77, 103
- Instanzmethoden 90, 91, 94, 96
- Instanzvariablen 90, 91, 94
- int 14, 105
- Integer 96, 101
- Intensität 171
- Interaktion 49
- Interface 82, 88
  - ActionListener 134
  - Freundschaftsspiel 82
  - MouseListener 136
  - WindowListener 130
- Intervall 104
- IOException 49
- Iterationen 177
- JAMA 118
- Java
  - API 101, 125
  - Byte-Code 5
  - Compiler 16, 24
  - Development-Kit (JDK) 3
  - Erstellung eines Programms 24
  - Funktionen 41
  - Installation 3
  - Motivation 1
  - Operationen 11
  - primitive Datentypen 8
  - Programme 24
- Java-Plugin 140
- java.awt 123
- java.lang 101, 105
- java.util 104
- javadoc 229
- javax 179
- JDK 3
- k-means 195
  - allgemeine Formulierung 197
  - Implementierung 198
- k-nn 192
- Künstliche Intelligenz 189
- Kardinalitäten 235
- Kartenspiel 106
- Kasparov 157
- Klasse 29, 41, 91, 92, 94
  - BufferedImage 177
  - BufferedReader 49
  - Button 136
  - Color 172
  - DoubleListe 103
  - Fraktal 176
  - Frame 179
  - Glossar 241
  - Graphics 178
  - ImageConsumer 184
  - ImageFilter 184
  - ImageIO 179
  - ImageObserver 178, 184
  - ImageProducer 184
  - innere 132
  - innere, anonyme 132
  - InputStreamReader 49
  - KMeans 198
  - Knn 192, 198
  - KomplexeZahl 174
  - lokale 132
  - Math 103
  - Random 105
  - SpielThread 223
  - StringBuffer 102
  - TeeTrist 228
  - TeeTristBox 215
  - TeeTristPanel 227
  - TeeTristSpielfeld 219
  - TeeTristStein 216
  - TextField 134
  - TicTacToe 205
- Klassen 75, 101
- Klassenattribut 76
- Klassendiagramm 234
  - UML 234

- Klassenkonzept
  - einfach 29
  - erweitert 73
- Klassenmethoden 94, 104
- Klassenvariablen 94
- Klassifizierung 74, 189, 191
- Klassifizierungsalgorithmen 192
- Knopfdruck 134
- Kochrezept 20
- Kommandozeile 46
  - Glossar 241
- Kommentar 15, 30, 64
- Kommentierung 48
- Kommunikationsschnittstelle 226
- Komplexe Zahlen 174
- Komplexität 163, 164
  - des Schachspiels 157
- Komposition 236
- Komprimierung 195
- Konsole 4
  - Glossar 241
- Konsolenausgabe 71
- Konstanten 11
  - Glossar 241
- Konstruktoren 78, 93, 95
- Konventionen 29
  - Glossar 241
- Konvertierungsmethoden 95
- Konzept 63
  - das richtige 63
- Konzepte 2, 89
  - grundlegende 21
- Kramnik 157
- Kurzschriftweise 15
- Label 133, 190
- Laufzeitanalyse 160
  - Beispiel 160
- Layoutmanager 133
  - FlowLayout 133
- Leibniz 68
  - Leibniz-Reihe 68
- Lernen
  - Bottom-Up 2
  - Top-Down 2
- Lineare Algebra 118
- Lineare Gleichungssysteme 118, 120
- Liste 54
- literale Erzeugung 55
- log 104
- Long 96
- long 15, 105
- Lottoprogramm 105
- magic numbers 183
- main 42
- Marke 38
- Matrix 57, 119, 183
  - Determinante 118, 119
  - Eigenwerte 118
  - Rang 118
- Matrizen 55
- Mausereignisse 136
- max 104
- Maximization-Schritt 196
- MediaTracker 129
- Mehrheitsentscheid 195, 196
- Memoisierung 158
  - Beispiel 159
- Meta-Informationen 140
- Methode 41, 90
- min 104
- MinMax-Algorithmus 156, 202
- Mittelwert 105, 192
- Modifizierer 76
- Modularisierung 63
  - Glossar 241
- MouseAdapter 136
- mouseDragged 150
- mousePressed 151
- Mustererkennung 189
- Nachbarschaft 58
- Namensgleichheit 92
- Needleman-Wunsch-Algorithmus 166
- NumberFormatException 65
- Object 101
- Objekt 77, 79, 90, 92, 94, 103
- Objektorientierung 73, 174
  - Glossar 241
- Operationen 11, 14
  - DIV 14, 16, 19
  - logische 12
  - logisches ENTWEDER-ODER 13
  - logisches NICHT 13
  - logisches ODER 13
  - logisches UND 12

