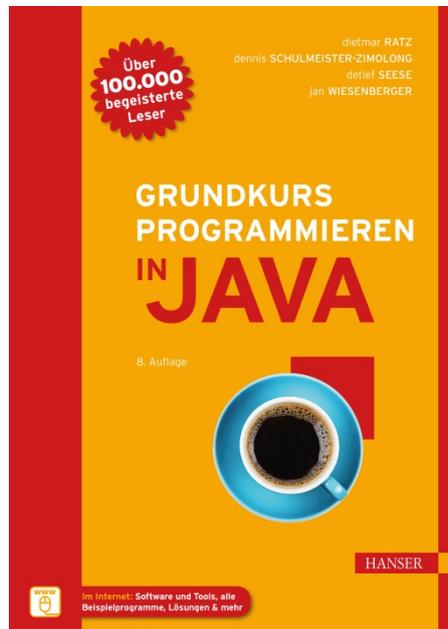


# HANSER



## Leseprobe

zu

## „Grundkurs Programmieren in Java“

von Dietmar Ratz, Dennis Schulmeister-Zimolong,  
Detlef Seese und Jan Wiesenberger

ISBN (Buch): 978-3-446-45212-1

ISBN (E-Book): 978-3-446-45384-5

Weitere Informationen und Bestellungen unter  
<http://www.hanser-fachbuch.de/978-3-446-45212-1>

sowie im Buchhandel

© Carl Hanser Verlag, München

# Inhaltsverzeichnis

<b>Vorwort</b> . . . . .	<b>17</b>
<b>Einleitung</b> . . . . .	<b>19</b>
Java – mehr als nur kalter Kaffee? . . . . .	19
Java für Anfänger – das Konzept dieses Buches . . . . .	20
Zusatzmaterial und Kontakt zu den Autoren . . . . .	21
Verwendete Schreibweisen . . . . .	22
<b>I Einstieg in das Programmieren in Java</b> . . . . .	<b>23</b>
<b>1 Einige Grundbegriffe aus der Welt des Programmierens</b> . . . . .	<b>25</b>
1.1 Computer, Software, Informatik und das Internet . . . . .	25
1.2 Was heißt Programmieren? . . . . .	27
1.3 Welche Werkzeuge brauchen wir? . . . . .	30
<b>2 Aller Anfang ist schwer</b> . . . . .	<b>33</b>
2.1 Mein erstes Programm . . . . .	33
2.2 Formeln, Ausdrücke und Anweisungen . . . . .	34
2.3 Zahlenbeispiele . . . . .	35
2.4 Verwendung von Variablen . . . . .	36
2.5 „Auf den Schirm!“ . . . . .	36
2.6 Das Programmgerüst . . . . .	37
2.7 Eingeben, übersetzen und ausführen . . . . .	39
2.8 Übungsaufgaben . . . . .	40
<b>3 Grundlagen der Programmierung in Java</b> . . . . .	<b>41</b>
3.1 Grundelemente eines Java-Programms . . . . .	41
3.1.1 Kommentare . . . . .	43
3.1.2 Bezeichner und Namen . . . . .	45
3.1.3 Literale . . . . .	46
3.1.4 Reservierte Wörter, Schlüsselwörter . . . . .	47
3.1.5 Trennzeichen . . . . .	47

3.1.6	Interpunktionszeichen . . . . .	48
3.1.7	Operatorsymbole . . . . .	49
3.1.8	<b>import</b> -Anweisungen . . . . .	49
3.1.9	Zusammenfassung . . . . .	50
3.1.10	Übungsaufgaben . . . . .	51
3.2	Erste Schritte in Java . . . . .	52
3.2.1	Grundstruktur eines Java-Programms . . . . .	52
3.2.2	Ausgaben auf der Konsole . . . . .	53
3.2.3	Eingaben von der Konsole . . . . .	55
3.2.4	Schöner programmieren in Java . . . . .	55
3.2.5	Zusammenfassung . . . . .	56
3.2.6	Übungsaufgaben . . . . .	57
3.3	Einfache Datentypen . . . . .	57
3.3.1	Ganzzahlige Datentypen . . . . .	57
3.3.1.1	Literalkonstanten in anderen Zahlensystemen . . . . .	59
3.3.1.2	Unterstrich als Trennzeichen in Literalkonstanten . . . . .	60
3.3.2	Gleitkommatypen . . . . .	61
3.3.3	Der Datentyp <b>char</b> für Zeichen . . . . .	63
3.3.4	Zeichenketten . . . . .	63
3.3.5	Der Datentyp <b>boolean</b> für Wahrheitswerte . . . . .	64
3.3.6	Implizite und explizite Typumwandlungen . . . . .	64
3.3.7	Zusammenfassung . . . . .	65
3.3.8	Übungsaufgaben . . . . .	66
3.4	Der Umgang mit einfachen Datentypen . . . . .	66
3.4.1	Variablen . . . . .	67
3.4.2	Operatoren und Ausdrücke . . . . .	70
3.4.2.1	Arithmetische Operatoren . . . . .	72
3.4.2.2	Bitoperatoren . . . . .	74
3.4.2.3	Zuweisungsoperator . . . . .	76
3.4.2.4	Vergleichsoperatoren und logische Operatoren . . . . .	76
3.4.2.5	Inkrement- und Dekrementoperatoren . . . . .	79
3.4.2.6	Priorität und Auswertungsreihenfolge der Operatoren . . . . .	79
3.4.3	Allgemeine Ausdrücke . . . . .	81
3.4.4	Ein- und Ausgabe . . . . .	81
3.4.4.1	Statischer Import der IOTools-Methoden . . . . .	83
3.4.5	Zusammenfassung . . . . .	84
3.4.6	Übungsaufgaben . . . . .	84
3.5	Anweisungen und Ablaufsteuerung . . . . .	88
3.5.1	Anweisungen . . . . .	88
3.5.2	Blöcke und ihre Struktur . . . . .	89
3.5.3	Entscheidungsanweisung . . . . .	89
3.5.3.1	Die <b>if</b> -Anweisung . . . . .	89
3.5.3.2	Die <b>switch</b> -Anweisung . . . . .	91

3.5.4	Wiederholungsanweisungen, Schleifen . . . . .	95
3.5.4.1	Die <b>for</b> -Anweisung . . . . .	95
3.5.4.2	Vereinfachte <b>for</b> -Schleifen-Notation . . . . .	96
3.5.4.3	Die <b>while</b> -Anweisung . . . . .	97
3.5.4.4	Die <b>do</b> -Anweisung . . . . .	97
3.5.4.5	Endlosschleifen . . . . .	98
3.5.5	Sprungbefehle und markierte Anweisungen . . . . .	99
3.5.6	Zusammenfassung . . . . .	101
3.5.7	Übungsaufgaben . . . . .	101
<b>4</b>	<b>Referenzdatentypen . . . . .</b>	<b>111</b>
4.1	Felder . . . . .	113
4.1.1	Was sind Felder? . . . . .	115
4.1.2	Deklaration, Erzeugung und Initialisierung von Feldern . . . . .	117
4.1.3	Felder unbekannter Länge . . . . .	119
4.1.4	Referenzen . . . . .	121
4.1.5	Ein besserer Terminkalender . . . . .	126
4.1.6	Mehrdimensionale Felder . . . . .	128
4.1.7	Mehrdimensionale Felder unterschiedlicher Länge . . . . .	132
4.1.8	Vorsicht, Falle: Kopieren von mehrdimensionalen Feldern . . . . .	134
4.1.9	Vereinfachte <b>for</b> -Schleifen-Notation . . . . .	135
4.1.10	Zusammenfassung . . . . .	136
4.1.11	Übungsaufgaben . . . . .	137
4.2	Klassen . . . . .	140
4.2.1	Willkommen in der ersten Klasse! . . . . .	141
4.2.2	Komponentenzugriff bei Objekten . . . . .	145
4.2.3	Ein erstes Adressbuch . . . . .	145
4.2.4	Klassen als Referenzdatentyp . . . . .	147
4.2.5	Felder von Klassen . . . . .	150
4.2.6	Vorsicht, Falle: Kopieren von geschachtelten Referenzdatentypen . . . . .	153
4.2.7	Zusammenfassung . . . . .	154
4.2.8	Übungsaufgaben . . . . .	154
<b>5</b>	<b>Methoden, Unterprogramme . . . . .</b>	<b>157</b>
5.1	Methoden . . . . .	158
5.1.1	Was sind Methoden? . . . . .	158
5.1.2	Deklaration von Methoden . . . . .	159
5.1.3	Parameterübergabe und Ergebnisrückgabe . . . . .	160
5.1.4	Aufruf von Methoden . . . . .	161
5.1.5	Überladen von Methoden . . . . .	163
5.1.6	Variable Argumentanzahl bei Methoden . . . . .	164
5.1.7	Vorsicht, Falle: Referenzen als Parameter . . . . .	165
5.1.8	Sichtbarkeit und Verdecken von Variablen . . . . .	168
5.1.9	Zusammenfassung . . . . .	169

5.1.10	Übungsaufgaben . . . . .	170
5.2	Rekursiv definierte Methoden . . . . .	171
5.2.1	Motivation . . . . .	171
5.2.2	Gute und schlechte Beispiele für rekursive Methoden . . . . .	173
5.2.3	Zusammenfassung . . . . .	176
5.3	Die Methode <code>main</code> . . . . .	176
5.3.1	Kommandozeilenparameter . . . . .	176
5.3.2	Anwendung der vereinfachten <code>for</code> -Schleifen-Notation . . . . .	178
5.3.3	Zusammenfassung . . . . .	178
5.3.4	Übungsaufgaben . . . . .	179
5.4	Methoden aus anderen Klassen aufrufen . . . . .	180
5.4.1	Klassenmethoden . . . . .	181
5.4.2	Die Methoden der Klasse <code>java.lang.Math</code> . . . . .	182
5.4.3	Statischer Import . . . . .	183
5.5	Methoden von Objekten aufrufen . . . . .	184
5.5.1	Instanzmethoden . . . . .	184
5.5.2	Die Methoden der Klasse <code>java.lang.String</code> . . . . .	185
5.6	Übungsaufgaben . . . . .	187
 <b>II Objektorientiertes Programmieren in Java . . . . .</b>		<b>193</b>
<b>6</b>	<b>Die objektorientierte Philosophie . . . . .</b>	<b>195</b>
6.1	Die Welt, in der wir leben . . . . .	195
6.2	Programmierparadigmen – Objektorientierung im Vergleich . . . . .	196
6.3	Die vier Grundpfeiler objektorientierter Programmierung . . . . .	198
6.3.1	Generalisierung . . . . .	198
6.3.2	Vererbung . . . . .	200
6.3.3	Kapselung . . . . .	203
6.3.4	Polymorphie . . . . .	204
6.3.5	Weitere wichtige Grundbegriffe . . . . .	205
6.4	Modellbildung – von der realen Welt in den Computer . . . . .	206
6.4.1	Grafisches Modellieren mit UML . . . . .	206
6.4.2	Entwurfsmuster . . . . .	207
6.5	Zusammenfassung . . . . .	208
6.6	Übungsaufgaben . . . . .	209
<b>7</b>	<b>Der grundlegende Umgang mit Klassen . . . . .</b>	<b>211</b>
7.1	Vom Referenzdatentyp zur Objektorientierung . . . . .	211
7.2	Instanzmethoden . . . . .	213
7.2.1	Zugriffsrechte . . . . .	213
7.2.2	Was sind Instanzmethoden? . . . . .	214
7.2.3	Instanzmethoden zur Validierung von Eingaben . . . . .	217
7.2.4	Instanzmethoden als erweiterte Funktionalität . . . . .	218
7.3	Statische Komponenten einer Klasse . . . . .	219

7.3.1	Klassenvariablen und -methoden . . . . .	220
7.3.2	Klassenkonstanten . . . . .	222
7.4	Instanziierung und Initialisierung . . . . .	225
7.4.1	Konstruktoren . . . . .	225
7.4.2	Überladen von Konstruktoren . . . . .	227
7.4.3	Der statische Initialisierer . . . . .	229
7.4.4	Der Mechanismus der Objekterzeugung . . . . .	231
7.5	Zusammenfassung . . . . .	236
7.6	Übungsaufgaben . . . . .	237
<b>8</b>	<b>Vererbung und Polymorphie . . . . .</b>	<b>257</b>
8.1	Wozu braucht man Vererbung? . . . . .	257
8.1.1	Aufgabenstellung . . . . .	257
8.1.2	Analyse des Problems . . . . .	258
8.1.3	Ein erster Ansatz . . . . .	258
8.1.4	Eine Klasse für sich . . . . .	259
8.1.5	Stärken der Vererbung . . . . .	260
8.1.6	Vererbung verhindern durch <b>final</b> . . . . .	263
8.1.7	Übungsaufgaben . . . . .	264
8.2	Die <b>super</b> -Referenz . . . . .	265
8.3	Überschreiben von Methoden und Variablen . . . . .	267
8.3.1	Dynamisches Binden . . . . .	267
8.3.2	Überschreiben von Methoden verhindern durch <b>final</b> . . . . .	269
8.4	Die Klasse <code>java.lang.Object</code> . . . . .	269
8.5	Übungsaufgaben . . . . .	273
8.6	Abstrakte Klassen und Interfaces . . . . .	273
8.6.1	Einsatzszenarien am Beispiel . . . . .	273
8.6.2	Abstrakte Klassen im Detail . . . . .	277
8.6.3	Interfaces im Detail . . . . .	280
8.7	Interfaces mit Default-Methoden und statischen Methoden . . . . .	283
8.7.1	Deklaration von Default-Methoden . . . . .	283
8.7.2	Deklaration von statischen Methoden . . . . .	284
8.7.3	Auflösung von Namensgleichheiten bei Default-Methoden . . . . .	285
8.7.4	Interfaces und abstrakte Klassen im Vergleich . . . . .	287
8.8	Weiteres zum Thema Objektorientierung . . . . .	287
8.8.1	Erstellen von Paketen . . . . .	287
8.8.2	Zugriffsrechte . . . . .	289
8.8.3	Innere Klassen . . . . .	290
8.8.4	Anonyme Klassen . . . . .	296
8.9	Zusammenfassung . . . . .	299
8.10	Übungsaufgaben . . . . .	299
<b>9</b>	<b>Exceptions und Errors . . . . .</b>	<b>311</b>
9.1	Eine Einführung in Exceptions . . . . .	312
9.1.1	Was ist eine Exception? . . . . .	312

9.1.2	Übungsaufgaben . . . . .	314
9.1.3	Abfangen von Exceptions . . . . .	314
9.1.4	Ein Anwendungsbeispiel . . . . .	315
9.1.5	Die <code>RuntimeException</code> . . . . .	318
9.1.6	Übungsaufgaben . . . . .	319
9.2	Exceptions für Fortgeschrittene . . . . .	321
9.2.1	Definieren eigener Exceptions . . . . .	321
9.2.2	Übungsaufgaben . . . . .	323
9.2.3	Vererbung und Exceptions . . . . .	323
9.2.4	Vorsicht, Falle! . . . . .	327
9.2.5	Der <b>finally</b> -Block . . . . .	329
9.2.6	Die Klassen <code>Throwable</code> und <code>Error</code> . . . . .	333
9.2.7	Zusammenfassung . . . . .	335
9.2.8	Übungsaufgaben . . . . .	335
9.3	Assertions . . . . .	336
9.3.1	Zusicherungen im Programmcode . . . . .	336
9.3.2	Ausführen des Programmcodes . . . . .	337
9.3.3	Zusammenfassung . . . . .	338
9.4	Mehrere Ausnahmetypen in einem <b>catch</b> -Block . . . . .	338
9.5	Ausblick: <b>try</b> -Block mit Ressourcen . . . . .	340
<b>10</b>	<b>Fortgeschrittene objektorientierte Programmierung . . . . .</b>	<b>341</b>
10.1	Aufzählungstypen . . . . .	341
10.1.1	Deklaration eines Aufzählungstyps . . . . .	342
10.1.2	Instanzmethoden der <b>enum</b> -Objekte . . . . .	342
10.1.3	Selbstdefinierte Instanzmethoden für <b>enum</b> -Objekte . . . . .	343
10.1.4	Übungsaufgaben . . . . .	344
10.2	Generische Datentypen . . . . .	346
10.2.1	Herkömmliche Generizität . . . . .	347
10.2.2	Generizität durch Typ-Parameter . . . . .	349
10.2.3	Einschränkungen der Typ-Parameter . . . . .	351
10.2.4	Wildcards . . . . .	353
10.2.5	Bounded Wildcards . . . . .	354
10.2.6	Generische Methoden . . . . .	356
10.2.7	Verkürzte Notation bei generischen Datentypen . . . . .	358
10.2.8	Ausblick . . . . .	361
10.2.9	Übungsaufgaben . . . . .	361
10.3	Sortieren von Feldern und das Interface <code>Comparable</code> . . . . .	366
<b>11</b>	<b>Einige wichtige Hilfsklassen . . . . .</b>	<b>369</b>
11.1	Die Klasse <code>StringBuffer</code> . . . . .	369
11.1.1	Arbeiten mit <code>String</code> -Objekten . . . . .	369
11.1.2	Arbeiten mit <code>StringBuffer</code> -Objekten . . . . .	372
11.1.3	Übungsaufgaben . . . . .	374
11.2	Die Wrapper-Klassen (Hüll-Klassen) . . . . .	375

11.2.1	Arbeiten mit „eingepackten“ Daten . . . . .	375
11.2.2	Aufbau der Wrapper-Klassen . . . . .	377
11.2.3	Ein Anwendungsbeispiel . . . . .	379
11.2.4	Automatische Typwandlung für die Wrapper-Klassen . . .	380
11.2.5	Übungsaufgaben . . . . .	382
11.3	Die Klassen <code>BigInteger</code> und <code>BigDecimal</code> . . . . .	382
11.3.1	Arbeiten mit langen Ganzzahlen . . . . .	383
11.3.2	Aufbau der Klasse <code>BigInteger</code> . . . . .	384
11.3.3	Übungsaufgaben . . . . .	386
11.3.4	Arbeiten mit langen Gleitkommazahlen . . . . .	387
11.3.5	Aufbau der Klasse <code>BigDecimal</code> . . . . .	390
11.3.6	Viele Stellen von Nullstellen gefällig? . . . . .	392
11.3.7	Übungsaufgaben . . . . .	394
11.4	Die Klasse <code>DecimalFormat</code> . . . . .	394
11.4.1	Standardausgaben in Java . . . . .	394
11.4.2	Arbeiten mit Format-Objekten . . . . .	395
11.4.3	Vereinfachte formatierte Ausgabe . . . . .	398
11.4.4	Übungsaufgaben . . . . .	398
11.5	Die Klassen <code>Date</code> und <code>Calendar</code> . . . . .	399
11.5.1	Arbeiten mit „Zeitpunkten“ . . . . .	399
11.5.2	Auf die Plätze, fertig, los! . . . . .	400
11.5.3	Spezielle <code>Calendar</code> -Klassen . . . . .	401
11.5.4	Noch einmal: Zeitmessung . . . . .	404
11.5.5	Übungsaufgaben . . . . .	405
11.6	Die Klassen <code>SimpleDateFormat</code> und <code>DateFormat</code> . . . . .	405
11.6.1	Arbeiten mit Format-Objekten für Datum/Zeit-Angaben . .	405
11.6.2	Übungsaufgaben . . . . .	410
11.7	Die <code>Collection</code> -Klassen . . . . .	410
11.7.1	„Sammlungen“ von Objekten – der Aufbau des Interface <code>Collection</code> . . . . .	411
11.7.2	„Sammlungen“ durchgehen – der Aufbau des Interface <code>Iterator</code> . . . . .	413
11.7.3	Mengen . . . . .	414
11.7.3.1	Das Interface <code>Set</code> . . . . .	414
11.7.3.2	Die Klasse <code>HashSet</code> . . . . .	415
11.7.3.3	Das Interface <code>SortedSet</code> . . . . .	416
11.7.3.4	Die Klasse <code>TreeSet</code> . . . . .	417
11.7.4	Listen . . . . .	419
11.7.4.1	Das Interface <code>List</code> . . . . .	419
11.7.4.2	Die Klassen <code>ArrayList</code> und <code>LinkedList</code> . . . . .	420
11.7.4.3	Suchen und Sortieren – die Klassen <code>Collections</code> und <code>Arrays</code> . . . . .	422
11.7.5	Verkürzte Notation bei <code>Collection</code> -Datentypen . . . . .	424
11.7.6	Übungsaufgaben . . . . .	426

11.8	Die Klasse <code>StringTokenizer</code> . . . . .	426
11.8.1	Übungsaufgaben . . . . .	429
<b>III</b>	<b>Grafische Oberflächen in Java . . . . .</b>	<b>431</b>
<b>12</b>	<b>Aufbau grafischer Oberflächen in Frames – von AWT nach Swing . . . . .</b>	<b>433</b>
12.1	Grundsätzliches zum Aufbau grafischer Oberflächen . . . . .	433
12.2	Ein einfaches Beispiel mit dem AWT . . . . .	434
12.3	Let's swing now! . . . . .	437
12.4	Etwas „Fill-in“ gefällig? . . . . .	439
12.5	Die AWT- und Swing-Klassenbibliothek im Überblick . . . . .	441
12.6	Übungsaufgaben . . . . .	443
<b>13</b>	<b>Swing-Komponenten . . . . .</b>	<b>445</b>
13.1	Die abstrakte Klasse <code>Component</code> . . . . .	445
13.2	Die Klasse <code>Container</code> . . . . .	446
13.3	Die abstrakte Klasse <code>JComponent</code> . . . . .	447
13.4	Layout-Manager, Farben und Schriften . . . . .	448
13.4.1	Die Klasse <code>Color</code> . . . . .	449
13.4.2	Die Klasse <code>Font</code> . . . . .	451
13.4.3	Layout-Manager . . . . .	452
13.4.3.1	Die Klasse <code>FlowLayout</code> . . . . .	453
13.4.3.2	Die Klasse <code>BorderLayout</code> . . . . .	455
13.4.3.3	Die Klasse <code>GridLayout</code> . . . . .	456
13.5	Einige Grundkomponenten . . . . .	458
13.5.1	Die Klasse <code>JLabel</code> . . . . .	459
13.5.2	Die abstrakte Klasse <code>AbstractButton</code> . . . . .	461
13.5.3	Die Klasse <code>JButton</code> . . . . .	461
13.5.4	Die Klasse <code>JToggleButton</code> . . . . .	463
13.5.5	Die Klasse <code>JCheckBox</code> . . . . .	464
13.5.6	Die Klassen <code>JRadioButton</code> und <code>ButtonGroup</code> . . . . .	465
13.5.7	Die Klasse <code>JComboBox</code> . . . . .	467
13.5.8	Die Klasse <code>JList</code> . . . . .	470
13.5.9	Die abstrakte Klasse <code>JTextComponent</code> . . . . .	473
13.5.10	Die Klassen <code>JTextField</code> und <code>JPasswordField</code> . . . . .	473
13.5.11	Die Klasse <code>JTextArea</code> . . . . .	476
13.5.12	Die Klasse <code>JScrollPane</code> . . . . .	478
13.5.13	Die Klasse <code>JPanel</code> . . . . .	480
13.6	Spezielle Container, Menüs und Toolbars . . . . .	481
13.6.1	Die Klasse <code>JFrame</code> . . . . .	482
13.6.2	Die Klasse <code>JWindow</code> . . . . .	483
13.6.3	Die Klasse <code>JDialog</code> . . . . .	483
13.6.4	Die Klasse <code>JMenuBar</code> . . . . .	486
13.6.5	Die Klasse <code>JToolBar</code> . . . . .	489