- logisches XOR 13
- MOD 14, 16
- relationale 14
- Optimierungsverfahren 197
- Outlier 192
  
- Package
  - Glossar 242
  - JAMA 118
  - java.awt 123
  - java.lang 105
  - java.util 104
  - Java3D 237
  - JavaMail 237
  - javax 179
  - JInk 237
  - ObjectDB 237
  - SNMP4J 237
- Packages 101, 120
- Parallelität 23
- Parameter 46
- Partieverläufe 204
- PATH 3, 5, 120
- Pattsituation 193
- Performancesteigerung 155
- Petrischale 57
- pi 11, 68
- Pivotelement 165
- Pixel 172
- plattformunabhängig 1
  - Glossar 242
- Populationsdynamik 62
- Postleitzahl 193
- pow 104
- Präsentation 211
- Primzahlen 44, 51, 161
- Prinzipien der Programmentwicklung 7
- private 76
- Problemlösung 153
- Programm 153
  - als Kochrezept 20
  - Block 29
  - Dateiname 24
  - Glossar 242
  - Kombinationen 23
  - Konventionen 10
  - Lesbarkeit 42, 48
- Programmablauf 21
  - Mehrfachschleifen 22
- Mehrfachverzweigung 22, 32
- Schleife 22, 33
- sequentiell 21, 30
- Sprung 22, 37
- Verzweigung 21, 31
- Programmeingaben
  - externe 46
- Programmentwicklung
  - Techniken 153
- Programmerstellung 20
  - Methoden 20
- Programmiersprache 160
- Projekt 233
  - Apfelmännchen 173
  - Black Jack 106
  - Conway's Game of Life 56
  - Fußballmanager 73
  - Lottoprogramm 105
  - Tetris 211
  - TicTacToe 202
  - Ziffernerkennung 189
- protected 88
- Protein-Sequenzen 166
- Prototypen 195
- Prozessablauf 234
- Pseudocode 24
- Pseudonym
  - Glossar 242
- public 41, 76
- Punktanhäufungen 195
- Punktnotation 90
- Punktwolke 195
  
- QuickSort 160, 165
  
- Räuber-Beute-Simulation 62
- Rückgabewert 42
- Rauschen 193
- Realteil 174
- rechenintensiv 147
- Redundanz 41
- Referenz 90, 93, 95
- Referenzmenge 193
- Referenzvariablen 89, 91, 97
- Rekursion 154
  - Beispiel 155, 156
- Rekursionsformel 158
- Rekursionsprozess 159
- Repräsentanten 195