13.7	Übungsaufgaben	492
<b>14</b>	<b>Ereignisverarbeitung</b>	<b>495</b>
14.1	Zwei einfache Beispiele	496
14.1.1	Zufällige Grautöne als Hintergrund	496
14.1.2	Ein interaktiver Bilderrahmen	499
14.2	Programmiervarianten für die Ereignisverarbeitung	503
14.2.1	Innere Klasse als Listener-Klasse	503
14.2.2	Anonyme Klasse als Listener-Klasse	503
14.2.3	Container-Klasse als Listener-Klasse	504
14.2.4	Separate Klasse als Listener-Klasse	505
14.3	Event-Klassen und -Quellen	507
14.4	Listener-Interfaces und Adapter-Klassen	510
14.5	Listener-Registrierung bei den Event-Quellen	516
14.6	Auf die Plätze, fertig, los!	519
14.7	Übungsaufgaben	524
<b>15</b>	<b>Einige Ergänzungen zu Swing-Komponenten</b>	<b>529</b>
15.1	Zeichnen in Swing-Komponenten	529
15.1.1	Grafische Darstellung von Komponenten	529
15.1.2	Das Grafikkoordinatensystem	530
15.1.3	Die abstrakte Klasse <code>Graphics</code>	531
15.1.4	Ein einfaches Zeichenprogramm	534
15.1.5	Layoutveränderungen und der Einsatz von <code>revalidate</code>	536
15.2	Noch mehr Swing gefällig?	538
15.3	Übungsaufgaben	540
<b>IV</b>	<b>Threads, Datenströme und Netzwerkanwendungen</b>	<b>543</b>
<b>16</b>	<b>Parallele Programmierung mit Threads</b>	<b>545</b>
16.1	Ein einfaches Beispiel	545
16.2	Threads in Java	547
16.2.1	Die Klasse <code>Thread</code>	548
16.2.2	Das Interface <code>Runnable</code>	552
16.2.3	Threads vorzeitig beenden	554
16.3	Wissenswertes über Threads	556
16.3.1	Lebenszyklus eines Threads	556
16.3.2	Thread-Scheduling	558
16.3.3	Dämon-Threads und Thread-Gruppen	558
16.4	Thread-Synchronisation und -Kommunikation	559
16.4.1	Das Leser/Schreiber-Problem	560
16.4.2	Das Erzeuger/Verbraucher-Problem	563
16.5	Threads in Swing-Anwendungen	571
16.5.1	Auf die Plätze, fertig, los!	571

16.5.2	Spielereien . . . . .	574
16.5.3	Swing-Komponenten sind nicht Thread-sicher . . . . .	577
16.6	Übungsaufgaben . . . . .	578
<b>17</b>	<b>Ein- und Ausgabe über I/O-Streams . . . . .</b>	<b>581</b>
17.1	Grundsätzliches zu I/O-Streams in Java . . . . .	582
17.2	Dateien und Verzeichnisse – die Klasse <code>File</code> . . . . .	582
17.3	Ein- und Ausgabe über Character-Streams . . . . .	585
17.3.1	Einfache <code>Reader</code> - und <code>Writer</code> -Klassen . . . . .	586
17.3.2	Gepufferte <code>Reader</code> - und <code>Writer</code> -Klassen . . . . .	589
17.3.3	Die Klasse <code>StreamTokenizer</code> . . . . .	591
17.3.4	Die Klasse <code>PrintWriter</code> . . . . .	592
17.3.5	Die Klassen <code>IOTools</code> und <code>Scanner</code> . . . . .	594
17.3.5.1	Was machen eigentlich die <code>IOTools</code> ? . . . . .	594
17.3.5.2	Konsoleneingabe über ein <code>Scanner</code> -Objekt . . . . .	595
17.4	Ein- und Ausgabe über Byte-Streams . . . . .	596
17.4.1	Einige <code>InputStream</code> - und <code>OutputStream</code> -Klassen . . . . .	597
17.4.2	Die Serialisierung und Deserialisierung von Objekten . . . . .	599
17.4.3	Die Klasse <code>PrintStream</code> . . . . .	601
17.5	Streams im <code>try</code> -Block mit Ressourcen . . . . .	602
17.6	Einige abschließende Bemerkungen . . . . .	604
17.6.1	Das Paket <code>java.nio</code> . . . . .	605
17.6.2	Das Paket <code>java.nio.file</code> . . . . .	606
17.6.2.1	Das Interface <code>Path</code> und die Klasse <code>Paths</code> . . . . .	606
17.6.2.2	Die Klasse <code>Files</code> . . . . .	607
17.7	Übungsaufgaben . . . . .	610
<b>18</b>	<b>Client/Server-Programmierung in Netzwerken . . . . .</b>	<b>613</b>
18.1	Wissenswertes über Netzwerkkommunikation . . . . .	614
18.1.1	Protokolle . . . . .	614
18.1.2	IP-Adressen . . . . .	616
18.1.3	Ports und Sockets . . . . .	617
18.2	Client/Server-Programmierung . . . . .	618
18.2.1	Die Klassen <code>ServerSocket</code> und <code>Socket</code> . . . . .	619
18.2.2	Ein einfacher Server . . . . .	621
18.2.3	Ein einfacher Client . . . . .	624
18.2.4	Ein Server für mehrere Clients . . . . .	625
18.2.5	Ein Mehrzweck-Client . . . . .	628
18.2.6	Client/Server-Kommunikation über URLs . . . . .	631
18.3	Übungsaufgaben . . . . .	632
<b>19</b>	<b>Lambda-Ausdrücke, Streams und Pipeline-Operationen . . . . .</b>	<b>637</b>
19.1	Lambda-Ausdrücke . . . . .	637
19.1.1	Lambda-Ausdrücke in Aktion – zwei Beispiele . . . . .	638
19.1.2	Lambda-Ausdrücke im Detail . . . . .	641

19.1.3	Lambda-Ausdrücke und funktionale Interfaces . . . . .	643
19.1.4	Vordefinierte funktionale Interfaces und Anwendungen auf Datenstrukturen . . . . .	645
19.1.5	Methodenreferenzen als Lambda-Ausdrücke . . . . .	649
19.1.6	Zugriff auf Variablen aus der Umgebung innerhalb eines Lambda-Ausdrucks . . . . .	652
19.1.7	Übungsaufgaben . . . . .	653
19.2	Streams und Pipeline-Operationen . . . . .	654
19.2.1	Streams in Aktion . . . . .	655
19.2.2	Streams und Pipelines im Detail . . . . .	657
19.2.3	Erzeugen von endlichen und unendlichen Streams . . . . .	658
19.2.4	Die Stream-API . . . . .	661
19.2.5	Übungsaufgaben . . . . .	664
<b>V</b>	<b>Abschluss, Ausblick und Anhang . . . . .</b>	<b>667</b>
<b>20</b>	<b>Blick über den Tellerrand . . . . .</b>	<b>669</b>
20.1	JShell für kleine Skripte . . . . .	670
20.2	Das Java-Modulsystem . . . . .	674
20.3	Bühne frei für JavaFX . . . . .	681
20.4	Beginn einer neuen Zeitrechnung . . . . .	690
20.5	Webprogrammierung und verteilte Systeme . . . . .	692
20.6	Zu guter Letzt . . . . .	695
<b>A</b>	<b>Der Weg zum guten Programmierer . . . . .</b>	<b>697</b>
A.1	Die goldenen Regeln der Code-Formatierung . . . . .	698
A.2	Die goldenen Regeln der Namensgebung . . . . .	701
A.3	Zusammenfassung . . . . .	703
<b>B</b>	<b>Ohne Werkzeug geht es nicht . . . . .</b>	<b>705</b>
B.1	Die API-Dokumentation zum Nachschlagen . . . . .	706
B.2	Die IDE, dein Freund und Helfer . . . . .	708
B.3	Alle Versionen stets im Griff . . . . .	710
B.4	Testen bitte nicht vergessen . . . . .	712
B.5	Der Automat kann es besser . . . . .	714
<b>C</b>	<b>Die Klasse <code>IOTools</code> – Tastatureingaben in Java . . . . .</b>	<b>717</b>
C.1	Kurzbeschreibung . . . . .	717
C.2	Anwendung der <code>IOTools</code> -Methoden . . . . .	718
<b>Glossar</b>	<b>. . . . .</b>	<b>721</b>
<b>Stichwortverzeichnis</b>	<b>. . . . .</b>	<b>741</b>

# Vorwort

Unsere moderne Welt mit ihren enormen Informations- und Kommunikationsbedürfnissen wäre ohne Computer und mobile Endgeräte wie Smartphones und deren weltweite Vernetzung undenkbar. Ob wir Einkäufe abwickeln, uns Informationen beschaffen, Reisen buchen, Bankgeschäfte tätigen oder einfach nur Mitteilungen verschicken – wir benutzen diese Techniken wie selbstverständlich. Dienstleistungen, Produkte, die Arbeitswelt und das gesellschaftliche Leben basieren in zunehmendem Maße auf Software, und die Digitalisierung schreitet in allen Bereichen immer weiter voran. Ob als Nutzer, als Auftraggeber oder als Entwickler – Schul- und Hochschulabgänger werden mit Sicherheit an ihrem späteren Arbeitsplatz in irgendeiner Weise mit Software oder gar Softwareentwicklung zu tun haben. Aber auch außerhalb des Berufslebens können alle von Kenntnissen darüber, wie Programme im Allgemeinen oder z. B. Smartphone-Apps im Speziellen arbeiten, nur profitieren. Die Chance, so früh wie möglich zu lernen, wie man unsere digitale Welt mitgestalten kann, sollte jeder wahrnehmen – und Programmieren lernen ist hierfür ein erster Schritt.

Eine qualifizierte Programmiergrundausbildung ist unerlässlich, um an der Gestaltung moderner Informatikanwendungen mitwirken zu können. Leider erscheint vielen das Erlernen einer Programmiersprache zu Beginn einer weitergehenden Informatikausbildung als unüberwindbare Hürde. Die häufig angepriesene Mächtigkeit und Komplexität der mittlerweile gängigen Ausbildungssprache Java schürt bei nicht wenigen Programmieranfängern den Zweifel, jemals in die „Geheimnisse“ des Programmierens eingeweiht zu werden.

Aufgrund unserer langjährigen Erfahrungen aus Lehrveranstaltungen für Studierende unterschiedlicher Fachrichtungen, in denen in der Regel rund zwei Drittel der Teilnehmer bis zum Kursbeginn noch nicht selbst programmierten, entschlossen wir uns, das vorliegende Buch zu verfassen. Hauptanforderung dabei war die „Verständlichkeit auch für Anfänger“, um Schülerinnen und Schülern, Studentinnen und Studenten, aber auch Hausfrauen und Hausmännern einen leicht verständlichen Grundkurs „Programmieren in Java“ zu vermitteln. Auf theoretischen Ballast oder ein breites Informatikfundament wollten wir bewusst verzichten. Wir hofften, unser Konzept, auch absolute Neulinge behutsam in die Materie einzuführen, überzeugt unsere Leserinnen und Leser. Diese Hoffnung wurde mehr als erfüllt – zahlreiche überaus positive Leserkommentare unterstreichen

dies. So liegt nun bereits die achte, überarbeitete Auflage vor, in der wir viele konstruktive Umgestaltungsvorschläge von Leserinnen und Lesern berücksichtigt und außerdem jüngste Neuerungen der Sprache Java aufgenommen haben. Hält man Ausschau nach dem erfolgreichsten aller Bücher, stößt man wohl auf die Bibel. Das Buch der Bücher steht für hohe Auflagen und eine große Leserschaft. In unzählige Sprachen übersetzt, stellt die Bibel den Traum eines jeden Autors dar. Was Sie hier in den Händen halten, hat mit der Bibel natürlich ungefähr so viel zu tun wie eine Weinbergschnecke mit der Formel 1. Zwar ist auch dieses Buch in mehrere Teile untergliedert und stammt aus mehr als einer Feder – mit göttlichen Offenbarungen und Prophezeiungen können wir dennoch nicht aufwarten. Sie finden in diesem Buch auch weder Hebräisch noch Latein. Im schlimmsten Falle treffen Sie auf etwas, das Ihnen trotz unserer guten Vorsätze (zumindest zu Beginn Ihrer Lektüre) wie Fachchinesisch oder böhmische Dörfer vorkommen könnte. Lassen Sie sich davon aber nicht abschrecken – im Glossar im Anhang können Sie „Übersetzungen“ für den Fachjargon jederzeit nachschlagen.

Etlichen Personen, die zur Entstehung dieses Buches beitrugen, wollen wir an dieser Stelle herzlichst danken: An erster Stelle zu erwähnen ist unser langjähriger ehemaliger Co-Autor Jens Scheffler, der zusammen mit seinen damaligen Tutorenkollegen Thomas Much, Michael Ohr und Oliver Wagner viel Schweiß und Mühe in die Erstellung eines ersten Vorlesungsskripts steckte. Eine wichtige Rolle für die „Reifung“ bis zur vorliegenden Buchfassung spielten unsere „Korrektoren“ und „Testleser“. Hagen Buchwald, Michael Decker, Mario Dehner, Tobias Dietrich, Marc Goutier, Rudi Klatter, Niklas Kühl, Roland Küstermann, Jonas Lehner, Joachim Melcher, Cornelia Richter-von Hagen, Sebastian Ratz, Frank Schlottmann, Oliver Schöll, Lukas Struppek, Janna Ulrich und Leonard von Hagen brachten mit großem Engagement wertvolle Kommentare und Verbesserungsvorschläge ein oder unterstützten uns beim Auf- und Ausbau der Buch-Website, bei der Überarbeitung von Grafiken oder mit der Erstellung von Aufgaben und der Bereitstellung von Tools. Schließlich sind da noch mehrere Studierenden-Jahrgänge der Studiengänge Wirtschaftsingenieurwesen, Wirtschaftsmathematik, Technische Volkswirtschaftslehre und Wirtschaftsinformatik, die sich im Rahmen unserer Lehrveranstaltungen „Programmieren I“, „Programmierung kommerzieller Systeme“, „Fortgeschrittene Programmieretechniken“, „Web-Programmierung“ und „Verteilte Systeme“ mit den zugehörigen Webseiten, Foliensätzen und Übungsblättern „herumgeschlagen“ und uns auf Fehler und Unklarheiten hingewiesen haben. Das insgesamt sehr positive Feedback, auch aus anderen Studiengängen, war und ist Ansporn für uns, diesen Grundkurs Programmieren weiterzuentwickeln. Schließlich geht auch ein Dankeschön an die Leserinnen und Leser, die uns per E-Mail Hinweise und Tipps für die inhaltliche Verbesserung von Buch und Website zukommen ließen.

Zu guter Letzt geht unser Dank an Frau Brigitte Bauer-Schiewek und Frau Irene Weilhart vom Carl Hanser Verlag für die gewohnt gute Zusammenarbeit.

# Einleitung

Kennen Sie das auch? Sie gehen in eine Bar und sehen eine wunderschöne Frau bzw. einen attraktiven Mann – vielleicht *die* Partnerin oder *den* Partner fürs Leben! Sie kontrollieren unauffällig den Sitz Ihrer Kleidung, schlendern elegant zum Tresen und schenken ihr/ihm ein zuckersüßes Lächeln. Ihre Blicke sagen mehr als tausend Worte, jeder Zentimeter Ihres Körpers signalisiert: „Ich will Dich!“ In dem Moment jedoch, als Sie ihr/ihm unauffällig Handy-Nummer und E-Mail zustecken wollen, betritt ein Schrank von einem Kerl bzw. die Reinkarnation von Marilyn Monroe die Szene. Frau sieht Mann, Mann sieht Frau, und Sie sehen einen leeren Stuhl und eine Rechnung über drei Milchshakes und eine Cola.

Wie kann Ihnen dieses Buch helfen, so etwas zu vermeiden? Die traurige Antwort lautet: Gar nicht! Sie können mit diesem Buch weder Ihren Schwarm beeindrucken noch Konkurrenten oder Konkurrentinnen niederschlagen (denn dafür ist es einfach zu leicht). Wenn Sie also einen schnellen Weg zum sicheren Erfolg suchen, sind Sie wohl mit anderen Werken besser beraten.

Aber im Ernst: Wozu ist das Buch also wirklich zu gebrauchen? Die folgenden Seiten verraten es Ihnen.

## Java – mehr als nur kalter Kaffee?

Seit dem Einzug von Internet und World Wide Web (WWW) ins öffentliche Leben surfen, mailen und chatten Milliarden von Menschen täglich in der virtuellen Welt. Es gehört beinahe schon zum guten Ton, im Netz der Netze vertreten zu sein oder dessen Inhalte in irgendeiner Form mitzugestalten.

Der rasanten Entwicklung von Web-Technologien hatte es die Firma Sun, die 2010 von Oracle übernommen wurde, zu verdanken, dass sich ihre 1995 vorgestellte Programmiersprache Java schnell zu einer der populärsten entwickelte. Am eigentlichen Sprachkonzept war nur wenig Neues, denn die geistigen Väter hatten sich stark an der Sprache C++ orientiert. Im Gegensatz zu C++ konnten mit Java jedoch Programme erstellt werden, die sich direkt in Webseiten einbinden und ausführen lassen. Java war somit die erste Sprache für das WWW.

Natürlich ist für Java die Entwicklung nicht stehen geblieben. Die einstige „Netzsprache“ hat sich in ihrer aktuellen Version (siehe z. B. [43] und [36]), mit der wir

in diesem Buch arbeiten, zu einer vollwertigen Konkurrenz zu den anderen gängigen Sprachen gemausert. Datenbank- oder Netzwerkzugriffe, anspruchsvolle Grafikanwendungen, Spieleprogrammierung – alles ist möglich. Gerade in dem heute so aktuellen Bereich „Verteilte Anwendungsentwicklung“ bietet Java ein breites Spektrum an Möglichkeiten. Mit wenigen Programmzeilen gelingt es, Anwendungen zu schreiben, die das Internet bzw. das World Wide Web (WWW) nutzen. Grundlage dafür bildet die umfangreiche Java-Klassenbibliothek, die Sammlung einer Vielzahl vorgefertigter Klassen und Interfaces, die einem das Programmiererleben wesentlich vereinfachen. Nicht minder interessante Teile dieser Klassenbibliothek statten Java-Programme mit enormen, weitgehend plattformunabhängigen grafischen Fähigkeiten aus. So können auch Programme mit grafischen Oberflächen portabel bleiben.

Dies erklärt sicherlich auch das große Interesse, das der Sprache Java in den letzten Jahren entgegengebracht wurde. Bedenkt man die Anzahl von Buchveröffentlichungen, Zeitschriftenbeiträgen, Webseiten, Newsgroups, Foren und Blogs zum Thema, so wird der erfolgreiche Weg, den die Sprache Java hinter sich hat, offensichtlich. Auch im kommerziellen Bereich ist Java nicht mehr wegzudenken, denn die Produktpalette der meisten großen Softwarehäuser weist mittlerweile eine Java-Schiene auf. Und wer heute auch nur mit einem Smartphone telefoniert, kommt häufig (bewusst oder unbewusst) mit Java in Berührung. Für Sie als Leserin oder Leser dieses Buchs bedeutet das jedenfalls, dass es sicherlich kein Fehler ist, Erfahrung in der Programmierung mit Java zu haben.<sup>1</sup>

## Java für Anfänger – das Konzept dieses Buches

Viele Autoren von Java-Büchern gehen in ihren Kursen den Weg, gleich von Anfang an in die Objektorientierung einstieg. Wir haben uns jedoch entschieden, diese Thematik erst mal zurückzustellen und den Lernprozess anders auszugestalten. Vergleichen Sie das mit dem Erlernen einer gesprochenen Sprache wie zum Beispiel Spanisch. Auch hier empfiehlt es sich unserer Meinung nach, sich erst einen grundlegenden Wortschatz anzueignen und ein Verständnis dafür zu entwickeln, wie sich die Sprache „anfühlt“, bevor man sich mit Regeln über guten Schreibstil oder Besonderheiten der spanischen Lyrik auseinandersetzen kann.

Wie schreibt man nun ein Buch über das Programmieren für den absoluten Neuesteiger, wenn man selbst seit vielen Jahren programmiert? Vor diesem Problem standen die Autoren einst. Das Buch sollte den Leserinnen und Lesern die Konzepte von Java korrekt vermitteln, ohne sie zu überfordern. Maßstab für die Qualität dieses Buches war ferner die Anforderung, dass es sich optimal als Begleitmaterial für einführende und weiterführende Vorlesungen in Bachelor-Studiengängen einsetzen ließ.

---

<sup>1</sup> Als potenzieller Berufseinsteiger/-in oder -umsteiger/-in wissen Sie vielleicht ein Lied davon zu singen, wenn Sie sich Stellenanzeigen im Bereich Softwareentwicklung ansehen – Java scheint allgegenwärtig zu sein.

Weil die Autoren auf viele Jahre studentische Programmierausbildung in einführenden und weiterführenden Kursen des Instituts für Angewandte Informatik und Formale Beschreibungsverfahren (Institut AIFB) am Karlsruher Institut für Technologie (KIT – Universität des Landes Baden-Württemberg und nationales Großforschungszentrum in der Helmholtz-Gemeinschaft) sowie an der Dualen Hochschule Baden-Württemberg (DHBW) zurückblicken können, gab und gibt es natürlich gewisse Erfahrungswerte darüber, welche Themen gerade den Neulingen besondere Probleme bereiteten. Daher rührt auch der Entschluss, das Thema „Objektorientierung“ zunächst in den Hintergrund zu stellen. Fast jedes Java-Buch beginnt mit diesem Thema und vergisst, dass man zuerst programmieren und „algorithmisch denken“ können muss, bevor man die Vorteile der objektorientierten Programmierung erkennen, nutzen und schätzen lernen kann. Seien Sie deshalb nicht verwirrt, wenn Sie dieses sonst so beliebte Schlagwort vor Seite 193 wenig zu Gesicht bekommen.

Unser Buch setzt keinerlei Vorkenntnisse aus den Bereichen Programmieren, Programmiersprachen und Informatik voraus. Sie können es also verwenden, nicht nur, um Java, sondern auch das Programmieren zu erlernen. Alle Kapitel sind mit Übungsaufgaben ausgestattet, die Sie zum besseren Verständnis bearbeiten sollten. *Man lernt eine Sprache nur, wenn man sie auch spricht!*

In den Teilen III und IV führen wir Sie auch in die Programmierung fortgeschrittener Anwendungen auf Basis der umfangreichen Java-Klassenbibliothek ein. Wir können und wollen dabei aber nicht auf jedes Detail eingehen, sodass wir alle Leserinnen und Leser bereits an dieser Stelle dazu animieren möchten, regelmäßig einen Blick in die sogenannte API-Spezifikation<sup>2</sup> der Klassenbibliothek [43] zu werfen – nicht zuletzt, weil wir im „Programmieralltag“ von einem routinierten Umgang mit API-Spezifikationen nur profitieren können. Sollten Sie Schwierigkeiten haben, sich mit dieser von Oracle zur Verfügung gestellten Dokumentation der Klassenbibliothek zurechtzufinden, hilft Ihnen bestimmt unser kleines Kapitel in Anhang B.

## Zusatzmaterial und Kontakt zu den Autoren

Alle Leserinnen und Leser sind herzlich eingeladen, die Autoren über Fehler und Unklarheiten zu informieren. Wenn eine Passage unverständlich war, sollte sie zur Zufriedenheit künftiger Leserinnen und Leser anders formuliert werden. Wenn Sie in dieser Hinsicht also Fehlermeldungen, Anregungen oder Fragen haben, können Sie über unsere Website

<http://www.grundkurs-java.de/>

<sup>2</sup> API steht für Application Programming Interface, die Programmierschnittstelle für eine Klasse, ein Paket oder eine ganze Klassenbibliothek.

Kontakt mit den Autoren aufnehmen. Dort finden Sie auch alle Beispielprogramme aus dem Buch, Lösungshinweise zu den Übungsaufgaben und ergänzende Materialien zum Download sowie Literaturhinweise, interessante Links, eine Liste eventueller Fehler im Buch und deren Korrekturen. Dozentinnen und Dozenten, die das Material dieses Buchs oder sogar Teile unserer Vorlesungsfolien für eigene Vorlesungen nutzen möchten, sollten sich mit uns in Verbindung setzen. Im Literaturverzeichnis haben wir sowohl Bücher als auch Internet-Links angegeben, die aus unserer Sicht als weiterführende Literatur geeignet sind und neben Java im Speziellen auch einige weitere Themenbereiche wie zum Beispiel Informatik, Algorithmen, Nachschlagewerke, Softwaretechnik, Objektorientierung und Modellierung einbeziehen.

## Verwendete Schreibweisen

Wir verwenden *Kursivschrift* zur Betonung bestimmter Wörter und **Fettschrift** zur Kennzeichnung von Begriffen, die im entsprechenden Abschnitt erstmals auftauchen und definiert bzw. erklärt werden. Im laufenden Text wird *Maschinschrift* für Bezeichner verwendet, die in Java vordefiniert sind oder in Programmbeispielen eingeführt und benutzt werden, während reservierte Wörter (Schlüsselwörter, Wortsymbole), die in Java eine vordefinierte, unveränderbar festgelegte Bedeutung haben, in **fetter Maschinschrift** gesetzt sind. Beide Schriften kommen auch in den vom Text abgesetzten Listings und Bildschirmausgaben von Programmen zum Einsatz. Java-Programme sind teilweise ohne und teilweise mit führenden Zeilennummern abgedruckt. Solche Zeilennummern sind dabei lediglich als Orientierungshilfe gedacht und natürlich *kein* Bestandteil des Java-Programms.

Literaturverweise auf Bücher und Web-Links werden stets in der Form [nr] mit der Nummer nr des entsprechenden Eintrags im Literaturverzeichnis angegeben.

# Kapitel 7

## Der grundlegende Umgang mit Klassen

Im letzten Kapitel haben wir erfahren, dass sich die objektorientierte Philosophie aus den vier Konzepten Generalisierung, Vererbung, Kapselung und Polymorphie zusammensetzt. Wir haben jeden dieser Begriffe – in der Theorie – erklärt und uns die Idee klarzumachen versucht, die hinter der Objektorientierung steht. Wir haben jedoch noch nicht gelernt, diese Konzepte in Java umzusetzen. In diesem und dem folgenden Kapitel soll dieser Mangel behoben werden. Anhand einfacher Beispiele werden wir lernen, wie sich Klassen auch in Java zu mehr als nur einfachen Datenspeichern mausern.

### 7.1 Vom Referenzdatentyp zur Objektorientierung

In diesem Kapitel werden wir versuchen, verschiedene Aspekte im Leben eines *Studierenden* zu modellieren. Wir beginnen hierbei mit einer einfachen Klasse, wie wir sie schon aus den vorigen Kapiteln kennen:

```
1  /** Diese Klasse simuliert einen Studenten */
2  public class Student {
3
4      /** Der Name des Studenten */
5      public String name;
6
7      /** Die Matrikelnummer des Studenten */
8      public int nummer;
9  }
10
```

Wie Sie sehen, haben wir die Klasse allerdings nicht *Studierender* genannt, was dem aktuellen geschlechtsneutralen Sprachgebrauch an den Hochschulen eher entsprechen würde. Der Einfachheit (und Kürze) halber haben wir uns dazu ent-

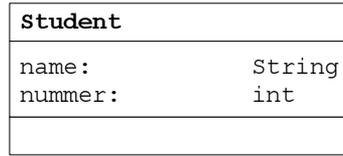


Abbildung 7.1: Die Klasse `Student`, erste Version

geschlossen, die Klasse `Student` zu nennen. Natürlich soll diese Klasse aber sowohl weibliche als auch männliche Student(inn)en modellieren.<sup>1</sup>

Abbildung 7.1 zeigt diesen einfachen Klassenaufbau im UML-Klassendiagramm. Unsere Klasse setzt sich aus zwei Instanzvariablen namens `name` und `nummer` zusammen. Erstgenannte speichert den Namen des Studierenden, Letztere die Matrikelnummer.<sup>2</sup> Wir können diese Klasse nun wie gewohnt instanziiieren (d. h. Objekte aus ihr erzeugen) und diese dann mit Werten belegen:

```
Student studi = new Student();
studi.name = "Karla Karlsson";
studi.nummer = 12345;
```

Bis zu diesem Punkt haben wir an unserer Klasse keine Arbeiten vorgenommen, die wir nicht aus Kapitel 4 schon zur Genüge kennen. Wir wollen diesen Entwurf nun bezüglich unserer vier Grundprinzipien überprüfen:

- Bei unserer Klasse `Student` handelt es sich um eine einzelne Klasse, nicht um eine Hierarchie. Wir haben somit keine weiteren Klassen und können damit keine Eigenschaften in Superklassen auslagern. Das Thema Generalisierung ist also in diesem Beispiel nicht weiter wichtig.
- Ähnliches gilt für die Bereiche Vererbung und Polymorphie. Beide Begriffe spielen erst bei der Arbeit mit mehr als einer Klasse eine wichtige Rolle. Hiermit beschäftigen wir uns aber erst im nächsten Kapitel näher.
- Bleibt also die Frage, ob wir uns bezüglich der Kapselung für ein gutes Modell entschieden haben. Haben wir die interne Struktur unserer Klasse von der Schnittstelle nach außen getrennt? Könnten wir die Instanzvariablen einfach verändern, ohne hiermit Probleme zu verursachen?

An dieser Stelle müssen wir den letzten Punkt leider klar und deutlich verneinen. Unsere Instanzvariablen sind von außen her überall zugänglich. Wir schreiben unsere Werte direkt in sie hinein und lesen sie aus ihnen direkt wieder aus. Wenn wir die Matrikelnummer später in einem `String` ablegen wollen (z. B. weil wir eine Datenbank benutzen, die keine einfachen Datentypen versteht), müssen wir sämtliche Programme überarbeiten, die diese Variablen benutzen. Wir werden deshalb

<sup>1</sup> Wir hoffen, dass unsere *Leserinnen* aufgrund dieser Namenswahl das Buch jetzt nicht empört aus der Hand legen. Wir werden in Übungsaufgabe 7.2 dafür sorgen, dass man sogar explizit zwischen weiblichen und männlichen Studierenden unterscheiden kann.

<sup>2</sup> Eine von der Verwaltung der Hochschule vergebene eindeutige Nummer, unter der die Daten eines Studierenden hinterlegt werden.

im nächsten Abschnitt erfahren, wie wir mit Hilfe sogenannter **Zugriffsmethoden** eine bessere Form der Datenkapselung erreichen.

## 7.2 Instanzmethoden

### 7.2.1 Zugriffsrechte

Wir beginnen damit, unsere Daten vor der Außenwelt zu „verstecken“. Gemäß der Idee des **data hiding** sorgen wir dafür, dass niemand außerhalb der Klasse auf unsere Instanzvariablen zugreifen kann.

Um dieses Ziel zu erreichen, ändern wir die sogenannten **Zugriffsrechte** für die einzelnen Variablen. Momentan haben unsere Variablen die Zugriffsrechte **public**, das heißt, sie sind *öffentlich zugänglich*. Konkret bedeutet es, dass jede andere Klasse auf die Variablen lesenden und schreibenden Zugriff hat. Genau das wollen wir jedoch verhindern!

Um dieses Ziel zu erreichen, setzen wir die Zugriffsrechte von **public** auf **private**. Privater Zugriff ist das genaue Gegenteil von öffentlichem Zugriff: Während bei Ersterem *jede* Klasse auf die Variablen Zugriff hat, kann nun *keine* Klasse mehr auf die Variablen zugreifen, nicht einmal eigene Subklassen. Eine Ausnahme stellt natürlich eben jene Klasse dar, in der die Instanzvariablen definiert sind. Es handelt sich hierbei also wirklich um ihre *privaten* Variablen, die nur der Klasse selbst „gehören“.

Abbildung 7.2 zeigt diese Modifikation im UML-Diagramm. Wir sehen, dass private Variablen durch ein Minuszeichen vor dem Variablennamen markiert werden. Fehlt dieses Symbol oder ist es durch ein Pluszeichen ersetzt, geht man von öffentlichen Zugangsrechten aus.<sup>3</sup>

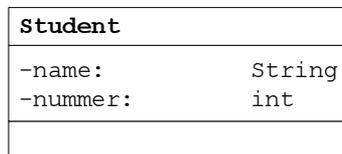


Abbildung 7.2: Die Klasse Student, zweite Version

Die entsprechende Umsetzung in unserem Java-Programm ist relativ einfach: Wir ersetzen lediglich das Schlüsselwort **public** bei den entsprechenden Variablen durch das Schlüsselwort **private**:

```

1  /** Diese Klasse simuliert einen Studenten */
2  public class Student {
3
4      /** Der Name des Studenten */

```

<sup>3</sup> Neben öffentlichem und privatem Zugriff gibt es zwei weitere Formen des Zugriffs (siehe Abschnitt 8.8.2).

```

5  private String name;
6
7  /** Die Matrikelnummer des Studenten */
8  private int nummer;
9  }
10

```

Wenn wir nun (z. B. in einer Klasse namens Schnipsel) wie im vorigen Abschnitt die Instanzvariablen durch einfache Zugriffe der Form

```

studi.name = "Karla Karlsson";
studi.nummer = 12345;

```

setzen wollen, erhalten wir beim Übersetzen eine Fehlermeldung der Form

```

_____ Konsole _____
Variable name in class Student not accessible
    from class Schnipsel.

```

Das heißt: Die Zugriffe wurden verweigert.

## 7.2.2 Was sind Instanzmethoden?

Wie können wir aber nun Daten aus einer Klasse auslesen oder sie setzen, wenn wir hierzu überhaupt nicht berechtigt sind?

Die Antwort haben wir im vorigen Kapitel bereits angedeutet: Wir fügen der Klasse sogenannte **Instanzmethoden** hinzu. Diese Methoden werden ähnlich wie in Kapitel 5 definiert:

```

_____ Syntaxregel _____
public «RUECKGABETYP» «METHODENNAME» ( «PARAMETERLISTE» )
{
    // hier den auszufuehrenden Code einfuegen
}

```

Wenn Sie dies mit der Syntaxregelbox auf Seite 159 vergleichen, stellen Sie als einzigen Unterschied das Wörtchen **static** fest, das unserer Methodendefinition nun fehlt. Durch Weglassen dieses Wortes wird eine Methode an ein spezielles Objekt gebunden, das heißt, sie existiert nur in Zusammenhang mit einer speziellen *Instanz*. Da die Methode aber nun zu einem bestimmten Objekt gehört, hat sie auch Zugriff auf dessen spezielle Eigenschaften – also seine Instanzvariablen.

Abbildung 7.3 zeigt eine entsprechende Erweiterung unseres Klassenmodells im UML-Diagramm. Wir tragen in das untere, bislang leer gebliebene Kästchen unsere Methoden ein. Hierbei verwenden wir als Schreibweise

```
+ «METHODENNAME» ( «PARAMETERLISTE» ) : «RUECKGABETYP»
```

wobei das Pluszeichen wie bei den Instanzvariablen für öffentlichen Zugriff (**public**) steht. Wir definieren also folgende vier Methoden:

<b>Student</b>	
-name:	String
-nummer:	int
+getName():	String
+setName(String):	void
+getNummer():	int
+setNummer(int):	void

Abbildung 7.3: Die Klasse Student, dritte Version

### ■ Die Methode

```
public String getName()
```

soll den Inhalt der Instanzvariablen `name` auslesen und als Resultat der Methode zurückliefern. Unser ausformulierter Java-Code lautet wie folgt:

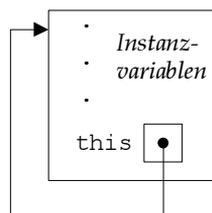
```
/** Gib den Namen des Studenten als String zurueck */
public String getName() {
    return this.name;
}
```

Achten Sie darauf, dass wir die Instanzvariable durch `this.name` angesprochen haben. Das Schlüsselwort `this` liefert innerhalb eines Objektes immer eine Referenz auf das Objekt selbst. Jedes Objekt hat somit quasi eine Komponentenvariable `this`, die eine Referenz auf das Objekt selbst enthält. Wir können also sämtliche Instanzvariablen in der aus Abschnitt 4.2.2 bekannten Form

Syntaxregel

```
«OBJEKTNAME». «VARIABLENNAME»
```

erreichen, indem wir für den Platzhalter `«OBJEKTNAME»` schlicht und ergreifend `this` einsetzen. Abbildung 7.4 verdeutlicht nochmals die Bedeutung der `this`-Referenz.

Abbildung 7.4: Die `this`-Referenz

- Die Methode

```
public void setName(String name)
```

soll nun den Inhalt der Instanzvariablen `name` durch das übergebene `String`-Argument ersetzen:

```
/** Setze den Namen des Studenten auf einen bestimmten Wert */
public void setName(String name) {
    this.name = name;
}
```

Obwohl der Parameter `name` und die Instanzvariable `name` den gleichen Bezeichner haben, gibt es an dieser Stelle keinerlei Konflikte. Der Compiler kann beide Variablen voneinander unterscheiden, da wir die Instanzvariable mit Hilfe der `this`-Referenz ansprechen.

- Die Methode

```
public int getNummer()
```

liest nun den Inhalt unserer `nummer` aus und gibt ihn, genau wie bei der Methode `getName`, als Ergebnis zurück.<sup>4</sup> Ausformuliert lautet das wie folgt:

```
/** Gib die Matrikelnummer des Studenten als Integer zurueck */
public int getNummer() {
    return nummer;
}
```

An dieser Stelle ist zu erwähnen, dass wir in der Methode bewusst auf das Schlüsselwort `this` verzichtet haben. Dennoch lässt sich das Programm übersetzen. Der Grund dafür liegt darin, dass der Übersetzer in einem gewissen Ausmaß „mitdenkt“. Findet er in der Methode oder den übergebenen Parametern keine Variable, die den Namen `nummer` besitzt, sucht er diese unter den Instanzvariablen.

- Zuletzt formulieren wir eine Methode

```
public void setNummer(int n)
```

zum Setzen der Instanzvariablen. Auch hier wollen wir auf die Verwendung der `this`-Referenz verzichten. Um mögliche Namenskonflikte zu vermeiden, haben wir dem übergebenen Parameter einen anderen Namen (`n` statt `nummer`) gegeben:

```
/** Setze die Matrikelnummer des Studenten auf einen
    bestimmten Wert */
public void setNummer(int n) {
    nummer = n;
}
```

<sup>4</sup> Hierbei mag unsere deutsch-englische Namensgebung etwas belustigend klingen, aber wir wollen von Anfang an den bestehenden Konventionen folgen, wonach Methoden, die dem Auslesen von Werten dienen, als **get-Methoden** und Methoden, die Werte einer Instanzvariablen setzen, als **set-Methoden** bezeichnet werden.

Wir haben unsere Klasse `Student` nun bezüglich des Prinzips der Datenkapselung überarbeitet, indem wir sämtliche Instanzvariablen vor der Außenwelt versteckt (data hiding) und den Zugriff von außen nur noch durch get- und set-Methoden ermöglicht haben.

Am Ende dieses Abschnitts könnte man leicht vermuten, dass Instanzmethoden nicht viel mehr als einfachste Schreib/Lesemethoden sind. Wozu also das Prinzip der Datenkapselung? Steckt denn wirklich nicht mehr dahinter?

Wie so oft steckt der Teufel natürlich auch hier wieder einmal im Detail. Instanzmethoden können viel mehr als nur Werte schreiben und lesen. Wir könnten sämtliche bisher definierten Unterprogramme (vgl. Kapitel 5) als Instanzmethoden definieren, wenn wir das Wort `static` weglassen und sie somit an ein Objekt binden<sup>5</sup> – doch das verschafft uns natürlich keinen Vorteil. Die beiden folgenden Abschnitte zeigen jedoch spezielle Anwendungen, die uns die wahre Macht von Instanzmethoden demonstrieren.

### 7.2.3 Instanzmethoden zur Validierung von Eingaben

Die Matrikelnummer eines Studierenden ist eine von der Universitätsverwaltung vergebene Nummer, die einen Studierenden eindeutig identifiziert. Jeder Student bzw. jede Studentin erhält hierbei hochschulintern eindeutig eine solche Nummer zugeordnet. Umgekehrt ist jedoch nicht jede Zahl auch eine gültige Matrikelnummer. Um zu verhindern, dass sich Schreibfehler einschleichen oder ein Student (etwa bei Prüfungsanmeldungen) eine falsche Matrikelnummer angibt, müssen die Nummern gewisse Anforderungen, etwa bezüglich der Quersumme ihrer Ziffern, erfüllen. Eine einfache Form der Prüfung wäre etwa folgende:

*Eine Matrikelnummer ist genau dann gültig, wenn sie fünf Stellen sowie keine führenden Nullen hat und ungerade ist.*

Um also eine ganze Zahl vom Typ `int` auf ihre Gültigkeit zu überprüfen, müssen wir lediglich testen,

- ob die Zahl zwischen 10000 und 99999 liegt und
- ob bei Division durch 2 ein Rest verbleibt, also  $n \% 2 \neq 0$  gilt.

Diese Prüfung in eine Methode zu gießen, ist eine eher leichte Übung. Wir formulieren eine Instanzmethode `validateNummer`, wobei das Wort `validate` für „Überprüfung“ steht. Unsere Methode liefert einen `boolean`-Wert zurück. Ist dieser Wert `true`, so war die Validierung erfolgreich, d. h. wir haben eine für unser Beispiel gültige Matrikelnummer. Ist der Wert jedoch `false`, so haben wir eine ungültige Matrikelnummer vorliegen:

```
/** Pruefe die Matrikelnummer des Studenten
    auf ihre Gueltigkeit */
```

<sup>5</sup> In diesem Fall *müssen* wir allerdings immer ein Objekt erzeugen, um die entsprechenden Methoden aufzurufen.

```

public boolean validateNummer() {
    return
        (nummer >= 10000 && nummer <= 99999 && nummer % 2 != 0);
}

```

Wir können nun also unserem Studierenden nicht nur eine Matrikelnummer zuweisen, sondern auch anschließend überprüfen, ob diese Nummer überhaupt gültig war. Hier stellt sich natürlich die Frage, ob unsere Klasse das nicht auch *automatisch* tun kann? Können wir nicht einfach festlegen, dass wir in unserer Klasse nur gültige Matrikelnummern hinterlegen dürfen?

Die Antwort auf diese Frage lautet wieder einmal: *Ja, das lässt sich machen!* Wir werden unsere `set`-Methode einfach so modifizieren, dass sie den eingegebenen Wert automatisch überprüft:

```

/** Setze die Matrikelnummer des Studenten auf einen best. Wert */
public void setNummer(int n) {
    int alteNummer = nummer;
    nummer = n;
    if (!validateNummer()) { // neue Nummer ist nicht gueltig
        nummer = alteNummer;
    }
}

```

Unsere angepasste Methode durchläuft die Prüfung in mehreren Schritten. Zuerst setzt sie die Matrikelnummer des Studierenden auf den neuen Wert, speichert aber den alten Wert in der Variable `alteNummer` ab. Anschließend ruft sie die `validate`-Methode `validateNummer` auf. War die Validierung erfolgreich, d. h. haben wir eine gültige Matrikelnummer, so wird die Methode beendet. Andernfalls wird die alte Nummer aus `alteNummer` ausgelesen und wieder in die Instanzvariable zurückgeschrieben.

Mit unserer neuen Zugriffsmethode haben wir eine Funktionalität erreicht, die ohne Datenkapselung nicht möglich gewesen wäre. Wir weisen unserem Studenten-Objekt nicht einfach mehr eine Matrikelnummer zu, sondern überprüfen diese automatisch auf ihre Korrektheit. Eine solche Validierung kann uns in vielerlei Hinsicht von Nutzen sein; etwa, um Eingabefehler über die Tastatur zu erkennen. Das Wichtigste bei der ganzen Sache ist allerdings, dass wir für diese Erweiterung keine Veränderung an der alten Schnittstelle vornehmen mussten. Benutzer sind weiterhin in der Lage, Matrikelnummern mit `getNummer` und `setNummer` aus- und einzulesen. Programme, die vielleicht schon für die alte Klasse geschrieben waren, sind auch weiterhin lauffähig – obwohl zum Zeitpunkt der Entwicklung mit einer älteren Version gearbeitet wurde!

## 7.2.4 Instanzmethoden als erweiterte Funktionalität

Neben dem reinen Setzen und Auslesen von Werten können wir Instanzmethoden auch nutzen, um unseren Klassen zusätzliche Eigenschaften und Fähigkeiten zu verleihen, die sie bislang nicht besaßen.

So wollen wir etwa in diesem Abschnitt erreichen, dass Instanzen unserer Klasse eine Beschreibung ihrer selbst ausgeben können. Eine Studentin namens „Susi Sorglos“ mit der Matrikelnummer 92653 soll sich etwa in der Form

```

Susi Sorglos (92653)

```

auf dem Bildschirm darstellen lassen.

Um diesen Zweck zu erfüllen, schreiben wir eine Methode namens `toString`, in der wir aus den Instanzvariablen eine textuelle Beschreibung generieren:

```

/** Gib eine textuelle Beschreibung dieses Studenten aus */
public String toString() {
    return name + " (" + nummer + ')';
}

```

Diese Methode kombiniert die Variablen `name` und `nummer` und erzeugt aus ihnen einen `String`. Instanzieren wir nun in unserem Hauptprogramm ein Objekt der Klasse `Student`,

```

Student studi = new Student();
studi.setName("Karla Karlsson");
studi.setNummer(12345);

```

können wir dieses Objekt durch die einfache Zeile

```

System.out.println(studi.toString());

```

auf dem Bildschirm ausgeben. Unsere Klasse ist somit in der Lage, aus ihrem inneren Zustand selbstständig eine neue Information (hier etwa eine Textbeschreibung) zu erzeugen. Unser reiner Datencontainer hat auf diese Weise ein gewisses Maß an Selbstständigkeit erreicht!

In Abschnitt 8.4 werden wir übrigens feststellen, dass für obige Bildschirmausgabe auch die Zeile

```

System.out.println(studi);

```

ausgereicht hätte. Grund hierfür ist der Umstand, dass jedes Objekt eine Methode `toString` besitzt. Wenn wir ein Objekt mit der `println`-Methode auszugeben versuchen, ruft das druckende Objekt<sup>6</sup> genau diese `toString`-Methode auf. In unserer Klasse `Student` haben wir diese Methode überschrieben, das heißt, wir haben mit Hilfe der Polymorphie eine maßgeschneiderte Ausgabe für unsere Klasse modelliert.