- RGB-Farbmodell 127, 171, 184
- Rotationsgleichung 219
- round 104
- Rundungsfehler 16
- Satz des Pythagoras 174
- Saul Needleman 166
- Schach 204
- Schachmotor 140
- Schachprogramm
  - Deep Blue 157
  - Deep Fritz 157
  - Deep Junior 157
  - FUSch 140
  - X3D Fritz 157
- Schachprogrammierung 164, 204
- Schachspieler 140
- Schema
  - konzeptuell 211
- Schlüsselwörter 10
  - Glossar 242
  - reservierte 10
- Schleife 22, 37, 47, 55, 68
  - do-while 36
  - Endlosschleife 36
  - for 34
  - innere 39
  - verschachtelt 22
  - while 35
- Schnittstelle 82
- Schrift 126
- Schrittgröße
  - konstant 34
- Schwerpunkt 192
- SDK 3
- Seiteneffekte 63
- Selbstähnlichkeit 173
- Selbststudium 2
- sequentiell 30
- Short 96
- short 15
- Sieb des Eratosthenes 161
- Signatur 92
  - parameterlos 93
- sin 104
- Sonderzeichen
  - deutsche 25
- Sortieralgorithmen 162
- BubbleSort 163
- InsertionSort 162
- QuickSort 165
- Sortierung 154
- Speicherlaufwand 160
- Speicherbedarf 8
- Speichermanagement 94
- Spezialisierung 74, 75, 77
- Spielbrettmatrix 206
- Spiellogik 211
- Spieltheorie 156, 202
- Sprache
  - textsensitive 10
- Sprungmarke 39
- sqrt 104
- Standardabweichung 105
- Standardbibliotheken 101
- Standardfensterklasse 130
- Standardnormalverteilung 105
- static 41, 94
- statische Attribute 94
- statische Methoden 94
- String 31, 96, 101
  - Liste 46
- Suchtiefe 157
  - begrenzt 204
  - unbegrenzt 204
- SVN 237
- Swing 179
- switch 31, 32
- Symbole 7, 14
- syntaktisch 25
- Syntax 66, 79, 90
  - Glossar 242
- Syntaxhervorhebung
  - Glossar 242
- System.exit 131
- System.in 49
- Systembeschreibung 211
- Systemzeit 106
- Türme von Hanoi 169
- tan 104
- Tastenkombination 124
- Teilungsoperatoren 16
- terminale Stellung 204
- terminieren 23
  - Glossar 242
- Testphase 229
- Tetris 211

- Implementierung 215
- Interaktion 212
- Klassendiagramm 213
- Spielregeln 211
- Testphase 229
- Textausgaben 126
- TextField 134, 144
- textsensitive Sprachen
  - Glossar 242
- this 91
- Thread 23, 212
  - Definition 224
  - Glossar 242
- TicTacToe 156, 202, 205
- Trainingsdaten 190
  - einlesen 190
- Trainingsdatenbank 195
- Trainingsroutine 199
- Trainingsvektoren 192
- Transformation
  - linear 218
- Transparenz 184
- Transpositionstabellen 205
- true 11
- try-catch 66
  - einfach 66
  - mehrfach 67
- type 48
- Typsicherheit 1
- Typumwandlungen 16
  - explizite 16
  - implizite 18
- Überladen 92
  - Konstruktoren 92
- Umgebungsvariablen 3
- Umkodierung 195
- UML 233
  - Abhängigkeiten 235
  - Aggregation 236
  - Beziehungen zwischen Klassen 235
  - Kardinalitäten 235
  - Klasse 234
  - Komposition 236
  - Modifier 234
  - Vererbung 235
- unüberwachtes Lernen 195
- Ursprung 192
- Variablen 9, 63
  - Bezeichnung 9
  - deklarieren 9
  - Glossar 243
  - Vergleich zu Konstanten 11
  - Wert zuweisen 9
- Vector 108
- Vektor 195, 218
  - 192-dimensional 191
- Vererbung 73, 75
- Vererbungsstruktur 102
- Vererbungsvariante 82
- Vergleich von Algorithmen 160
- Vergleichsmechanismus 103
- Vergleichsoperatoren 14
- Verwendungsschema 48
- virtuelle Maschine 243
- Visualisierung 193
- void 42
- Wahrheitswerte 7
- Wahrscheinlichkeitsmaß 104
- Wahrscheinlichkeitsverteilung 84
- Wartefunktion 126
- Wellenlängen 171
- Wertetabelle 12, 13
- while 35
- WindowAdapter 131
- worst-case-Analyse 160
- Wrapperklassen 95, 103
  - Boolean 96
  - Byte 96
  - Character 96
  - Double 96
  - Integer 96
  - Long 96
  - Short 96
- Zählvariable 37
- Zahlen 8
  - gebrochene 16
- Zeichen 7, 14
- Zeichenelemente 126
- Zeichenfunktionen 125
- Zeichenkette 31, 96
  - vergleichen 97
- Zelle 56
- Zellkonstellationen 60
- Zellulärer Automat 56

- Ziffernerkennung 195  
Zufallszahlen 57, 60, 104, 106  
  ganzzahlig 105  
  gebrochene 105  
Zug 204
- Zugmöglichkeiten 205  
Zuständigkeitsbereich 234  
Zuweisung 9  
  Glossar 243  
zyklische Muster 61