## 7.3 Statische Komponenten einer Klasse

Wir haben im letzten Abschnitt mit den Instanzvariablen und -methoden ein wichtiges Gebiet des objektorientierten Programmierens kennengelernt. Die Möglichkeit, Variablen oder sogar ganze Methoden einem bestimmten Objekt zuzuordnen zu können, hat uns Perspektiven erschlossen, die wir mit unseren bisherigen Programmiererfahrungen nicht sahen.

<sup>6</sup> Auch die Methode `println` ist Instanzmethode eines Objektes, des sogenannten Ausgabestroms. Das Objekt `System.out` ist ein solcher Strom.

Hier stellt sich jedoch die Frage, wie sich das früher Gelernte mit diesen neuen Technologien vereinbaren lässt. Instanzmethoden ähneln vom Aufbau her zwar unseren Methoden aus Kapitel 5, sind aber schon insofern vollkommen verschieden, als sie zu einem speziellen Objekt gehören. Müssen wir also unser ganzes Wissen über Bord werfen?

Natürlich nicht! Aus objektorientierter Sicht handelt es sich bei unseren früher verwendeten Methoden um die sogenannten **Klassenmethoden**, auch **statische Methoden** genannt. In diesem Kapitel haben wir bisher nur Instanzmethoden definiert – also Methoden, die einer ganz bestimmten *Instanz* einer Klasse gehören. Klassenmethoden wiederum folgen dem gleichen Schema. Statt einer einzelnen Instanz gehören sie allerdings der gesamten *Klasse*, das heißt, alle Objekte teilen sich eine einzige Methode. Diese Methode existiert vielmehr sogar, wenn *kein einziges Objekt* zu unserer Klasse existiert.

Unsere früheren Programme haben diesen Umstand ausgenutzt, um Ihnen als Anfänger die objektorientierte Sichtweise zu ersparen. Wir haben Klassen definiert (jedes unserer Programme war eine Klassendefinition) und diese nur mit Klassenmethoden gefüllt. Obwohl wir nie eine Instanz dieser Klassen erzeugt haben, konnten wir die einzelnen Methoden problemlos aufrufen. Jetzt, da Sie im Begriff sind, ein OO-Profi zu werden, wissen Sie es natürlich besser. Nehmen Sie eines Ihrer alten Programme, und versuchen Sie, mit Hilfe des `new`-Operators eine Instanz zu bilden. Es wird Ihnen gelingen.

### 7.3.1 Klassenvariablen und -methoden

Am ehesten wird der Nutzen von statischen Komponenten deutlich, wenn wir mit einem konkreten Anwendungsfall beginnen. Unsere Klasse `Student` besitzt momentan zwei Datenelemente, nämlich den Namen und die Matrikelnummer des Studenten bzw. der Studentin.

Aus statistischer Sicht mag es vielleicht interessant sein, die Zahl der instanziierten Studentenobjekte zu zählen. Wird beispielsweise eine neue Universität eröffnet und verwendet diese von Anfang an unsere Studentenverwaltung, so könnte man aus dieser Variablen erfahren, wie viele Studierende es im Laufe der Geschichte an dieser Universität gegeben hat.

Nun stehen wir jedoch vor dem Problem, dass wir diese Variable – wir wollen sie der Einfachheit halber einmal `zaehler` nennen – keiner speziellen Instanz unserer Klasse zuordnen können. Vielmehr handelt es sich hierbei um eine Eigenschaft, die zu der Gesamtheit *aller* Studentenobjekte gehört. Die Anzahl aller Studenten macht keine Aussage über einen speziellen Studenten, sondern über die Studenten an sich. Sie sollte daher *allen* Studenten angehören, sprich, eine **statische Komponente** der Klasse `Student` sein.

Wir erzeugen deshalb eine Variable, die keiner bestimmten Instanz, sondern der gesamten Klasse gehört, gemäß der folgenden Regel:<sup>7</sup>

<sup>7</sup> Der initiale Wert könnte an dieser Stelle auch wegfallen.

## Syntaxregel

```
private static <TYP> <VARIABLENNAME> = <INITIALWERT>;
```

Wir stellen fest, dass sich die Definition von Klassenvariablen nicht sehr von dem unterscheidet, was wir in Abschnitt 4.2 über Instanzvariablen gelernt haben. Mit Hilfe des Wortes **private** schützen wir unsere Variable vor Zugriffen von außerhalb. Typ, Variablenname und Initialwert sind uns ebenfalls bekannt und würden im Fall unseres Zählers zu folgender Definition führen:

```
private static int zaehler = 0;
```

Neu ist für uns an dieser Stelle lediglich das Schlüsselwort **static**, das wir bislang nur aus unseren Methoden im ersten Teil des Buches kannten. Dieses Wort weist eine Variable oder Methode als statische Komponente einer Klasse aus. Wenn wir eine Variable also als **static** beschreiben, gehört sie allen Instanzen einer Klasse zugleich. Wir können den Inhalt der Variablen auslesen, indem wir eine entsprechende get-Methode definieren:

```
/** Gib die Zahl der erzeugten Studentenobjekte zurueck */
public static int getZaehler() {
    return zaehler;
}
```

Beachten Sie hierbei, dass wir auch bei dieser Methode das Schlüsselwort **static** verwendet haben, die Methode also der Klasse, nicht den Objekten zugeordnet haben. Die Methode `getZaehler` ist also eine Klassenmethode, die wir etwa durch einen Aufruf der Form

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

aus jedem beliebigen Programm aufrufen können, ohne eine konkrete Referenz auf ein Studentenobjekt zu besitzen.

Wie können wir aber nun ein Objekt so erzeugen, dass der interne (private) Zähler korrekt erhöht wird? Zu diesem Zweck entwerfen wir eine Methode `createStudent`, die uns ein neues Studentenobjekt erzeugt. Auch diese Methode müssen wir statisch machen, da sie schließlich gerade zum Erzeugen von Objekten benutzt werden soll, also nicht aus einem Objekt heraus aufgerufen wird:

```
/** Erzeugt ein neues Studentenobjekt */
public static Student createStudent() {
    zaehler++; // erhoehe den Zaehler
    return new Student();
}
```

Unsere Methode zählt bei Aufruf zuerst die Variable `zaehler` hoch und aktualisiert somit deren Stand. Im zweiten Schritt wird mit Hilfe des **new**-Operators ein neues Objekt erzeugt und dieses als Ergebnis zurückgegeben. Nun können wir in unseren Programmen Studentenobjekte durch einen einfachen Methodenaufruf erzeugen lassen und somit den Zähler korrekt aktualisieren:

```
Student studi = Student.createStudent();
System.out.println(Student.getZaehler());
```

Leider hat diese Methode, neue Studentenobjekte zu erzeugen, einen gewaltigen Pferdefuß: bei älteren Programmen, die ihre Objekte noch mit Hilfe des `new`-Operators erzeugen, funktioniert der Zähler nicht korrekt. Wir laufen auch immer Gefahr, dass andere Programmierer, die unsere Klasse `Student` benutzen, den Fehler begehen, Objekte direkt zu erzeugen. Wir werden in Abschnitt 7.4.1 jedoch eine Methode kennenlernen, diese Probleme auf elegante Art und Weise zu lösen.

<b>Student</b>	
-name:	String
-nummer:	int
-zaehler:	<u>int</u>
<hr/>	
+getName():	String
+setName(String):	void
+getNummer():	int
+setNummer(int):	void
+validateNummer():	boolean
+toString():	String
+getZaehler():	<u>int</u>
+createStudent():	Student

Abbildung 7.5: Die Klasse `Student`, mit Objektzähler

Jetzt werfen wir noch einen Blick auf unsere gewachsene Klasse `Student` im UML-Klassendiagramm (Abbildung 7.5). Klassenmethoden und Klassenvariablen werden im UML-Diagramm durch Unterstreichung gekennzeichnet. Wir stellen fest, dass wir – obwohl unsere Klasse inzwischen beträchtlich gewachsen ist – durch die Grafik noch immer einen schnellen Überblick über die Komponenten erhalten, aus denen sich die Klasse zusammensetzt. Oft ist es sinnvoll, private Variablen nicht in das UML-Diagramm einzuzichnen, denn für den Entwurf eines Systems von Klassen (hierzu dient uns UML) ist es letztendlich ausreichend zu wissen, welche Schnittstelle eine Klasse nach außen zu bieten hat. Dadurch lassen sich große Klassen übersichtlicher gestalten. Auch wir wollen nachfolgend gelegentlich von dieser Regel Gebrauch machen.

### 7.3.2 Klassenkonstanten

Wie wir aus Abschnitt 3.4.1 wissen, ist es möglich, mit Hilfe des Schlüsselwortes `final` aus „normalen“ Variablen `final`-Variablen zu machen, sie also zu symbolischen Konstanten werden zu lassen. Das gilt natürlich nicht nur für lokale Variablen innerhalb einer Methode, sondern auch für Klassenvariablen, die durch das vorangestellte `final` zu Klassenkonstanten werden.

Konstanten werden in Java häufig dann eingesetzt, wenn man eine nichtssagende Codierung durch eine selbst erklärende Begrifflichkeit erklären will oder wenn

man schwer zu merkende Werte wie etwa den Wert der mathematischen Konstanten  $\pi$  (gesprochen „pi“, etwa 3.14... ) benennen will. Hierbei gilt ja als Konvention, dass wir Konstanten in unseren Programmen immer groß schreiben. Im Falle von  $\pi$  verwendet Java die Bezeichnung `PI`. Da diese Konstante in der Klasse `Math` deklariert ist, können wir sie bekanntlich über `Math.PI` ansprechen.

Auch für die Modellierung unserer Studierenden können wir Klassenkonstanten einsetzen. Wenn sich ein Student bzw. eine Studentin für ein bestimmtes *Studienfach* an einer Hochschule einschreibt, wird dieses Fach in den Systemen vieler Hochschulverwaltungen mit einer bestimmten Nummer identifiziert.

**Tabelle 7.1:** Zuordnung Studienfach – Verwaltungsnummer

Studienfach	Verwaltungsnummer
Architektur	3
Biologie	5
Germanistik	7
Geschichte	6
Informatik	2
Mathematik	1
Physik	9
Politologie	8
Wirtschaftswissenschaften	4

Tabelle 7.1 zeigt eine derartige fiktive Nummerierung. Wir wollen diese Nummern verwenden und erweitern unsere Klasse `Student` um eine ganzzahlige Variable `fach` (inklusive get- und set-Methoden):

```
/** Studienfach des Studenten */
private int fach;

/** Gib das Studienfach des Studenten als Integer zurueck */
public int getFach() {
    return fach;
}

/** Setze das Studienfach des Studenten auf einen bestimmten Wert */
public void setFach(int fach) {
    this.fach = fach;
}
```

Um unsere Variable nun mit einem der obigen Werte zu füllen, definieren wir in unserer Klasse `Student` einige *finale* Klassenvariablen:

```
/** Konstante fuer das Studienfach Mathematik */
public static final int MATHEMATIKSTUDIUM = 1;

/** Konstante fuer das Studienfach Informatik */
public static final int INFORMATIKSTUDIUM = 2;

/** Konstante fuer das Studienfach Architektur */
public static final int ARCHITEKTURSTUDIUM = 3;
```

```

/** Konstante fuer das Studienfach Wirtschaftswissenschaften */
public static final int WIRTSCHAFTLICHESSTUDIUM = 4;

/** Konstante fuer das Studienfach Biologie */
public static final int BIOLOGIESTUDIUM = 5;

/** Konstante fuer das Studienfach Geschichte */
public static final int GESCHICHTSSTUDIUM = 6;

/** Konstante fuer das Studienfach Germanistik */
public static final int GERMANISTIKSTUDIUM = 7;

/** Konstante fuer das Studienfach Politologie */
public static final int POLITOLOGIESTUDIUM = 8;

/** Konstante fuer das Studienfach Physik */
public static final int PHYSIKSTUDIUM = 9;

```

Jede dieser Variablen stellt nun eine ganze Zahl dar, die wir als statische Klassenvariable etwa durch die Codezeile

```
Student.INFORMATIKSTUDIUM
```

ansprechen können. Ein Versuch, den Inhalt der Variablen nachträglich abzuändern, schlägt fehl: So liefert etwa die Zeile

```
Student.INFORMATIKSTUDIUM = 23;
```

eine Fehlermeldung der Form

————— Konsole —————

```
Can't assign a value to a final variable: INFORMATIKSTUDIUM
Student.INFORMATIKSTUDIUM = 23;
```

Wir haben also konstante, *unveränderliche* Werte geschaffen, mit denen wir unsere Programme lesbarer und sicherer bezüglich Tippfehlern machen können. Verdeutlichen können wir uns dies, indem wir zum Beispiel die Ausgabe unserer `toString`-Methode um einen (mehr oder weniger) sinnvollen Spruch erweitern, der die verschiedenen Studiengänge charakterisiert. Ohne die Ziffern in Tabelle 7.1 nachschlagen zu müssen, gelingt uns das mühelos:

```

/** Gib eine textuelle Beschreibung dieses Studenten zurueck */
public String toString() {
    String res = name + " (" + nummer + ")\n";
    switch(fach) {
        case MATHEMATIKSTUDIUM:
            return res + " ein Mathestudent " +
                "(oder auch zwei, oder drei).";
        case INFORMATIKSTUDIUM:
            return res + " ein Informatikstudent.";
        case ARCHITEKTURSTUDIUM:
            return res + " angehender Architekt.";
        case WIRTSCHAFTLICHESSTUDIUM:
            return res + " ein Wirtschaftswissenschaftler.";
        case BIOLOGIESTUDIUM:

```

```

    return res + " Biologie ist seine Staerke.";
case GESCHICHTSSTUDIUM:
    return res + " sollte Geschichte nicht mit Geschichten " +
        "verwechseln.";
case GERMANISTIKSTUDIUM:
    return res + " wird einmal Germanist gewesen tun sein.";
case POLITOLOGIESTUDIUM:
    return res + " kommt bestimmt einmal in den Bundestag.";
case PHYSIKSTUDIUM:
    return res + " studiert schon relativ lange Physik.";
default:
    return res + " keine Ahnung, was der Mann studiert.";
}
}

```

## 7.4 Instanziierung und Initialisierung

In diesem Abschnitt beschäftigen wir uns mit der Frage, wie wir Einfluss auf den Erzeugungsprozess eines Objektes nehmen können. Bereits auf Seite 221 hatten wir festgestellt, dass es uns gelingen müsste, in irgendeiner Form Einfluss auf den `new`-Operator zu nehmen. Unsere Methode `createStudent` und der besagte Operator taten schließlich nicht mehr das Gleiche; nur die `create`-Methode zählte unseren Zähler korrekt hoch.

Nun lernen wir Mittel und Wege kennen, unser Vorhaben in die Tat umzusetzen.

### 7.4.1 Konstruktoren

Erinnern wir uns: Bevor wir die Methode `createStudent` erschufen, hatten wir unsere Objekte durch eine Zeile der Form

```
Student studi = new Student();
```

instanziiert, wobei der `new`-Operator angewendet wurde, entsprechend der bereits auf Seite 144 beschriebenen Regel

Syntaxregel

```
«INSTANZNAME» = new «KLASSENNAME» ();
```

Wenn wir uns diese Zeile etwas genauer ansehen, so fallen uns die runden Klammern am Ende auf. Diese Klammern kennen wir bislang nur vom Aufruf von Methoden her! Ruft die Verwendung des `new`-Operators etwa ebenfalls eine Methode auf?

Tatsächlich ist der Vorgang des „Erbauens“ eines Objektes etwas komplizierter. In Abschnitt 7.4.4 gehen wir auf die tatsächlichen Mechanismen näher ein. Wir können aber an dieser Stelle schon vereinfacht sagen, dass am Ende dieses Vorganges tatsächlich eine Art von Methode aufgerufen wird: der sogenannte **Konstruktor**. Konstruktoren sind keine Methoden im eigentlichen Sinn, da sie nicht – wie etwa Klassen- oder Instanzmethoden – explizit aufgerufen werden. Sie haben auch

keinen Rückgabetyt (nicht einmal `void`). Die Definition des Konstruktors erfolgt nach dem Schema:<sup>8</sup>

Syntaxregel

```
public «KLASSENNAME» ( «PARAMETERLISTE» )
{
    // hier den auszufuehrenden Code einfuegen
}
```

Aus dieser Regel schließen wir zwei wichtige Dinge:

1. Der Konstruktor heißt immer so wie die Klasse.
2. Der Konstruktor verfügt über eine Parameterliste, in der wir Argumente vereinbaren können (was wir im nächsten Abschnitt auch tun werden).

Mit dieser einfachen Regel können wir nun also Einfluss auf die Erzeugung unseres Objektes nehmen – genau das wollen wir auch tun. Wir beginnen mit dem einfachsten Fall: einem Konstruktor, der keinerlei Argumente besitzt und absolut nichts tut:

```
public Student () {}
```

Dieser Konstruktor, manchmal auch als **Standard-Konstruktor** oder **Default-Konstruktor** bezeichnet, wurde bisher vom Übersetzer automatisch erzeugt. Er wird vom System aufgerufen, wenn wir z. B. mit

```
Student studi = new Student();
```

ein Objekt instanziiert. Der Standard-Konstruktor wird nur angelegt, wenn man keine eigenen Konstruktoren anlegt – und nur dann! Wenn wir also im Folgenden eigene Konstruktoren für unsere Klassen definieren, wird für diese vom System kein Standard-Konstruktor mehr angelegt.

Der folgende Konstruktor aktualisiert unsere Klassenvariable `zaehler`, indem er sie automatisch um den Wert 1 erhöht:

```
/** Argumentloser Konstruktor */
public Student () {
    zaehler++;
}
```

Wenn wir nun mit Hilfe des `new`-Operators ein Studentenobjekt erzeugen, so wird durch den Aufruf des Konstruktors der Zähler automatisch aktualisiert. Wir können uns also die zusätzliche Erhöhung in unserer `createStudent`-Methode sparen:

```
/** Erzeugt ein neues Studentenobjekt */
public static Student createStudent () {
    return new Student ();
}
```

<sup>8</sup> Hierbei kann man statt `public` natürlich auch andere Zugriffsrechte vergeben.

Tatsächlich stellen wir fest, dass es nun wieder keinen Unterschied mehr bedeutet, ob wir unsere Objekte mit `new` oder mit `createStudent` erzeugen. Der Prozess der Instanziierung wurde somit vereinheitlicht, die auf Seite 221 angemahnte Abwärtskompatibilität<sup>9</sup> wiederhergestellt.

## 7.4.2 Überladen von Konstruktoren

Wir wollen neben den bisher vorhandenen Daten eine weitere Instanzvariable definieren: In der ganzzahligen Variable `geburtsjahr` möchten wir das Jahr hinterlegen, in dem der betreffende Student bzw. die betreffende Studentin geboren wurde.

```
/** Geburtsjahr eines Studenten */  
private int geburtsjahr;
```

Die Variable `geburtsjahr` soll im Gegensatz zu unseren bisherigen Instanzvariablen jedoch eine Besonderheit besitzen. Wir definieren zwar eine get-Methode, mit der wir den Wert der Variablen auslesen können

```
/** Gib das Geburtsjahr des Studenten als Integer zurueck */  
public int getGeburtsjahr() {  
    return geburtsjahr;  
}
```

formulieren aber keine set-Methode, um den entsprechenden Wert zu setzen bzw. zu verändern. Der Grund hierfür ist relativ einfach. Alle bisher definierten Werte können sich ändern. Der Student bzw. die Studentin kann heiraten und den Namen seines Partners annehmen. Er kann sein Studienfach oder die Universität wechseln, was den Inhalt der Variablen `fach` und `nummer` beeinflussen würde. Nur eines kann unser(e) Student(in) niemals verändern: das Jahr, in dem er bzw. sie geboren wurde.

Wir wollen also den Inhalt der Variablen beim Erzeugen festlegen. Danach soll diese Variable von außen nicht mehr verändert werden können. Im Fall unseres argumentlosen Konstruktors sähe dies etwa wie folgt aus:

```
/** Argumentloser Konstruktor */  
public Student() {  
    zaehler++;  
    geburtsjahr = 2000;  
}
```

Wir setzen also den Inhalt unserer Variablen auf einen Standardwert, das Jahr 2000, was natürlich insbesondere deshalb unbefriedigend ist, weil nur ein geringer Teil der heute Studierenden in diesem Jahr geboren wurde. Deshalb definieren wir einen zweiten Konstruktor, in dem wir das Geburtsjahr als einen Parameter übergeben:

```
/** Konstruktor, bei dem sich das Geburtsjahr setzen laesst. */  
public Student(int geburtsjahr) {  
    zaehler++;
```

<sup>9</sup> Dies bedeutet, dass Programme, die für ältere Versionen unserer Klasse `Student` geschrieben wurden, auch mit unserer neuen Version funktionieren.

```

        this.geburtsjahr = geburtsjahr;
    }

```

Wir haben unseren Konstruktor also **überladen**, wie wir es schon in Abschnitt 5.1.5 mit Methoden gemacht haben. Analog dazu unterscheidet Java auch die Konstruktoren einer Klasse

- anhand der *Zahl* der Argumente,
- anhand des *Typs* der Argumente und
- anhand der *Position* der Argumente.

Wir können beim Überladen also den gleichen Regeln folgen – unsere Definition des zweiten Konstruktors war somit korrekt – und ihn wie gewohnt verwenden, indem wir das Geburtsjahr innerhalb der Klammern des **new**-Operators mit auf-führen. So generiert etwa die folgende Zeile einen im Jahr 1999 geborenen Studenten:

```

Student studi = new Student(1999);

```

In den Übungsaufgaben beschäftigen wir uns noch einmal mit dem Überladen von Konstruktoren. Da Sie diesen Mechanismus jedoch bereits von den Methoden her kennen, stellt er bei Weitem kein Hexenwerk mehr dar.

An diesem Punkt jedoch noch eine kleine Anmerkung, die die Programmierung insbesondere von vielen Konstruktoren in einer Klasse vereinfacht. Wenn wir einen Blick auf unsere beiden Konstruktoren werfen, so stellen wir fest, dass sich diese in ihrer Struktur sehr ähneln:

```

/** Argumentloser Konstruktor */
public Student() {
    zaehler++;
    geburtsjahr = 2000;
}

/** Konstruktor, bei dem sich das Geburtsjahr setzen laesst. */
public Student(int geburtsjahr) {
    zaehler++;
    this.geburtsjahr = geburtsjahr;
}

```

Beide Konstruktoren erhöhen zuerst den Zähler und setzen dann die Variable `geburtsjahr` auf einen vorbestimmten Wert. Unser argumentloser Konstruktor ist hierbei gewissermaßen ein „Spezialfall“ des anderen Konstruktors, da er das Geburtsjahr nicht übergeben bekommt, sondern auf einen festen Wert setzt. Wir können diesen Konstruktor also einfacher formulieren, indem wir ihn auf seinen „großen Bruder“ zurückführen:

```

public Student() {
    this(2000);
}

```

Hierbei verwenden wir das Schlüsselwort **this**, um einen Konstruktor aus einem anderen Konstruktor heraus aufzurufen. Dieser Vorgang kann nur innerhalb von Konstruktoren und auch dort nur einmal geschehen – nämlich *als allererster Befehl innerhalb des Konstruktors*. Dieser eine erlaubte Aufruf gestattet es uns jedoch, nicht

jede einzelne Codezeile doppelt formulieren zu müssen. Insbesondere bei großen und aufwändigen Konstruktoren erspart uns das eine Menge Arbeit.

### 7.4.3 Der statische Initialisierer

Spätestens seit Gaston Leroux' Erfolgsroman wissen wir es alle: Eine wirklich erfolgreiche Institution benötigt ein *Phantom*. Angefangen mit dem Phantom der (Pariser) Oper übertrug sich dieser Trend mittels Hollywoodstreifen auf Filmstudios, Krankenhäuser und sonstige öffentliche Gebäude.

Wir wollen dieser Entwicklung Rechnung tragen und auch unserer Universität ein Phantom spendieren. Dieses Phantom soll eine konstante Klassenvariable sein und unter dem Namen `Student.PHANTOM` angesprochen werden können:

```
/** Diese Konstante repraesentiert
    das Phantom des Campus */
public static final Student PHANTOM;
```

Unser Phantom soll die Matrikelnummer `-12345` besitzen, auf den Namen „Erik le Phant“ hören und im Jahr 1735 geboren sein. Ferner soll er offiziell gar nicht existieren, das heißt, seine Existenz soll den Studentenzähler nicht beeinflussen.

An dieser Stelle bekommen wir mit der Initialisierung unserer Konstanten anscheinend massive Probleme:

1. Die Konstante `Student.PHANTOM` soll zusammen mit der Klasse existieren, ohne dass wir sie in unserem Hauptprogramm erst in irgendeiner Form initialisieren müssen.
2. Die Zahl `-12345` ist keine gültige Matrikelnummer. Unsere `setNummer`-Methode würde diesen Wert nicht als gültige Eingabe akzeptieren. Wir können diesen Wert also von außen nicht setzen.
3. Jedes Mal, wenn wir mit dem `new`-Operator ein Objekt erzeugen, wird die interne Variable `zaehler` automatisch hochgezählt. Da wir aber von außen nur lesenden Zugriff auf den Zähler haben, können wir diesen Umstand nicht rückgängig machen.

Wie wir sehen, kommen wir an dieser Stelle mit einer Initialisierung „von außen“ nicht weiter. Wir benötigen eine Möglichkeit, statische Komponenten einer Klasse beim Systemstart<sup>10</sup> automatisch zu initialisieren. Hierfür verwenden wir den sogenannten **statischen Initialisierer**, umgangssprachlich oft einfach `static`-Block genannt.<sup>11</sup> Statische Initialisierer werden nach folgender Regel erschaffen:

Syntaxregel

```
static {
    // hier den auszufuehrenden Code einfuegen
}
```

<sup>10</sup> Genauer gesagt, wenn wir die Klasse zum ersten Mal verwenden.

<sup>11</sup> Die offizielle englischsprachige Bezeichnung aus der Java Language Specification ist übrigens **static initializer**.

In einer Klasse können beliebig viele `static`-Blöcke auftreten. Sobald die Klasse dem Java-System bekannt gemacht wird (das sogenannte Laden der Klasse), werden die `static`-Blöcke in der Reihenfolge ausgeführt, in der sie im Programmcode auftauchen. Hierbei gelten die folgenden wichtigen Regeln:

- *Statische Initialisierer haben nur Zugriff auf statische Komponenten einer Klasse.* Sie können keine Instanzvariablen manipulieren, da diese nur innerhalb von Objekten existieren. Natürlich mit der Ausnahme, dass Sie innerhalb des `static`-Blocks ein Objekt, mit dem Sie arbeiten wollen, erzeugt haben.
- *Statische Initialisierer haben Zugriff auf alle (auch private) Teile einer Klasse.* Im Gegensatz zu einer Initialisierung „von außen“ befinden wir uns beim `static`-Block innerhalb der Klasse. Wir können selbst die für andere unsichtbaren Bereiche einsehen und manipulieren.
- *Statische Initialisierer haben nur Zugriff auf statische Komponenten, die im Programmcode vor ihnen definiert wurden.* Wenn Sie also eine statische Variable durch einen `static`-Block initialisieren wollen, muss der `static`-Block *nach* der Definition der Klassenvariable erfolgen.

Wir wollen diese Regeln nun berücksichtigen und unsere Konstante initialisieren. Hierzu erzeugen wir einen `static`-Block, den wir (um bezüglich der Reihenfolge auf Nummer sicher zu gehen) an das Ende unserer Klassendefinition setzen:

```

/* =====
   STATISCHE INITIALISIERUNG
   =====
*/

static {
    // Erzeuge das PHANTOM-Objekt
    PHANTOM = new Student(1735);
    PHANTOM.name = "Erik le Phant";
    PHANTOM.nummer = -12345;
    // Setze den Zaehler wieder zurueck
    zaehler = 0;
}

```

Gehen wir nun die einzelnen Zeilen unseres statischen Initialisierers genauer durch. In der ersten Zeile

```
PHANTOM = new Student(1735);
```

haben wir mit Hilfe des `new`-Operators ein neues Studentenobjekt (mit Geburtsdatum 1735) erzeugt und der Konstante `PHANTOM` zugewiesen. Unsere Konstante ist somit belegt und kann nicht mehr verändert werden.

In der folgenden Zeile werden wir nun anscheinend gegen diesen Grundsatz verstoßen. Wir nutzen unseren direkten Zugriff auf die `private` Instanzvariable `name` aus und setzen ihren Inhalt auf den Namen „Erik le Phant“:

```
PHANTOM.name = "Erik le Phant";
```

Haben wir somit gegen das Gesetz, finale Variablen nicht mehr verändern zu können, verstoßen? Die Antwort lautet *nein*, und ihre Begründung liegt wieder einmal in dem Umstand, dass es sich bei Klassen um Referenzdatentypen handelt.

In unserer finalen Variablen `PHANTOM` steht nämlich nicht das Objekt selbst, sondern eine *Referenz*, also ein Verweis auf das tatsächliche Objekt. Diese Referenz ist konstant, das heißt, unsere Variable wird immer auf ein und dasselbe Studentenobjekt verweisen. Das Objekt selbst ist jedoch ein ganz „normaler“ Student und kann als solcher von uns auch manipuliert<sup>12</sup> werden.

In der folgenden Zeile nutzen wir unseren Zugriff auf private Komponenten aus, um den Wert der Matrikelnummer auf `-12345` zu setzen:

```
PHANTOM.nummer = -12345;
```

Da wir hierbei den Wert der Variablen direkt setzen, also nicht über die `set`-Methode gehen, wird die `validate`-Methode für unsere Variable `nummer` nicht aufgerufen. Wir können den Inhalt unserer Variablen somit ungestört auf einen (eigentlich nicht erlaubten) Wert setzen.

Nun kümmern wir uns noch um den statischen Objektzähler. Dass der `new`-Operator unsere Variable `zaehler` auf den Wert 1 gesetzt hat, konnten wir nicht verhindern. Wir machen dies im Nachhinein jedoch wieder rückgängig, indem wir unseren Objektzähler einfach wieder auf null setzen:

```
zaehler = 0;
```

Wir haben innerhalb weniger Zeilen einen statischen Initialisierer geschaffen, der

1. die Konstante `Student.PHANTOM` automatisch initialisiert, sobald die Klasse benutzt wird,
2. die Matrikelnummer auf den (eigentlich inkorrekten) Wert `-12345` setzt und somit die automatische Prüfung umgeht und
3. den `zaehler` wieder zurücksetzt, sodass unser Phantom in der Objektzählung nicht erscheint.

Unsere Probleme sind also gelöst.

#### 7.4.4 Der Mechanismus der Objekterzeugung

Wir haben in den letzten Abschnitten verschiedene Mechanismen kennengelernt, um Klassen- und Instanzvariablen mit Werten zu belegen. Unsere Konstrukto-ren spielen hierbei eine wichtige Rolle, sind aber nicht die einzigen wichtigen Bestandteile des Instanziierungsprozesses. Wenn wir beispielsweise unserer Variablen `name` in ihrer Definition

```
private String name = "DummyStudent";
```

einen Initialisierer hinzufügen und ferner im Konstruktor die Zeile

```
this.name = "Namenlos";
```

hinzufügen – auf welchen Wert wird unser Studentename bei der Initialisierung dann gesetzt? Ist er dann „Namenlos“ oder ein „DummyStudent“?

<sup>12</sup> Natürlich lehnen wir jegliche Manipulation von Studierenden grundsätzlich ab. Das Beispiel dient lediglich zu Ausbildungszwecken und erfolgt auch nur an unserem Phantom.

Um diese Frage beantworten zu können, sollte man (zumindest in groben Zügen) den Mechanismus verstehen, mit dem unsere Objekte erzeugt werden. Wir werden uns deshalb in diesem Abschnitt näher damit beschäftigen. Zu diesem Zweck betrachten wir zwei einfache Klassen, die in Abbildung 7.6 skizziert sind.

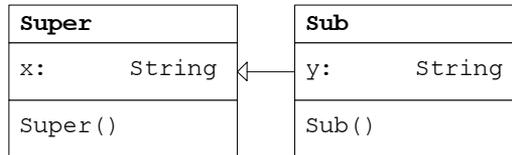


Abbildung 7.6: Beispielklassen für Abschnitt 7.4.4

Die Klassen `Super` und `Sub` stehen in einer verwandtschaftlichen Beziehung zueinander: `Sub` ist die Subklasse von `Super`. Sie erbt somit deren Eigenschaften, das heißt in diesem Fall die öffentliche Instanzvariable `x`. Ferner wird in `Sub` eine zweite Instanzvariable namens `y` definiert, die also die Funktionalität der Superklasse um ein weiteres Datum ergänzt. Im Folgenden werden wir uns mit der Frage beschäftigen, welche Aktionen innerhalb des Systems beim Aufruf eines Konstruktors<sup>13</sup> der Subklasse in der Form

```
new Sub();
```

ausgelöst werden.

Wir betrachten erst einmal die Theorie. Ein Objekt wird vom System in den folgenden Schritten angelegt:

1. Das System organisiert Speicherplatz, um den Inhalt sämtlicher Instanzvariablen abspeichern zu können, die innerhalb des Objektes benötigt werden. In unserem Fall wären das für ein `Sub`-Objekt also die Variablen `x` und `y`. Sollte nicht genug Speicher vorhanden sein, entsteht ein sogenannter `OutOfMemory`-Fehler, der das gesamte Java-System zum Absturz bringen kann. In Ihren Programmen wird dies aber normalerweise nicht der Fall sein.
2. Die Instanzvariablen werden mit ihren Standardwerten (Default-Werten, gemäß Tabelle 7.2) belegt.
3. Der Konstruktor wird mit den übergebenen Werten aufgerufen. Hierbei wird in Java nach dem folgenden System vorgegangen:
  - (a) Ist die erste Anweisung des Konstruktorrumpfes *kein* Aufruf eines anderen Konstruktors (also weder `this(...)` noch `super(...)`), so wird implizit der Aufruf des Standard-Konstruktors der direkten Superklasse

<sup>13</sup> Die Konstruktoren werden im UML-Diagramm wie Methoden dargestellt, allerdings lässt man den Rückgabetypp weg. Jede unserer beiden Klassen besitzt also einen argumentlosen Konstruktor.

Tabelle 7.2: Default-Werte von Instanzvariablen

Datentyp	Standardwert
<code>byte</code>	<code>(byte) 0</code>
<code>short</code>	<code>(short) 0</code>
<code>int</code>	<code>0</code>
<code>long</code>	<code>0L</code>
<code>float</code>	<code>0.0f</code>
<code>double</code>	<code>0.0d</code>
<code>char</code>	<code>(char) 0</code>
<code>boolean</code>	<code>false</code>
Referenzdatentyp	<code>null</code>

`super()` ergänzt und auch aufgerufen. Unmittelbar nach diesem impliziten Aufruf werden alle in der Klasse mit Initialisierern deklarierten Instanzvariablen mit den entsprechenden Werten initialisiert. Haben wir etwa in unserer Klasse `Sub` die Variable `y` in der Form

```
public String y = "vor Sub-Konstruktor";
```

definiert, lautet der Wert von `y` nun also `vor Sub-Konstruktor`. Erst danach werden die restlichen Anweisungen des Konstruktorrumpfes ausgeführt. Auf das Schlüsselwort `super` gehen wir im nächsten Kapitel noch genauer ein.

- (b) Ist die erste Anweisung innerhalb des Konstruktorrumpfes von der Form `super(...)`, wird der entsprechende Konstruktor der direkten Superklasse aufgerufen. Danach werden alle in der Klasse mit Initialisierern deklarierten Instanzvariablen mit den entsprechenden Werten initialisiert und die restlichen Anweisungen des Konstruktorrumpfes ausgeführt.
- (c) Ist die erste Anweisung innerhalb des Konstruktorrumpfes von der Form `this(...)`, wird der entsprechende Konstruktor derselben Klasse aufgerufen. Danach sind alle in der Klasse mit Initialisierern deklarierten Instanzvariablen bereits initialisiert, und es werden nur noch die restlichen Anweisungen des Konstruktorrumpfes ausgeführt.

Wir werden diese Regeln nun an unserem konkreten Beispiel anzuwenden versuchen. Hierfür werfen wir zunächst einen Blick auf die Definition unserer beiden Klassen in Java:

```
1 public class Super {
2
3     /** Eine oeffentliche Instanzvariable */
4     public String x = "vor Super-Konstruktor";
5
6     /** Ein argumentloser Konstruktor */
7     public Super() {
8         System.out.println("Super-Konstruktor gestartet.");
9         System.out.println("x = " + x);
```

```

10     x = "nach Super-Konstruktor";
11     System.out.println("Super-Konstruktor beendet.");
12     System.out.println("x = " + x);
13 }
14 }

```

Unsere Klasse `Sub` leitet sich hierbei von der Klasse `Super` ab, was wir in Java durch das Schlüsselwort **extends** zum Ausdruck bringen. Der restliche Aufbau der Klasse ergibt sich auch aus dem dazugehörigen UML-Diagramm 7.6:

```

1  public class Sub extends Super {
2
3      /** Eine weitere oeffentliche Instanzvariable */
4      public String y = "vor Sub-Konstruktor";
5
6      /** Ein argumentloser Konstruktor */
7      public Sub() {
8          System.out.println("Sub-Konstruktor gestartet.");
9          System.out.println("x = " + x);
10         System.out.println("y = " + y);
11         x = "nach Sub-Konstruktor";
12         y = "nach Sub-Konstruktor";
13         System.out.println("Sub-Konstruktor beendet.");
14         System.out.println("x = " + x);
15         System.out.println("y = " + y);
16     }
17 }

```

Wenn wir nach dem allgemeinen Muster vorgehen, unterteilt sich der Instanzierungsvorgang in verschiedene Schritte. Wir haben den Ablauf in neun Einzelschritte zerlegt, die in Abbildung 7.7 grafisch dargestellt sind:

1. Im Speicher wird Platz für ein Objekt der Klasse `Sub` reserviert. Es werden die Instanzvariablen `x` und `y` angelegt und mit den Default-Werten initialisiert.
2. Der Konstruktor wird aufgerufen. Da wir in unserem Code nicht explizit mit **super** gearbeitet haben, ruft das System automatisch den argumentlosen Konstruktor der Superklasse auf. Bei dessen Ablauf wird zunächst (automatisch) die Variable `x` initialisiert.
3. Im weiteren Ablauf des Super-Konstruktors wird eine Meldung auf dem Bildschirm ausgegeben (durch Zeile 8 und 9 im Programmcode).
4. Danach wird der Inhalt der Variable `x` auf den Wert „nach Super-Konstruktor“ gesetzt.
5. Bevor der Konstruktor der Superklasse beendet wird, gibt er eine entsprechende Meldung auf dem Bildschirm aus (Zeile 11 bis 12). Der Konstruktor der Superklasse wurde ordnungsgemäß beendet.
6. Nun wird der Konstruktor der Klasse `Sub` fortgesetzt mit der (automatischen) Initialisierung von `y`, d. h. die Variable wird auf „vor Sub-Konstruktor“ gesetzt.

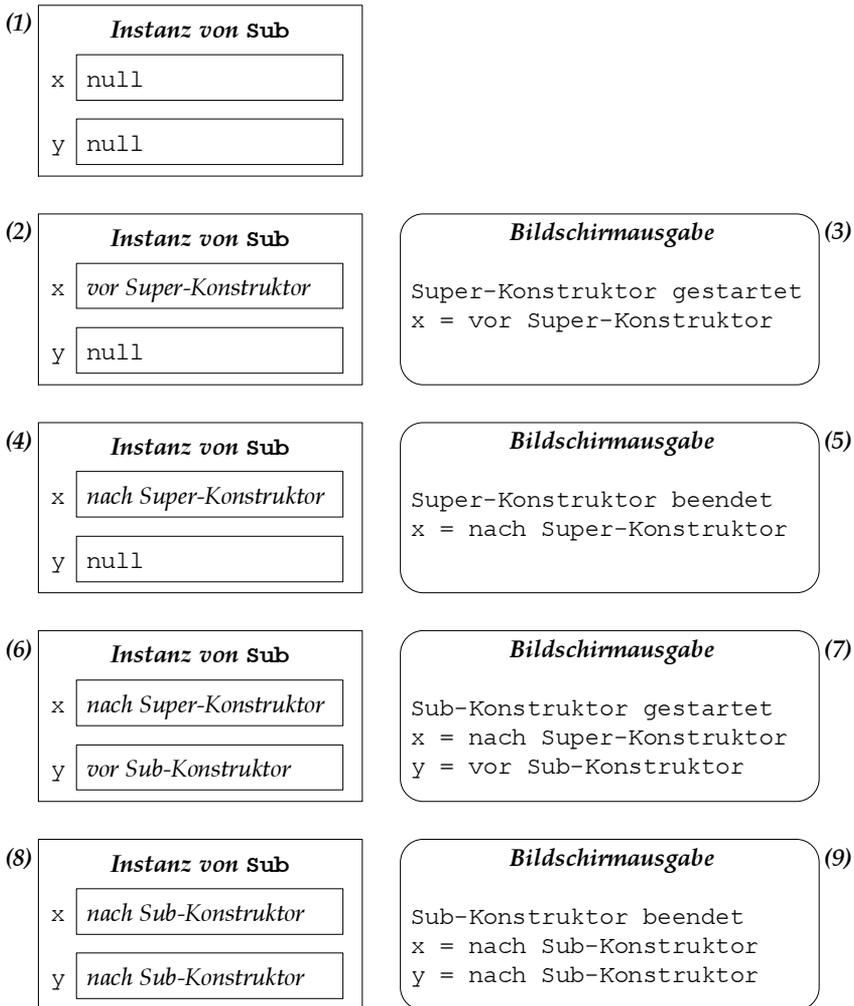


Abbildung 7.7: Instanziierungsprozess von Sub- und Superklasse

- Nun erfolgt die eigentliche Ausführung unseres Konstruktors der Klasse `Sub`. Zu Beginn des Konstruktors wird eine entsprechende Meldung ausgegeben; die Variablen `x` und `y` haben die Werte „nach Super-Konstruktor“ bzw. „vor Sub-Konstruktor“.
- Zuletzt werden die Variablen `x` und `y` wiederum auf einen neuen Wert gesetzt (Zeile 11 und 12 im Programmtext der Klasse `Sub`).
- In der anschließenden Bildschirmausgabe wird uns diese Veränderung bestätigt.

Die komplette Ausgabe unseres Programms lautet also wie folgt:

```
                          Konsole
Super-Konstruktor gestartet.
x = vor Super-Konstruktor
Super-Konstruktor beendet.
x = nach Super-Konstruktor

Sub-Konstruktor gestartet.
x = nach Super-Konstruktor
y = vor Sub-Konstruktor
Sub-Konstruktor beendet.
x = nach Sub-Konstruktor
y = nach Sub-Konstruktor
```

Wie wir sehen, haben unsere Variablen während des Instanzierungsprozesses bis zu drei verschiedene Werte angenommen. Wir können diese Zahl beliebig steigern, indem wir die Zahl der sich voneinander ableitenden Klassen erhöhen. In jeder Superklasse können wir einen Konstruktor definieren, der den Wert einer Instanzvariable verändert.

Im Allgemeinen ist es natürlich nicht sinnvoll, seine Programme auf diese Weise zu verfassen – der Quelltext wird dann unleserlich und ist schwer nachzuvollziehen. Das Wissen um den Instanzierungsprozess hilft uns jedoch weiter, um etwa die Eingangsfrage unseres Abschnitts bezüglich der Klasse `Student` beantworten zu können. Machen Sie sich anhand der Regeln klar, warum die richtige Antwort „Namenlos“ lautet.

## 7.5 Zusammenfassung

Wir haben anhand eines einfachen Anwendungsfalles – der Klasse `Student` – die grundlegenden Mechanismen kennengelernt, um in Java mit Klassen umzugehen. Wir haben Instanzvariablen und Instanzmethoden kennengelernt – Variablen und Methoden also, die direkt einem Objekt zugeordnet sind. Dieses neue Konzept stand im Gegensatz zu unserer bisherigen Vorgehensweise, Methoden als statische Komponenten einer Klasse zu erklären. Die Verwendung dieser statischen Komponenten, also Klassenvariablen und Klassenmethoden, haben wir dennoch nicht vollständig verworfen, sondern anhand eines einfachen Beispiels (der Variablen `zaehler`) ihren praktischen Nutzen in der Objektorientierung demonstriert.

Wir haben die Schlüsselworte `public` und `private` kennengelernt, mit deren Hilfe wir Teile einer Klasse öffentlich machen oder vor der Außenwelt verstecken konnten. Dabei haben wir gelernt, wie man dem Prinzip der Datenkapselung entspricht, indem wir Variablen privat deklariert und Lese- und Schreibzugriff über entsprechende (öffentliche) Methoden gewährt haben. Auf diese Weise war es uns beispielsweise möglich, Benutzereingaben wie die Matrikelnummer automatisch auf ihre Gültigkeit zu überprüfen.

Zum Schluss haben wir uns in diesem Kapitel sehr intensiv mit dem Entstehungsprozess eines Objektes beschäftigt. Wir haben gelernt, wie man mit Konstruktoren dynamische Teile eines Objektes initialisiert und wie man `static`-Blöcke einsetzt, um statische Komponenten und Konstanten mit Werten zu belegen. Ferner haben wir uns mit dem Überladen von Konstruktoren befasst und an einem konkreten Beispiel erfahren, wie das Zusammenspiel von Initialisierern und Konstruktoren in Sub- und Superklasse funktioniert.

## 7.6 Übungsaufgaben

### Aufgabe 7.1

Fügen Sie der Klasse `Student` einen weiteren Konstruktor hinzu. In diesem Konstruktor soll man in der Lage sein, alle Instanzvariablen (Name, Nummer, Fach, Geburtsjahr) als Argumente zu übergeben. Erhöhen Sie den Zähler hierbei nicht selbst, sondern verwenden Sie das Schlüsselwort `this`, um einen der bereits vorhandenen Konstruktoren aufzurufen. Übergeben Sie diesem Konstruktor auch das gewünschte Geburtsjahr.

### Aufgabe 7.2

Fügen Sie der Klasse `Student` eine weitere private Instanzvariable `geschlecht` sowie finale Klassenvariablen `WEIBLICH` und `MAENNLICH` hinzu, sodass beim Arbeiten mit Objekten der Klasse `Student` explizit zwischen weiblichen und männlichen Studierenden unterschieden werden kann. Fügen Sie der Klasse `Student` weitere Konstruktoren hinzu, die diese neuen Variablen berücksichtigen. Verwenden Sie auch hier mit Hilfe des Schlüsselworts `this` bereits vorhandene Konstruktoren.

### Aufgabe 7.3

Wir nehmen an, dass alle Karlsruher Hochschulen über ein besonderes System verfügen, um Matrikelnummern auf Korrektheit zu überprüfen:

- Zuerst wird die (als siebenstellig festgelegte) Zahl in ihre Ziffern  $Z_1, Z_2 \dots Z_7$  aufgeteilt; für die Matrikelnummer 0848600 wäre also etwa

$$Z_1 = 0, Z_2 = 8, Z_3 = 4, Z_4 = 8, Z_5 = 6, Z_6 = 0, Z_7 = 0.$$

- Nun wird eine spezielle „gewichtete Quersumme“  $\Sigma$  der Form

$$\Sigma = Z_1 \cdot 2 + Z_2 \cdot 1 + Z_3 \cdot 4 + Z_4 \cdot 3 + Z_5 \cdot 2 + Z_6 \cdot 1$$

gebildet.

- Die Matrikelnummer ist genau dann gültig, wenn die letzte Ziffer der Matrikelnummer (also  $Z_7$ ) mit der letzten Ziffer der Quersumme  $\Sigma$  übereinstimmt.

Sie sollen nun eine spezielle Klasse `KarlsruherStudent` entwickeln, die lediglich Zahlen als Matrikelnummern zulässt, die diese Prüfung bestehen. Beginnen Sie zu diesem Zweck mit folgendem Ansatz:

```

1  /** Ein Student einer Karlsruher Hochschule */
2  public class KarlsruherStudent extends Student {
3
4  }
```

Die Klasse leitet sich wegen des Schlüsselworts `extends` von unserer allgemeinen Klasse `Student` ab, erbt somit also auch alle Variablen und Methoden. Gehen Sie nun in zwei Schritten vor, um unsere Klasse zu vervollständigen:

- Im Moment haben wir bei der neuen Klasse nicht die Möglichkeit, das Geburtsjahr zu setzen (machen Sie sich klar, warum). Aus diesem Grund verfassen Sie einen Konstruktor, dem man das Geburtsjahr als Argument übergeben kann. Da Sie keinen Zugriff auf die privaten Instanzvariablen haben, müssen Sie hierzu den entsprechenden Konstruktor der Superklasse aufrufen.
- Überschreiben Sie die `validateNummer`-Methode so, dass diese die Prüfung gemäß dem Karlsruher System durchführt. Aufgrund der Polymorphie wird die neue Methode das Original in allen Karlsruher Studentenobjekten ersetzen. Da die `set`-Methode jedoch die Validierung verwendet, haben wir die Wertzuweisung automatisch dem neuen System angepasst.

*Hinweis:* Das Aufspalten einer Zahl in ihre Einzelziffern haben wir in diesem Buch schon an mehreren Stellen besprochen. Verwenden Sie bereits vorhandene Algorithmen, und sparen Sie sich somit den Aufwand einer Neuentwicklung.

#### Aufgabe 7.4

Vervollständigen Sie den nachfolgenden Lückentext mit Angaben, die sich auf die Klassen `Klang`, `Krach` und `Musik` beziehen, die am Ende dieser Aufgabe angegeben sind:

- Die Klasse ... ist Superklasse der Klasse ....
- Die Klasse ... erbt von der Klasse ... die Variable(n) ....
- In den drei Klassen gibt es die Instanzvariable(n) ....
- In den drei Klassen gibt es die Klassenvariable(n) ....
- Auf die Variable(n) ... der Klasse `Klang` kann in der Klasse `Krach` und in der Klasse `Musik` zugegriffen werden.
- Auf die Variable(n) ... der Klasse `Krach` hat keine andere Klasse Zugriff.
- Die Variable(n) ... hat/haben in allen Instanzen der Klasse `Krach` den gleichen Wert.
- Der Konstruktor der Klasse `Klang` wird in den Zeilen ... aufgerufen.

- i) Die Methode `mehrPower` der Klasse `Klang` wird in den Zeilen ... bis ... überschrieben und in den Zeilen ... bis ... überladen.
- j) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in Zeile ... und in Zeile ... aufgerufen.
- k) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in Zeile ... aufgerufen.
- l) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in ... aufgerufen.
- m) Die Methode `toString`, die in den Zeilen 7 bis 9 definiert ist, wird in ... aufgerufen.
- n) Die Methoden ... sind Instanzmethoden.

Auf die nachfolgenden Klassen sollen sich Ihre Antworten beziehen:

```
1 public class Klang {
2     public int baesse, hoehen;
3     public Klang(int b, int h) {
4         baesse = b;
5         hoehen = h;
6     }
7     public String toString () {
8         return "B:" + baesse + " H:" + hoehen;
9     }
10    public void mehrPower (int b) {
11        baesse += b;
12    }
13 }
14 public class Krach extends Klang {
15     private int rauschen, lautstaerke;
16     public static int grundRauschen = 4;
17     public Krach (int l, int b, int h) {
18         super(b,h);
19         lautstaerke = 1;
20         rauschen = grundRauschen;
21     }
22     public void mehrPower (int b) {
23         baesse += b;
24         if (baesse > 10) {
25             lautstaerke -= 1;
26         }
27     }
28     public void mehrPower (int l, int b) {
29         lautstaerke += 1;
30         this.mehrPower(b);
31     }
32 }
33 public class Musik {
34     public static void main (String[] args) {
35         Klang k = new Klang(1,5);
36         Krach r = new Krach(4,17,30);
37         System.out.println(r);
```

```
38     r.mehrPower(3);
39     r.mehrPower(2,2);
40 }
41 }
```

## Aufgabe 7.5

Gegeben seien die folgenden Java-Klassen:

```
1  class Maus {
2      Maus() {
3          System.out.println("Maus");
4      }
5  }
6
7  class Katze {
8      Katze() {
9          System.out.println("Katze");
10     }
11 }
12
13 class Ratte extends Maus {
14     Ratte() {
15         System.out.println("Ratte");
16     }
17 }
18
19 class Fuchs extends Katze {
20     Fuchs() {
21         System.out.println("Fuchs");
22     }
23 }
24
25 class Hund extends Fuchs {
26     Maus m = new Maus();
27     Ratte r = new Ratte();
28     Hund() {
29         System.out.println("Hund");
30     }
31     public static void main(String[] args) {
32         new Hund();
33     }
34 }
```

Geben Sie an, was beim Start der Klasse Hund ausgegeben wird.

## Aufgabe 7.6

Gegeben seien die folgenden Klassen:

```
1  class Eins {
2      public long f;
3      public static long g = 2;
4      public Eins (long f) {
5          this.f = f;
```

```
6     }
7     public Object clone() {
8         return new Eins(f + g);
9     }
10 }
11
12 class Zwei {
13     public Eins h;
14     public Zwei (Eins eins) {
15         h = eins;
16     }
17     public Object clone() {
18         return new Zwei(h);
19     }
20 }
21
22 public class TestZwei {
23     public static void main (String[] args) {
24         Eins e1 = new Eins(1), e2;
25         Zwei z1 = new Zwei(e1), z2;
26         System.out.print (Eins.g + "-");
27         System.out.println(z1.h.f);
28         e2 = (Eins) e1.clone();
29         z2 = (Zwei) z1.clone();
30         e1.f = 4;
31         Eins.g = 5;
32         System.out.print (e2.f + "-");
33         System.out.print (e2.g + "-");
34         System.out.print (z1.h.f + "-");
35         System.out.print (z2.h.f + "-");
36         System.out.println(z2.h.g);
37     }
38 }
```

Geben Sie an, was beim Aufruf der Klasse `TestZwei` ausgegeben wird.

### Aufgabe 7.7

Die folgenden sechs Miniaturprogramme haben alle ein und denselben Sinn. Sie definieren eine Klasse, die eine private Instanzvariable besitzt, die bei der Instanziierung gesetzt werden soll. Mit Hilfe einer `toString`-Methode kann ein derart erzeugtes Objekt (in der `main`-Methode) auf dem Bildschirm ausgegeben werden. Von diesen sechs Programmen sind zwei jedoch dermaßen verkehrt, dass sie beim Übersetzen einen Compilierungsfehler erzeugen. Drei weitere Programme beinhalten logische Fehler, die der Compiler zwar nicht erkennen kann, die aber bei Ablauf des Programms zutage treten. Finden Sie das eine funktionierende Programm, *ohne* die Programme in den Computer einzugeben. Begründen Sie bei den anderen Programmen jeweils, warum sie nicht funktionieren:

```
1 public class Fehler1 {
2
3     /** Private Instanzvariable */
4     private String name;
```

```
5
6  /** Konstruktor */
7  public Fehler1(String name) {
8      name = name;
9  }
10
11 /** String-Ausgabe */
12 public String toString() {
13     return "Name = " + name;
14 }
15
16 /** Hauptprogramm */
17 public static void main(String[] args) {
18     System.out.println(new Fehler1("Testname"));
19 }
20
21 }

1 public class Fehler2 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public Fehler2(String name) {
8         this.name = name;
9     }
10
11 /** String-Ausgabe */
12 public String toString() {
13     return "Name = " + name;
14 }
15
16 /** Hauptprogramm */
17 public static void main(String[] args) {
18     System.out.println(new Fehler2("Testname"));
19 }
20
21 }

1 public class Fehler3 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public Fehler3(String nom) {
8         name = nom;
9     }
10
11 /** String-Ausgabe */
12 public String toString() {
13     return "Name = " + name;
14 }
15
16 /** Hauptprogramm */
```

```
17     public static void main(String[] args) {
18         System.out.println(new Fehler2("Testname"));
19     }
20
21 }

1  public class Fehler4 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public Fehler4(String nom) {
8         name = nom;
9     }
10
11     /** String-Ausgabe */
12     public String toString() {
13         return "Name = " + name;
14     }
15
16     /** Hauptprogramm */
17     public static void main(String[] args) {
18         System.out.println(new Fehler4("Testname"));
19     }
20
21 }

1  public class Fehler5 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public void Fehler5(String name) {
8         this.name = name;
9     }
10
11     /** String-Ausgabe */
12     public String toString() {
13         return "Name = " + name;
14     }
15
16     /** Hauptprogramm */
17     public static void main(String[] args) {
18         System.out.println(new Fehler5("Testname"));
19     }
20
21 }

1  public class Fehler6 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public Fehler6(String nom) {
```

```

8     name = nom;
9 }
10
11 /** String-Ausgabe */
12 public String toString() {
13     return "Name = " + name;
14 }
15
16 /** Hauptprogramm */
17 public static void main(String[] args) {
18     Fehler6 variable = new Fehler6();
19     variable.name = "Testname";
20     System.out.println(variable);
21 }
22 }

```

### Aufgabe 7.8

Es sei folgende einfache Klasse gegeben, die zur Speicherung von Daten über Tennisspieler (zum Beispiel bei einem Turnier) verwendet werden könnte.

```

1 public class TennisSpieler {
2     public String name;           // Name des Spielers
3     public int alter;            // Alter in Jahren
4     public int altersDifferenz (int alter) {
5         return Math.abs(alter - this.alter);
6     }
7 }

```

- Erläutern Sie den Aufbau der Klasse grafisch.
- Was passiert durch die nachfolgenden Anweisungen?

```

TennisSpieler maier;
maier = new TennisSpieler();

```

Warum benötigt man die zweite Anweisung überhaupt?

- Erläutern Sie die Bedeutung der `this`-Referenz grafisch und anhand der Methode `altersDifferenz`.
- Wie erfolgt der Zugriff auf die Daten (Variablen) und Methoden der Klasse?
- Was versteht man unter einem Konstruktor, und wie würde ein geeigneter Konstruktor für die Klasse `TennisSpieler` aussehen? Wenn Sie die Klasse um diesen Konstruktor ergänzen, ist dann die Anweisung

```

TennisSpieler maier = new TennisSpieler();

```

noch zulässig?

- Erläutern Sie den Unterschied zwischen Instanzvariablen und Klassenvariablen.
- Erweitern Sie die Klasse `TennisSpieler` um eine Instanzvariable namens `verfolger`, die eine Referenz auf einen weiteren Tennisspieler (den unmittelbaren Verfolger in der Weltrangliste) darstellt, und um eine Instanzvariable

`startNummer`, die es ermöglicht, allen Tennisspielern (z. B. bei der Erzeugung eines neuen Objektes für eine Teilnehmerliste eines Turniers) eine (eindeutige) ganzzahlige Nummer zuzuordnen.

- h) Erweitern Sie die Klasse `TennisSpieler` um eine Klassenvariable namens `folgeNummer`, die die jeweils nächste zu vergebende Startnummer enthält.
- i) Modifizieren Sie den Konstruktor der Klasse `TennisSpieler` so, dass er jeweils eine entsprechende Startnummer vergibt und die Klassenvariable `folgeNummer` jeweils erhöht. Geben Sie auch eine Überladung dieses Konstruktors an, die es ermöglicht, bei der Objekterzeugung auch noch den Verfolger in der Weltrangliste anzugeben.
- j) Wie verändert sich der Wert der Variablen `startNummer` und `folgeNummer` in den Objekten `maier`, `schmid` und `berger` mit den nachfolgenden Anweisungen?

```
TennisSpieler maier = new TennisSpieler("H. Maier", 68);
TennisSpieler schmid = new TennisSpieler("G. Schmid", 45, maier);
TennisSpieler berger = new TennisSpieler("I. Berger", 36, schmid);
```

- k) Erläutern Sie den Unterschied zwischen Instanzmethoden und Klassenmethoden.
- l) Erweitern Sie die Klasse `TennisSpieler` um eine Instanzmethode namens `istLetzter`, die genau dann den Wert `true` liefert, wenn das `TennisSpieler`-Objekt keinen Verfolger in der Weltrangliste hat.
- m) Erweitern Sie die Klasse `TennisSpieler` um die Instanzmethode

```
public String toString () {
    String printText = name + " (" + startNummer + ")";
    if (verfolger != null)
        printText = printText + " liegt vor " + verfolger.toString();
    return printText;
}
```

die es ermöglicht, dass man Objekte der Klasse innerhalb von Zeichenkettenausdrücken (also auch in Ausgabeanweisungen) mit + verknüpfen bzw. automatisch nach String wandeln lassen kann. Was würden die Zeilen

```
System.out.println(maier);
System.out.println(schmid);
System.out.println(berger);
```

ausgeben?

- n) Wie kann man vermeiden, dass ein(e) Programmierer(in) bei der Bearbeitung der Objekte der Klasse `TennisSpieler` die (von den Konstruktoren automatisch generierten) Startnummern überschreibt? Wie lässt sich dann trotzdem lesender Zugriff auf die Startnummern ermöglichen?

### Aufgabe 7.9

Schreiben Sie eine Klasse `Mensch`, die *private* Instanzvariablen beinhaltet, um eine laufende Nummer (`int`), den Vornamen (`String`), den Nachnamen (`String`), das Alter (`int`) und das Geschlecht (`boolean`, mit `true` für männlich) eines Menschen zu speichern. Außerdem soll die Klasse eine *private* Klassenvariable namens `gesamtZahl` (zur Information über die Anzahl der bereits erzeugten Objekte der Klasse) beinhalten, die mit dem Wert 0 zu initialisieren ist.

Statten Sie die Klasse mit einem Konstruktor aus, der als Parameter das Alter als `int`-Wert, das Geschlecht als `boolean`-Wert und den Vor- und Nachnamen als `String`-Werte übergeben bekommt und die entsprechenden Instanzvariablen des Objekts mit diesen Werten belegt. Außerdem soll der Objektzähler `gesamtZahl` um 1 erhöht und danach die laufende Nummer des Objekts auf den neuen Wert von `gesamtZahl` gesetzt werden.

Statten Sie die Klasse außerdem mit folgenden Instanzmethoden aus:

- a) `public int getAlter()`  
Diese Methode soll das Alter des Objekts zurückliefern.
- b) `public void setAlter(int neuesAlter)`  
Diese Methode soll das Alter des Objekts auf den Wert `neuesAlter` setzen.
- c) `public boolean getIstMaennlich()`  
Diese Methode soll den `boolean`-Wert (die Angabe des Geschlechts) des Objekts zurückliefern.
- d) `public boolean aelterAls(Mensch m)`  
Wenn das Alter des Objekts größer ist als das Alter von `m`, soll diese Methode den Wert `true` zurückliefern, andernfalls den Wert `false`.
- e) `public String toString()` Diese Methode soll eine Zeichenkette zurückliefern, die sich aus dem Vornamen, dem Nachnamen, dem Alter, dem Geschlecht und der laufenden Nummer des Objekts zusammensetzt.

Zum Test Ihrer Klasse `Mensch` können Sie eine einfache Klasse `TestMensch` schreiben, die mit Objekten der Klasse `Mensch` arbeitet und den Konstruktor und alle Methoden der Klasse `Mensch` testet. Testen Sie dabei auch,

- ob der Compiler wirklich Zugriffe auf die privaten Instanzvariablen verweigert und
- ob der Compiler für ein Objekt `m` der Klasse `Mensch` tatsächlich bei einer Anweisung

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

automatisch die `toString()`-Methode aufruft!

**Aufgabe 7.10**

Ein Punkt  $p$  in der Ebene mit der Darstellung  $p = (x_p, y_p)$  besitzt die  $x$ -Koordinate  $x_p$  und die  $y$ -Koordinate  $y_p$ . Die Strecke  $\overline{pq}$  zwischen zwei Punkten  $p = (x_p, y_p)$  und  $q = (x_q, y_q)$  hat nach Pythagoras die Länge  $L(\overline{pq}) = \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}$  (siehe auch Abbildung 7.8).

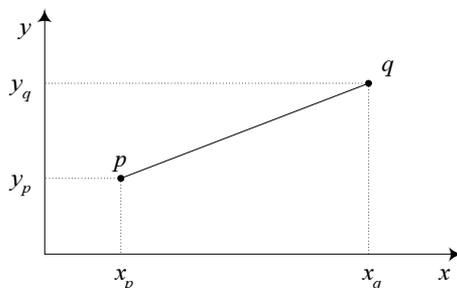


Abbildung 7.8: Definition einer Strecke

Unter Verwendung der objektorientierten Konzepte von Java soll in einem Programm mit solchen Punkten und Strecken in der Ebene gearbeitet werden. Dazu sollen

- eine Klasse `Punkt` zur Darstellung und Bearbeitung von Punkten,
- eine Klasse `Strecke` zur Darstellung und Bearbeitung von Strecken und
- eine Klasse `TestStrecke` für den Test bzw. die Anwendung dieser beiden Klassen

implementiert werden. Gehen Sie wie folgt vor:

- Implementieren Sie die Klasse `Punkt` mit zwei privaten Instanzvariablen `x` und `y` vom Typ `double`, die die  $x$ - und  $y$ -Koordinaten eines Punktes repräsentieren, und statten Sie die Klasse `Punkt` mit Konstruktoren und Instanzmethoden aus. Schreiben Sie
  - einen Konstruktor mit zwei `double`-Parametern (die  $x$ - und  $y$ -Koordinaten des Punktes),
  - eine Methode `getX()`, die die  $x$ -Koordinate des Objekts zurückliefert,
  - eine Methode `getY()`, die die  $y$ -Koordinate des Objekts zurückliefert,
  - eine `void`-Methode `read()`, die die  $x$ - und  $y$ -Koordinaten des Objekts einliest, und
  - eine `String`-Methode `toString()`, die die `String`-Darstellung des Objekts in der Form `(xStr, yStr)` zurückliefert, wobei `xStr` und `yStr` die `String`-Darstellungen der Werte von  $x$  und  $y$  sind.

b) Implementieren Sie die Klasse `Strecke` mit zwei privaten Instanzvariablen `p` und `q` vom Typ `Punkt`, die die beiden Randpunkte einer Strecke repräsentieren, und stattdessen Sie die Klasse `Strecke` mit Konstruktoren und Instanzmethoden aus. Schreiben Sie

- einen Konstruktor mit zwei `Punkt`-Parametern (die Randpunkte der Strecke),
- eine **void**-Methode `read()`, die die beiden Randpunkte `p` und `q` des Objekts einliest (verwenden Sie dazu die Instanzmethode `read` der Objekte `p` und `q`),
- eine **double**-Methode `getLaenge()`, die (unter Verwendung der Instanzmethoden `getX` und `getY` der Randpunkte) die Länge des Strecken-Objekts berechnet und zurückliefert,
- eine **String**-Methode `toString()`, die die **String**-Darstellung des Objekts in der Form `pStr_qStr` zurückliefert, wobei `pStr` und `qStr` die **String**-Darstellungen für die Instanzvariablen `p` und `q` des Objekts sind.

c) Testen Sie Ihre Implementierung mit der folgenden Klasse:

```

1 public class TestStrecke {
2     public static void main(String[] args) {
3         Punkt ursprung = new Punkt(0.0,0.0);
4         Punkt endpunkt = new Punkt(4.0,3.0);
5         Strecke s = new Strecke(ursprung,endpunkt);
6         System.out.println("Die Laenge der Strecke " + s +
7             " betraegt " + s.getLaenge() + ".");
8         System.out.println();
9         System.out.println("Strecke s eingeben:");
10        s.read();
11        System.out.println();
12        System.out.println("Die Laenge der Strecke " + s +
13            " betraegt " + s.getLaenge() + ".");
14    }
15 }

```

### Aufgabe 7.11

Gegeben sei die folgende Klasse:

```

1 public class AchJa {
2
3     public int x;
4     static int ach;
5
6     int ja (int i, int j) {
7         int y;
8         if ((i <= 0) || (j <= 0) || (i % j == 0) || (j % i == 0)) {
9             System.out.print(i+j);
10            return i + j;
11        }

```

```

12     else {
13         x = ja(i-2, j);
14         System.out.print(" + ");
15         y = ja(i, j-2);
16         return x + y;
17     }
18 }
19
20 public static void main (String[] args) {
21     int n = 5, k = 2;
22     AchJa so = new AchJa ();
23     System.out.print("ja(" + n + ", " + k + ") = ");
24     ach = so.ja(n, k);
25     System.out.println(" = " + ach);
26 }
27 }

```

- Geben Sie an, um welche Art von Variablen es sich bei den in dieser Klasse verwendeten Variablen `x` in Zeile 3, `ach` in Zeile 4, `j` in Zeile 6, `y` in Zeile 7, `n` in Zeile 21 und `so` in Zeile 22 jeweils handelt. Verwenden Sie (sofern diese zutreffen) die Bezeichnungen Klassenvariable, Instanzvariable, lokale Variable und formale Variable (bzw. formaler Parameter).
- Geben Sie an, was das Programm ausgibt.
- Angenommen, die Zeile 24 würde in der Form

```
ach = ja(n, k);
```

gegeben sein. Würde der Compiler das Programm trotzdem übersetzen? Wenn nein, warum nicht?

### Aufgabe 7.12

Es sei folgende einfache Klasse gegeben, die zur Speicherung von Daten über Patienten in der Aufnahme einer Arztpraxis verwendet werden könnte.

```

1 public class Patient {
2     public String name;           // Name des Patienten
3     public int alter;            // Alter (in Jahren)
4     public int altersDifferenz (int alter) {
5         return Math.abs(alter - this.alter);
6     }
7 }

```

- Erläutern Sie den Aufbau der Klasse grafisch.
- Was passiert durch die nachfolgenden Anweisungen?

```
Patient maier;
maier = new Patient();
```

- Wie würde ein geeigneter Konstruktor für die Klasse `Patient` aussehen? Wenn Sie die Klasse um diesen Konstruktor ergänzen, ist dann die Anweisung

```
Patient maier = new Patient();
```

noch zulässig?

- d) Erweitern Sie die Klasse `Patient` um eine Instanzvariable `vorherDran`, die eine Referenz auf einen weiteren Patienten darstellt, und um eine Instanzvariable `nummer`, die es ermöglicht, allen Patienten (z. B. bei der Erzeugung eines neuen Objektes für eine Warteliste einer Praxis) eine (eindeutige) ganzzahlige Nummer zuzuordnen.
- e) Erweitern Sie die Klasse `Patient` um eine Klassenvariable `folgeNummer`, die die jeweils nächste zu vergebende Nummer enthält.
- f) Modifizieren Sie den Konstruktor der Klasse `Patient` so, dass er jeweils eine entsprechende Nummer vergibt und die Klassenvariable `folgeNummer` jeweils erhöht. Geben Sie auch eine Überladung dieses Konstruktors an, die es ermöglicht, auch noch den Vorgänger in der Warteliste anzugeben.
- g) Wie verändert sich der Wert der Variablen `nummer` und `folgeNummer` in den Objekten `maier`, `schmid` und `berger` mit den nachfolgenden Anweisungen?

```
Patient maier = new Patient("H. Maier", 68);
Patient schmid = new Patient("G. Schmid", 45, maier);
Patient berger = new Patient("I. Berger", 36, schmid);
```

- h) Erweitern Sie die Klasse `Patient` um eine Instanzmethode `istErster`, die genau dann den Wert `true` liefert, wenn das Patienten-Objekt keinen Vorgänger in der Warteliste hat.
- i) Erweitern Sie die Klasse `Patient` um die Instanzmethode

```
public String toString () {
    String printText = name + " (" + nummer + ")";
    if (vorherDran != null)
        printText = printText + " kommt nach " + vorherDran.toString();
    return printText;
}
```

die es ermöglicht, dass man Objekte der Klasse innerhalb von Zeichenkettenausdrücken (also auch in Ausgabeanweisungen) mit `+` verknüpfen kann. Was würden die Zeilen

```
System.out.println(maier);
System.out.println(schmid);
System.out.println(berger);
```

ausgeben?

- j) Wie vermeidet man, dass ein(e) Programmierer(in) bei der Bearbeitung der Objekte der Klasse `Patient` die (von den Konstruktoren automatisch generierten) Nummern überschreibt? Wie ermöglicht man dann trotzdem lesenden Zugriff auf die Identifikationsnummern?

### Aufgabe 7.13

Sie sollen verschiedene Fahrzeuge mittels objektorientierter Programmierung simulieren. Dazu ist Ihnen folgende Klasse vorgegeben:

```
1 public class Reifen {
2
3     /** Reifendruck */
4     private double druck;
5
6     /** Konstruktor */
7     public Reifen (double luftdruck) {
8         druck = luftdruck;
9     }
10
11     /** Zugriffsfunktion fuer Reifendruck */
12     public double aktuellerDruck () {
13         return druck;
14     }
15 }
```

Schreiben Sie eine Klasse `Fahrzeug`, die die Klasse `Reifen` verwendet und Folgendes beinhaltet:

a) **private** Instanzvariablen

- `name` vom Typ `String` (für die Bezeichnung des Fahrzeugs),
- `anzahlReifen` vom Typ `int` (für die Anzahl der Reifen des Fahrzeugs),
- `reifenArt` vom Typ `Reifen` (für die Angabe des Reifentyps des Fahrzeugs) und
- `faehrt` vom Typ `boolean` (für die Information über den Fahrzustand des Fahrzeugs);

- b) einen Konstruktor, der mit Parametern für Bezeichnung, Reifenanzahl und Reifendruck ausgestattet ist, in seinem Rumpf die entsprechenden Komponenten des Objekts belegt und außerdem das Fahrzeug in den Zustand „fährt nicht“ versetzt;
- c) eine öffentliche Instanzmethode `fahreLos()`, die die Variable `faehrt` des Fahrzeug-Objektes auf `true` setzt;
- d) eine öffentliche Instanzmethode `halteAn()`, die die Variable `faehrt` des Fahrzeug-Objektes auf `false` setzt;
- e) eine öffentliche Instanzmethode `status()`, die einen Informations-String über Bezeichnung, Fahrzustand, Reifenzahl und Reifendruck des Fahrzeug-Objektes auf den Bildschirm ausgibt.

### Aufgabe 7.14

Schreiben Sie ein Testprogramm, das in seiner `main`-Methode zunächst ein Fahrrad (verwenden Sie Reifen mit 4.5 bar) und ein Auto (verwenden Sie Reifen mit 1.9 bar) in Form von Objekten der Klasse `Fahrzeug` erzeugt und anschließend folgende Vorgänge durchführt:

1. mit dem Fahrrad losfahren,
2. mit dem Auto losfahren,
3. mit dem Fahrrad anhalten,
4. mit dem Auto anhalten.

Unmittelbar nach jedem der vier Vorgänge soll jeweils mittels der Methode `status()` der aktuelle Fahrzustand *beider* Fahrzeuge ausgegeben werden. Eine Ausgabe des Testprogramms sollte also etwa so aussehen:

```

————— Konsole —————
Zustand 1:
Fahrrad1 faehrt auf 2 Reifen mit je 4.5 bar
Auto1 steht auf 4 Reifen mit je 1.9 bar
Zustand 2:
Fahrrad1 faehrt auf 2 Reifen mit je 4.5 bar
Auto1 faehrt auf 4 Reifen mit je 1.9 bar
Zustand 3:
Fahrrad1 steht auf 2 Reifen mit je 4.5 bar
Auto1 faehrt auf 4 Reifen mit je 1.9 bar
Zustand 4:
Fahrrad1 steht auf 2 Reifen mit je 4.5 bar
Auto1 steht auf 4 Reifen mit je 1.9 bar
```

### Aufgabe 7.15

Gegeben seien die folgenden Klassen:

```

1 public class IntKlasse {
2     public int a;
3     public IntKlasse (int a) {
4         this.a = a;
5     }
6 }
7 public class RefIntKlasse {
8     public IntKlasse x;
9     public double y;
10    public RefIntKlasse(int u, int v) {
11        x = new IntKlasse(u);
12        y = v;
13    }
14 }
```

```

15 public class KlassenTest {
16     public static void copy1 (RefIntKlasse f, RefIntKlasse g) {
17         g.x.a = f.x.a;
18         g.y    = f.y;
19     }
20     public static void copy2 (RefIntKlasse f, RefIntKlasse g) {
21         g.x = f.x;
22         g.y = f.y;
23     }
24     public static void copy3 (RefIntKlasse f, RefIntKlasse g) {
25         g = f;
26     }
27     public static void main (String args[]) {
28         RefIntKlasse p = new RefIntKlasse(5,7);
29         RefIntKlasse q = new RefIntKlasse(1,2); // Ergibt das Ausgangsbild
30         // HIER FOLGT NUN EINE KOPIERAKTION:
31         ... /***
32     }
33 }

```

Das Ausgangsbild (mit Referenzen und Werten), das sich zur Laufzeit unmittelbar vor der Kopieraktion ergibt, sieht wie in Abbildung 7.9 beschrieben aus.

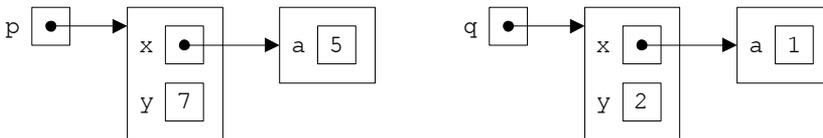


Abbildung 7.9: Ausgangsbild Aufgabe 7.15

- a) Welches Bild würde sich ergeben, wenn unmittelbar vor `/***`

```
copy1(p, q);
```

stehen würde? Zeichnen Sie den Zustand inklusive der Referenzen und Werte nach der Kopieraktion.

- b) Welches Bild würde sich ergeben, wenn unmittelbar vor `/***`

```
copy2(p, q);
```

stehen würde? Zeichnen Sie die Referenzen und Werte nach der Kopieraktion.

- c) Welches Bild würde sich ergeben, wenn unmittelbar vor `/***`

```
copy3(p, q);
```

stehen würde? Zeichnen Sie die Referenzen und Werte nach der Kopieraktion.

- d) Welches Bild würde sich ergeben, wenn unmittelbar vor `/***`

```
q = p;
```

stehen würde? Zeichnen Sie die Referenzen und Werte nach der Kopieraktion.

### Aufgabe 7.16

Gegeben sei folgende Klasse zur Darstellung und Bearbeitung von runden Glasböden:

```

1 public class Glasboden {
2     private double radius;
3     public Glasboden (double r) {
4         radius = r;
5     }
6     public void verkleinern (double x) {
7         // verkleinert den Radius des Glasboden-Objekts um x
8         radius = radius - x;
9     }
10    public double flaeche () {
11        // liefert die Flaeche des Glasboden-Objekts
12        return Math.PI * radius * radius;
13    }
14    public double umfang () {
15        // liefert den Umfang des Glasboden-Objekts
16        return 2 * Math.PI * radius;
17    }
18    public String toString() {
19        // liefert die String-Darstellung des Glasboden-Objekts
20        return "B(r=" + radius + ")";
21    }
22 }

```

- a) Ergänzen Sie die fehlenden Teile der Klasse `TrinkGlas`, die ein Trinkglas durch jeweils einen Glasboden und durch eine Füllstandsangabe darstellt:
- Ergänzen Sie zwei private Instanzvariablen `boden` vom Typ `Glasboden` und `fuellStand` vom Typ `double` (der Boden und der Füllstand des Glases).
  - Vervollständigen Sie den Konstruktor.
  - Vervollständigen Sie die Methode `verkleinern`, die die Größe des `TrinkGlas`-Objekts verändert, indem der Glasboden um den Wert `x` verkleinert und der Füllstand des Glases um den Wert `x` verringert wird.
  - Vervollständigen Sie die Methode `flaeche()`, die die Innenfläche (siehe Hinweis) des `TrinkGlas`-Objekts berechnet und zurückliefert.
  - Vervollständigen Sie die Methode `fuellMenge()`, die die Füllmenge (siehe Hinweis) des `TrinkGlas`-Objekts berechnet und zurückliefert.
  - Vervollständigen Sie die Methode `toString()`, die die String-Darstellung des Objekts in der Form `G(xyz, s=uvw)` zurückliefert, wobei `xyz` für die String-Darstellung der Instanzvariable `boden` und `uvw` für den Wert des Füllstandes des Trinkglases stehen sollen.

**Hinweis:** Bezeichnen  $F$  die Glasbodenfläche,  $U$  den Glasbodenumfang und  $s$  den Füllstand eines Trinkglases, so sollen die Innenfläche  $I$  und die Füllmen-

ge  $M$  dieses Trinkglases durch

$$I = F + U \cdot s \quad \text{und} \quad M = F \cdot s$$

berechnet werden.

```
public class TrinkGlas {
    .
    .
    .
    .
    public TrinkGlas (double fuellStand, Glasboden boden) {
        .
        .
        .
    }
    public void verkleinern (double x) {
        .
        .
        .
    }
    public double flaeche() {
        .
        .
        .
    }
    public double fuellMenge() {
        .
        .
        .
    }
    public String toString() {
        .
        .
        .
    }
}
```

- b) Ergänzen Sie die nachfolgende Klasse `TesteTrinkGlas`. In deren `main`-Methode soll zunächst ein `Trinkglas g` aus einem `Glasboden b` mit Radius 100 und Füllstand 50 konstruiert werden. Danach soll in einer Schleife das `Trinkglas` jeweils um den Wert 5 verkleinert und das aktuelle `Trinkglas`, seine bedeckte Innenfläche und seine Füllmenge ausgegeben werden.

Die Schleife soll nur durchlaufen werden, falls bzw. solange für die Innenfläche  $I$  und die Füllmenge  $M$  des Trinkglases gilt

$$I < \frac{M}{8}.$$



# Stichwortverzeichnis

| 75, 78, 339  
|| 78  
|= 76  
\* 72  
\*/ 44  
\*= 76  
+ 49  
++ 49  
+= 76  
- 49  
-- 49  
-= 76  
-> 641  
/ 72  
/\* 44  
/\*\* 44  
// 43  
/= 76  
:: 650  
< 76  
<< 75  
<= 76  
<> 359, 425  
== 49, 76  
> 76  
>= 76  
>> 75  
? : 49, 78  
% 72  
%= 76  
& 75, 78  
&= 76  
&& 78  
^ 75  
~ 75

Ablaufsteuerung 88  
abs 386, 392  
Abschlussoperationen 658, 661, 662  
abstract 258  
Abstract Window Toolkit 431, 441  
AbstractButton 461  
abstrakte Klassen 258, 273, 274, 277  
Absturz 721  
accept 619, 645  
ActionEvent 508, 509  
ActionListener 497, 511  
actionPerformed 497, 511  
Adapter-Klassen 514  
add 385, 390, 402, 412, 419, 441, 446, 447, 466, 487  
addActionListener 498  
addAll 412  
addItem 468  
addSeparator 487  
Adresse 67, 112  
after 400, 402  
Aggregation 206  
aktueller Parameter 162  
algorithmische Beschreibung 29  
Algorithmus 26, 721  
Alias-Namen 616  
allMatch 661  
ALT\_MASK 489  
ancestorAdded 512  
AncestorEvent 508  
AncestorListener 512  
ancestorMoved 513  
ancestorRemoved 513  
anonyme Klasse 296, 503  
ant 715

- Anteil, ganzzahliger 61
- Anweisungen 34, 88
  - Ausdrucks- 88
  - `break` 99
  - `case` 91
  - `continue` 100
  - `default` 91
  - `do` 97
  - Entscheidungs- 89
  - `for` 95
  - `if` 89
  - `import` 49
  - leere 88
  - markierte 99
  - `package` 288
  - `return` 101, 160
  - `switch` 91
  - `while` 97
  - Wiederholungs- 95
- Anwendungsfälle 206
- Anwendungsschicht 614
- Anwendungssoftware 26
  - `anyMatch` 661
- API 721
- API-Dokumentation 706
- API-Spezifikation 183
- `APPEND` 608
- `append` 372
- Applet 721
- `Application` 682
- Applikation 52, 722
- Applikationsserver 693
  - `apply` 645, 646
- Arbeitsspeicher 26, 112, 722
  - `args` 159, 176
- Argument, formales 159
- Argumentliste 159
  - `ArithmeticException` 312
- Arithmetische Operatoren 72
  - `arraycopy` 125, 167
  - `ArrayList` 420
  - `Arrays` 424
  - `asList` 424
- Assemblersprache 28
- `assert` 336
- `AssertionError` 336
- Assertions 336
- `ATOMIC_MOVE` 608
- Aufruf von Methoden 161
- Aufzählungstypen 341
- Ausdrücke 34, 70, 81
  - konstante 73
  - Lambda- 637, 641
- Ausdrucksanweisung 88
- Ausgabe 36, 53, 394, 395, 581, 585, 596, 604
  - formatierte 398
- Ausgabebezeichnung 36
- Ausgabegeräte 26
- Ausnahme 312, 338
- Aussage, logische 64
- Auswahllisten 467, 470
- Auswertungsreihenfolge 79
- Autoboxing 380
  - `AutoCloseable` 340, 604
- automatische Typkonvertierung 64, 163, 262
- automatische Typumwandlung 64, 163, 262
- Autounboxing 380
  - `average` 662
- AWT 431, 441
- `AWTEvent` 508
- Basis-Container 435, 437
- Baukastenprinzip 433
- Beautifier 91
- bedingtes logisches Oder 77
- bedingtes logisches Und 77
- Beenden 436
  - `before` 400, 402
- Behälter 433
- Benutzungsschnittstelle, grafische 433
- Berechnungen 70
- Betriebssystem 722
- Bezeichner 45
  - `BiConsumer` 645
  - `BiFunction` 645
  - `BigDecimal` 390
  - `BigInteger` 384
- Bildlaufleisten 478
- Bildschirmausgabe 36
- binäre Operatoren 49, 70
- binäre Zahlen 59, 722
- Binärfolge 57
  - `BinaryOperator` 646
  - `binarySearch` 422, 424
- Binden

- dynamisches 268
- BiPredicate 645
- Bit 74, 733
- Bitoperatoren 74
- BLACK 449
- Block 37, 49, 53, 89
  - Anfang 53
  - Ende 53
  - static- 229
  - Struktur 89
- BLUE 449
- BOLD 451
- Boolean 376
- boolean 64
- booleanValue 378
- BorderLayout 455
- BOTTOM 458
- Bounded Wildcards 354
- Boxing 380
- break 91, 99
- Browser 723
- Buchstaben 45
- Buffer 605
- BufferedInputStream 599
- BufferedOutputStream 599
- BufferedReader 589, 659
- BufferedWriter 589
- Bug 723
- Bugfix 723
- ButtonGroup 465
- Byte 74, 733
- Byte 376
- byte 58
- Byte Streams 582
- Byte-Ströme 582
- Bytecode 28, 30, 723
- byteValue 378
  
- Calendar 401
- Call by reference 166
- Call by value 162
- canRead 583
- canWrite 583
- CaretEvent 509
- CaretListener 511
- caretUpdate 511
- case 91
- Cast 64
  
- catch 314, 315
  - mehrfaches 339
- CENTER 453, 455, 458, 474
- ChangeEvent 508, 509
- ChangeListener 511
- Channel 605
- char 63
- Character 376
- Character Streams 582
- charValue 378
- CheckedInputStream 605
- CheckedOutputStream 605
- clear 412
- clearSelection 470
- Client 618, 723
- Client-Host 618
- Client-Rechner 618
- Client/Server-Programmierung 613
- clone 126
- close 585, 597, 604, 620
- CMD 30
- Code Formatter 91
- Codierung 29
- Collection 410, 411, 659
- Collections 422
- Color 449
- Comparable 366, 416
- Comparator 648
- compare 648
- compareTo 377, 386, 392, 400, 416
- Compiler 28, 30, 723
- Component 445
  - componentAdded 512
  - ComponentEvent 508
  - componentHidden 512
  - ComponentListener 512
  - componentMoved 512
  - componentRemoved 512
  - componentResized 512
  - componentShown 512
- Computer 25, 723
- Computersystem 26
- Consumer 645
- Container 433, 434, 480, 482, 483
- Container 440, 446
- ContainerEvent 508
- ContainerListener 512
- contains 412

- `containsAll` 412
- Content-Pane 440
- `continue` 100
- `copy` 473, 474, 607
- `COPY_ATTRIBUTES` 608
- `count` 661
- `countTokens` 427
- `CREATE` 608
- `createNewFile` 583
- `CTRL_MASK` 489
- `currentThread` 549
- `cut` 473, 474
- `CYAN` 449
  
- Dämon-Threads 558
- `DARK_GRAY` 449
- Data Binding 689
- data hiding 203, 213
- `DataInputStream` 597, 598
- `DataOutputStream` 597, 598
- `Date` 399
- Date/Time-API 670, 690
- `DateFormat` 405
- Datei 26, 582, 724
  - Namen 38
  - Namen-Erweiterung 38
  - namenerweiterung 26
- Datenbank 724
- Datenkapselung 143, 203
- Datentypen 36, 57, 724
  - einfache 57
  - elementar 111
  - ganzzahlige 57
  - generische 346
  - Gleitkomma- 61
  - komplex 111, 144
  - Referenz- 111, 144
- Datumsangaben 399, 401, 405
- `DAY_OF_MONTH` 402
- `DAY_OF_YEAR` 402
- Deadlock 558, 724
- `DecimalFormat` 395
- `default` 91, 284
- Default-Konstruktor 226
- Default-Methoden 283
- Default-Werte 232
- `DeflaterOutputStream` 605
- Deklaration 68
  - von Methoden 159
  - von Variablen 36
- Dekrementoperator 79
- Delegation Event Model 496
- `delete` 374, 583
- `DELETE_ON_CLOSE` 608
- `deleteCharAt` 374
- `deleteIfExists` 607
- Deserialisierung 599
- Diamond-Operator 359, 425
- Dienst 618
- directory 26
- `dispose` 482–484
- `DISPOSE_ON_CLOSE` 482
- `distinct` 661
- `divide` 385, 390
- Division 61
- DNS 616, 724
- `do` 97
- `DO_NOTHING_ON_CLOSE` 482
- Domain Name Service 616
- Domain-Namen 616
- Doppelklicken 724
- `Double` 376
- `double` 62
- `DoubleBuffer` 605
- `doubleValue` 378
- Download 725
- `drawArc` 532
- `drawLine` 531
- `drawOval` 531
- `drawPolygon` 531
- `drawPolyline` 531
- `drawRect` 531
- `drawString` 532
- dreistellige Operatoren 49, 70
- `Duration` 691
- dyadische Operatoren 49, 70
- dynamisches Binden 268
  
- E/A-Ströme 581
- `EAST` 455
- Editor 29, 725
- effektiv final 653
- einfache Datentypen 57
- Eingabe 581, 585, 596, 604
- Eingabeaufforderung 30
- Eingabegeräte 26

- Eingabestrom 717
- einstellige Operatoren 49, 70
- EJB-Container 694
- else** 89
- Elternklasse 259
- Enterprise Java Beans 694
- Entscheidungsanweisung 89
- Entwicklungsumgebung 725
  - integrierte 31
- Entwurfsmuster 207
- equals** 271, 367, 377, 386, 392, 400, 416
- equals-Vertrag** 271
- erben 200
- Ereignis 434, 495
- Ereignisempfänger 496
- Ereignisquellen 496
- Ereignisverarbeitung 495, 503
- Ergebnisrückgabe 160
- Ergebnistyp 72, 159
- Error** 333
- Error message 312
- erweitern 201
- Erweiterung 199
  - Dateinamen- 26, 38
- Erzeuger/Verbraucher-Problem 560
- Escape-Sequenzen 63
- Ethernet 614
- Event 495
- Event-Dispatching-Thread 577, 683
- EventListener** 510
- EventObject** 507
- Exception** 338
- Exception** 312, 318
- exists** 583
- exit** 559
- EXIT\_ON\_CLOSE** 437, 482
- exklusives Oder 74
- explizite Typkonvertierung 65
- Exponentenschreibweise 35, 62
- extends** 202, 259
- externer Speicher 26, 725
  
- false** 64
- Farben 434, 449
- FDDI 614
- Fehler 30
  - fachlicher 712
  - Regression 712
  - semantischer 712
  - syntaktischer 712
- Fehlermeldung 48, 59, 312
- Felder 113, 115
  - flache Kopie 126, 134, 153
  - Index 115
  - Initialisierer 119
  - Komponenten 115
  - Kopie 167
  - Länge 119
  - mehrdimensionale 128, 132
  - Referenzkopie 125, 134, 153
  - Tiefenkopie 126, 134, 153, 154
  - von Feldern 129
  - Zeile 129
- Feldinitialisierer 119
- Feldkomponenten 115
- Feldlänge 119
- Fenster 435, 437
- Fiber Distributed Data Interface 614
- File** 582
- file** 26
- File Transfer Protocol 614
- FileChannel** 605
- FileInputStream** 597
- FileOutputStream** 597
- FileReader** 316, 586
- Files** 607, 659
- FileWriter** 586
- fillArc** 532
- fillOval** 532
- fillPolygon** 532
- fillRect** 532
- filter** 661
- final** 69, 222, 263, 269
- final, effektiv** 653
- finally** 331
- find** 659
- first** 417
- flache Kopie 126, 134, 153
- Float** 376
- float** 62
- FloatBuffer** 605
- Floating point numbers 35
- floatValue** 378
- FlowLayout** 453
- flush** 585, 597
- FocusEvent** 508

- focusGained 512
- FocusListener 512
- focusLost 512
- Fokus 461
- folder 26
- Font 451
- Fonts 434, 451
- for 95
- forEach 647, 661
- formale Argumente 159
- formale Parameter 159
- Format 405
- Format 395
- format 396, 406
- formatierte Ausgabe 395, 398, 405
- Formeln 34
- Frame 435
- Freeware 91, 725
- FTP 614
- FULL 408
- Function 645
- Funktion 157
- funktionale Interfaces 643
  
- ganze Zahlen 35
- Ganzzahlen, lange 383
- ganzzahliger Anteil 61
- Garbage Collector 725
- gcd 386
- Generalisierung 198
- generate 659, 660
- generische Datentypen 346, 411
- generische Klassen 349
- generische Methoden 356
- gepufferte Ströme 589
- get 402, 419, 606, 646
- get-Methoden 216
- getActionCommand 510
- getBackground 445
- getByName 616
- getClickCount 510
- getComponents 447
- getContentPane 482–484
- getDateInstance 408
- getDateTimeInstance 408
- getFirstIndex 510
- getFont 445
- getForeground 445
- getHeight 445, 531
- getHostAddress 616
- getHostName 616
- getIcon 458
- getInputStream 620
- getInsets 531
- getInstance 401
- getItem 487, 510
- getItemAt 468
- getItemCount 468, 487
- getKeyStroke 488
- getLastIndex 510
- getLineCount 476
- getLineWrap 476
- getMaxSelectionIndex 470
- getMenu 487
- getMenuCount 487
- getMinSelectionIndex 470
- getName 549, 583
- getOutputStream 620
- getPriority 549, 558
- getSelectedIndex 468
- getSelectedIndices 471
- getSelectedItem 468
- getSelectedText 473
- getSelectedValuesList 471
- getSelectionMode 471
- getSource 507
- getStateChange 510
- getText 458, 473
- getThreadGroup 549
- getTime 400, 402
- getTimeInstance 408
- getToolTipText 447
- getWidth 445, 531
- getWindow 510
- getWrapStyleWord 476
- getX 510
- getY 510
- Gibibyte 733
- Gigabyte 733
- GirdPane 686
- Gleitkommadatentypen 61
- Gleitkommazahlen 35, 61, 61
  - lange 387
- gradle 715
- Grafikkoordinaten 530
- grafische Darstellung 529

- grafische Oberfläche [431, 433](#)
- Grammatik [725](#)
- Graphical User Interface [433](#)
- Graphics [531](#)
- GRAY [449](#)
- GREEN [449](#)
- GregorianCalendar [401](#)
- Gregorianischer Kalender [726](#)
- GridLayout [456](#)
- Gruppen, Thread- [559](#)
- GUI [433, 726](#)
- Gültigkeitsbereich [89](#)
- GZIPInputStream [605](#)
- GZIPOutputStream [605](#)
  
- Hacker [726](#)
- Hardware [26](#)
- Hashcode [272](#)
- hashCode [272, 367](#)
- HashSet [415](#)
- Hashtabellen [272](#)
- hasMoreElements [427](#)
- hasMoreTokens [427](#)
- hasNext [413](#)
- Hauptmethode [38, 53, 176](#)
- Hauptprogramm [176](#)
- Hauptroutine [159](#)
- HBox [686](#)
- headSet [417](#)
- heavyweight [441](#)
- hexadezimale Zahlen [59, 726](#)
- HIDE\_ON\_CLOSE [482](#)
- Hilfsklassen [369](#)
- höhere Programmiersprache [28](#)
- HORIZONTAL [489](#)
- HORIZONTAL\_SCROLLBAR\_ALWAYS [478](#)
- HORIZONTAL\_SCROLLBAR\_AS\_NEEDED [478](#)
- HORIZONTAL\_SCROLLBAR\_NEVER [478](#)
- Host [727](#)
- Host-Namen [616](#)
- HOURLY\_OF\_DAY [402](#)
- HTML [727](#)
- HTTP [614, 727](#)
- Hüll-Klassen [273, 375](#)
  - Autoboxing [380](#)
  - Autounboxing [380](#)
  - Boxing [380](#)
  - Unboxing [380](#)
- Hypertext Transfer Protocol [614](#)
  
- I/O-Stream [581](#)
- ICANN [616](#)
- Icon [459](#)
- IDE [31, 708, 727](#)
- if [89](#)
- Ikonisieren [436](#)
- ImageIcon [459](#)
- imperative Programmierung [196, 727](#)
- implements [275](#)
- implizite Typkonvertierung [64, 163, 262](#)
- implizite Typumwandlung [64, 163, 262](#)
- import [49](#)
- Index [115](#)
- indexOf [419](#)
- indizierte Variablen [115](#)
- InetAddress [616](#)
- Infix [71](#)
- InflaterInputStream [605](#)
- Informatik [27](#)
- Information Hiding PrincipleKEY [143](#)
- Initialisierer, statische [229](#)
- Initialisierung [69, 225](#)
- Inkrementoperator [79](#)
- innere Klasse [142, 290, 497, 503](#)
- input stream [717](#)
- InputStream [582, 596](#)
- InputStreamReader [586](#)
- insert [373](#)
- Insets [531](#)
- instanceof [271](#)
- Instant [690](#)
- Instanz [144](#)
- Instanziierung [144, 225](#)
- Instanzmethode [184, 203, 213, 214](#)
- Instanzvariablen [142](#)
- int [58](#)
- IntBuffer [605](#)
- Integer [376](#)
- integer [35](#)
- Integrierte Entwicklungsumgebung [31, 708](#)
- Interfaces [273, 275](#)
  - funktionale [643](#)
- Internet [27](#)
- Internetprotokoll [614](#)
- Interpreter [28, 30, 727](#)

- Interpunktionszeichen 48
- interrupt 549, 554
- interrupted 550
- Intranet 27
- InputStream 659
- intValue 378
- invalidate 537
- invokeAndWait 577
- invokeLater 577
- IOTools 595
- IP 614, 615, 727
- IP-Adresse 616, 728
- isAlive 549, 557
- isDaemon 549, 559
- isDirectory 583
- isEditable 468, 473
- isEmpty 412
- isEnabled 445
- isFile 583
- isFocusPainted 461
- isInterrupted 549, 554
- isModal 484
- isOpaque 447
- isSelected 461
- isSelectedIndex 471
- isSelectionEmpty 471
- isVisible 446
- IT 728
- ITALIC 451
- ItemEvent 508, 509
- ItemListener 511
- itemStateChanged 511
- Iterable 413
- iterate 659, 660
- Iterator 413
- iterator 412, 414
  
- Jakarta EE 693
- Jakarta EE 670
- JAR-Datei 728
- Jakarta Enterprise Edition 670
- Java
  - Bytecode 28, 30
  - Compiler 30
  - Development Kit 30
  - Interpreter 28, 30
- Java EE 693
- Java Enterprise Edition 670, 693
  
- Java Foundation Classes 434
- java.awt 434, 448
- java.awt.event 488, 489, 507
- java.io 316, 582, 591, 604
- java.lang 182, 287, 369, 372
- java.math 388
- java.net 613, 616, 619, 631
- java.nio 605
- java.nio.file 606
- java.text 395, 405
- java.util 399, 408, 410, 422, 424, 426, 507
- java.util.functions 645
- java.util.stream 654
- javadoc 44, 706
- JavaFX 431, 670, 681
  - Application-Thread 683
  - Container 686
  - Data Binding 689
  - Scene Builder 689
- JavaScript 671
- javax.swing 434, 447, 488
- javax.swing.event 507
- javax.swing.text 473
- Java EE 670
- JButton 461
- JCheckBox 464
- JComboBox 467
- JComponent 447
- JDialog 483
- JDK 30, 728
- JEE 670
- JFC 434
- JFrame 437, 482
- Jigsaw 670
- JLabel 439, 459
- JList 470
- JMenuBar 486
- join 566
- JPanel 480
- JPasswordField 473
- JRadioButton 465
- JScrollPane 478
- JShell 670, 671
- JTextArea 476
- JTextComponent 473
- JTextField 473
- JToggleButton 463
- JToolBar 489

JWindow 483

Kapselung 143, 203

KeyEvent 508

KeyListener 512

keyPressed 512

keyReleased 512

keyTyped 512

Kibibyte 74, 733

Kilobyte 74, 733

Kindklassen 259

Klapptafeln 467

Klassen 38, 38, 52, 140, 141, 195

abstrakte 258, 273, 274, 277

Adapter- 514

anonyme 296, 503

Attribute 142

Diagramm 143, 206

Eltern- 259

Exception- 312

generische 349

Hüll- 273, 375

innere 142, 290, 503

Instanz 144

Instanziierung 144

Instanzvariablen 142

Kapselung 143

Kind- 259

Komponentenvariablen 142

Methoden 142, 181, 220

Namen 38

Referenz 147

Sub- 199, 257, 259

Super- 199, 259

Variablen 142, 220

Wrapper- 273, 375

Klassendiagramm 143, 206

Klassenkonstanten 222

Klassenmethoden 181, 220

Klassenvariablen 142, 220

Klicken 728

Knöpfe 461, 463–465

Kommandozeile 30

Kommandozeilenparameter 176

Kommentare 43, 48

mit javadoc 44

Kommunikation, Thread- 559

Kompatibilität 728

Komponenten 433, 434

grafische Darstellung 529

statische 220

Komponentenvariablen 142

Komposition 205

Konkatenation, String- 370

Konsole 53

Konsolenfenster 30, 36, 729

Konstanten

Klassen- 222

Literal- 46, 59, 60

symbolische 69, 222

Zeichenketten- 369

konstanter Ausdruck 73

Konstruktoren 225

Default- 226

Standard- 226

Überladen 227, 228

Koordinatensystem 530

Kopie

Feld- 167

flache 126, 134, 153

mit clone 126

Referenz- 166

Tiefen- 126, 134, 153, 154

Labels 439, 459

Lambda-Ausdrücke 637, 641

Länge eines Feldes 119

Langzahlen 383, 387

last 417

lastIndexOf 419

launch 682

Layout 441

Layout-Manager 434, 452

LayoutManager 452

Lebenszyklus, Thread- 556

Leere Anweisung 88

Leerzeichen 48

LEFT 453, 458, 474

leichtgewichtige Prozesse 547

length 374, 583

Leser/Schreiber-Problem 559

LIGHT\_GRAY 449

lightweight 441

limit 661

line feed 37

lines 659

- LinkedList 420
- List 410, 419
- list 583, 659
- Liste 419
- Listener 434, 510
  - Registrierung 516
- ListSelectionEvent 509, 511
- ListSelectionListener 511
- ListSelectionModel 471
- Literale 46
- Literalkonstanten 46, 59, 60, 62, 63
  - null 46
- LocalDate 691
- LocalDateTime 691
- Locale 408
- LocalTime 691
- logische Aussagen 64
- logische Operatoren 76
  - Oder 74, 77, 339
  - Und 74, 77
- LONG 408
- Long 376
- long 58
- long integer 35
- LongStream 660
- longValue 378
- Look and Feel 518, 729
  
- MAGENTA 449
- main 38, 53, 159, 176
- make 715
- map 661
- Marke 99
- markierte Anweisungen 99
- Maschinensprache 27, 729
- Math 182
- maven 715
- max 386, 392
- MAX\_PRIORITY 558
- MAX\_VALUE 378
- Maximieren 436
- Mebibyte 733
- MEDIUM 408
- Megabyte 733
- Mehrdimensionale Felder 128, 132
- Mehrfachvererbung 276
- Menü 487
- Menüleisten 486
  
- Menge 414
- menuDeselected 513
- MenuEvent 508, 509
- MenuListener 513
- menuSelected 513
- Message 312
- META\_MASK 489
- Methoden 36, 38, 142, 157, 158
  - Argument 159
  - Aufruf 161
  - Call by value 162
  - Default- 283
  - Deklaration 159
  - dynamisches Binden 268
  - Ergebnisrückgabe 160
  - generische 356
  - get- 216
  - Instanz- 184, 203, 213, 214
  - Klassen- 181, 220
  - Kopf 159
  - Name 159
  - Parameter 159
  - Referenzen 649
  - rekursive 171
  - Rückgabotyp 160
  - Rumpf 159
  - set- 216
  - statische 220, 283
  - synchronisierte 562
  - terminieren 172
  - Überladen 164
  - Überschreiben 204, 266
  - variable Argumente 164, 424
  - Wertaufruf 162
- MILLISECOND 402
- min 386, 392
- MIN\_PRIORITY 558
- MIN\_VALUE 378
- Minimieren 436
- MINUTE 402
- mkdir 583
- modal 483
- Modell 195
- Modellierung 29, 195
- Modellierungsphase 206
- Modifikatoren 289
- Modul 675
- Modulsystem 670

- monadische Operatoren 49, 70
- Monitor 563
- MONTH 402
- mouseClicked 511
- mouseDragged 512
- mouseEntered 511
- MouseEvent 508
- mouseExited 511
- MouseListener 511
- MouseMotionListener 512
- mouseMoved 512
- mousePressed 511
- mouseReleased 511
- move 608
- multiply 385, 390
  
- Namen 45
  - Datei- 38
  - für Threads 556
  - Klassen- 38
  - Methoden- 159
- NaN 378
- negate 385, 390
- Negation 74
- NEGATIVE\_INFINITY 378
- net 27
- Netz 27
- Netzwerk 613
- Netzwerkprogrammierung 613
- Netzwerkschicht 614
- new 117, 144
- newInputStream 608
- newLine 590
- newOutputStream 608
- newPriority 549
- next 413
- nextElement 427
- nextToken 427, 591
- nextTokens 427
- NOFOLLOW\_LINKS 608
- NORM\_PRIORITY 558
- NORTH 455
- Notation 71
  - Infix 71
  - Postfix 71
  - Präfix 71
- notify 557, 566
- notifyAll 557, 566
  
- Null-Literal 46
- Null-Referenz 151
- Nullstellen 392
- NumberFormat 395
  
- Oberfläche, grafische 431, 433
- Object 566
- Object 269
- ObjectInputStream 599
- ObjectOutputStream 599
- Objekt 144
  - erzeugen 144
- Objekte 195, 729
- objektorientiert 38
- objektorientierte Programmierung 198, 730
- Oder
  - bedingtes logisches 77
  - exklusives 74
  - logisches 74, 77, 339
- of 659
- oktale Zahlen 59, 730
- OOP 730
- Open Source 91, 730
- openConnection 632
- openStream 631
- Operand 70
- Operationen
  - Abschluss- 658, 661, 662
  - Pipeline- 654, 657
  - Stream- 657
  - Zwischen- 658, 661
- Operator
  - . 145
  - Zugriffs- 145
- Operatoren 48, 49, 70
  - abkürzende Schreibweise 76
  - arithmetische 72
  - Auswertungsreihenfolge 79
  - binäre 49, 70
  - Bit- 74
  - Dekrement- 79
  - Diamond 359, 425
  - dreistellige 49, 70
  - dyadische 49, 70
  - einstellige 49, 70
  - Inkrement- 79
  - logische 76
  - monadische 49, 70

- Notation 71
- Prioritäten 79
- Reihenfolge 71
- Schiebe- 75
- ternäre 49, 70
- triadische 49, 70
- unäre 49, 70
- Vergleichs- 76
- Zuweisungs- 68, 76
- zweistellige 49, 70
- Operatorsymbole 49
- ORANGE 449
- Ordner 26
- OutOfMemoryError 334
- OutputStream 582, 597
- OutputStreamWriter 586
  
- pack 482, 483, 485
- package 288
- paint 530
- paintBorder 530
- paintChildren 530
- paintComponent 530
- Pakete 287
  - java.awt 434, 448
  - java.awt.event 488, 489, 507
  - java.io 316, 582, 591, 604
  - java.lang 182, 287, 369, 372
  - java.math 383
  - java.net 613, 616, 619, 631
  - java.nio 605
  - java.nio.file 606
  - java.text 395, 405
  - java.util 399, 408, 410, 422, 424, 426, 507
  - java.util.functions 645
  - java.util.stream 654
  - javax.swing 434, 447, 488
  - javax.swing.event 507
  - javax.swing.text 473
  - Prog1Tools 287, 718
- Paradigmen 196
- parallele Programmierung 545
- parallelStream 659
- Parameter
  - aktueller 162
  - formaler 159
  - Kommandozeilen- 176
  - Parameterliste 159, 160
  - parse 409
  - parseByte 378
  - parseDouble 378
  - parseFloat 378
  - parseInteger 378
  - parseLong 378
  - parseShort 378
  - paste 473
  - Path 606
  - Paths 606
  - Peer 441
  - Period 691
  - Peripheriegeräte 26
  - Philosophenproblem 569
  - physikalische Schicht 614
  - PI 223
  - PINK 449
  - Pipeline-Operationen 654, 657
  - PLAIN 451
  - Polymorphie 204, 257, 262
  - Popup-Menü 489
  - Port 617
  - Portabilität 731
  - POSITIVE\_INFINITY 378
  - Postfix 71
  - pow 385
  - pow 182
  - präemptives Scheduling 558
  - Präfix 71
  - Predicate 645
  - print 587, 593
  - printf 398
  - println 36, 37, 587, 593
  - PrintStream 601
  - PrintWriter 593
  - Prioritäten 49
    - der Operatoren 79
    - von Threads 558
  - Problemanalyse 29
  - problemorientierte Programmiersprache 28
  - Prog1Tools 287, 718
  - Programm 26
  - Programmieren 29
  - Programmiersprache 27
  - Programmierung
    - Client/Server- 613

- imperative 196
- Netzwerk- 613
- objektorientierte 198
- parallele 545
- Prompt 83, 719
- protected** 289
- Protokoll 614, 731
- Prozesse, leichtgewichtige 547
- Prozessor 26, 731
- public** 143
- Pulldown-Menü 487
- Punktoperator 145
- Python 671
  
- Quellcode 28, 731
- Quellprogramm 28
- Quelltext 28, 731
- Quicksort 173
  
- Rahmen 435, 437
- RAM 26, 731
- RandomAccessFile 605
- range 660
- rangeClosed 660
- read 316, 585, 586, 596, 597
- readBoolean 598
- readByte 598
- readChar 598, 718
- readDouble 598, 718
- Reader 582, 585
- readFloat 598
- readInt 598, 718
- readInteger 718
- readLine 589
- readLong 598
- readObject 599
- readShort 598
- RED 449
- reduce 661
- Referenz 112, 124, 147
  - Null- 151
- Referenzdatentypen 111, 123, 144, 166
- Referenzen
  - Methoden- 649
- Referenzkopie 125, 134, 153, 166
- Regel, Syntax- 41, 42
- Registrierung eines Listeners 516
- Rekursion 171
  - Nachteile 172
  - Vorteile 172
- rekursive Methoden 171
- remainder 385
- remove 412, 413, 420, 447, 466, 487
- removeAll 412, 487
- removeAllItems 468
- removeItem 468
- removeItemAt 468
- renameTo 583
- repaint 529
- Repaint-Manager 529
- replace 374
- REPLACE\_EXISTING 608
- replaceAll 647
- Reservierte Wörter 47
- Rest 61
- Resultatstyp 159
- retainAll 412
- return** 101, 160
- revalidate 537
- RGB-Farbmodell 449, 731
- RIGHT 453, 458, 474
- round 182
- ROUND\_CEILING 391
- ROUND\_DOWN 391
- ROUND\_FLOOR 391
- ROUND\_HALF\_DOWN 391
- ROUND\_HALF\_EVEN 391
- ROUND\_HALF\_UP 391
- ROUND\_UNNECESSARY 391
- ROUND\_UP 391
- Routinen 158
- Routing 615
- RTFM 732
- Rückgabetyt 159, 160
- Rumpf
  - einer Methode 159
  - einer Schleife 95
- run 547–549
- Rundungsfehler 62
- Runnable 548, 552
- RuntimeException 318
  
- Scanner 55, 595
- Schaltflächen 461, 463–465
- Scheduler 557, 558, 732
- Scheduling bei Threads 558
- Schiebeoperatoren 75

- Schleifen **95, 96, 135**
  - abweisende **97**
  - do **97**
  - Endlos- **98**
  - for **95**
  - nicht-abweisende **97**
  - Rumpf **95**
  - unendliche **98**
  - vereinfachte Notation **178, 413, 414**
  - while **97**
- Schließen **436**
- Schlüsselwörter **47**
  - abstract **258**
  - assert **336**
  - catch **315**
  - default **91, 284**
  - extends **202, 259**
  - final **69, 222, 263, 269**
  - finally **331**
  - implements **275**
  - protected **289**
  - public **143**
  - static **221**
  - super **266**
  - synchronized **562**
  - this **215**
    - catch **314**
    - throw **314, 322**
    - throws **317**
    - transient **600**
    - try **314**
    - var **70**
- Schnittstellen **203, 274**
- SECOND **402**
- Seiteneffekt **167**
- Semantik **732**
- semantische Ereignisse **508**
- Semikolon **34, 48**
- Sequenzdiagramm **207**
- Serialisierung **599**
  - Serializable **599**
- Server **618, 732**
- Server-Host **618**
- Server-Rechner **618**
- ServerSocket **619**
- Set **410, 414**
  - set **402, 420**
- set-Methoden **216**
  - setAccelerator **488**
  - setActionCommand **502**
  - setBackground **446**
  - setCharAt **374**
  - setDaemon **549**
  - setDefaultCloseOperation **437, 482, 484**
  - setEditable **468, 473**
  - setEnabled **446**
  - setFocusPainted **461**
  - setFont **446**
  - setForeground **446**
  - setHorizontalAlignment **458, 474**
  - setHorizontalScrollBarPolicy **478**
  - setHorizontalTextPosition **458**
  - setIcon **458**
  - setJMenuBar **482, 485**
  - setLayout **441, 447**
  - setLineWrap **476**
  - setLocation **446**
  - setMnemonic **488**
  - setModal **484**
  - setName **549**
  - setOpaque **447**
  - setPriority **558**
  - setSelected **461**
  - setSelectedIndex **468, 471**
  - setSelectedIndices **471**
  - setSelectedItem **468**
  - setSelectionMode **471**
  - setSize **435, 437, 446**
  - setSoTimeout **630**
  - setText **458, 473**
  - setTime **400, 402**
  - setTimeInMillis **404**
  - setTitle **435, 437, 482, 484**
  - setToolTipText **447**
  - setVerticalAlignment **458**
  - setVerticalScrollBarPolicy **478**
  - setVerticalTextPosition **458**
  - setVisible **435, 437, 446**
  - setWrapStyleWord **476**
- Shell **30**
- SHIFT\_MASK **489**
- SHORT **408**
- Short **376**
- short **58**
- shortValue **378**

- showConfirmDialog 486
- showInputDialog 486
- showMessageDialog 486
- Sichtbarkeit 143, 168
- Simple Mail Transfer Protocol 614
- SimpleDateFormat 405
- SINGLE\_SELECTION 471
- size 412
- Skript 671
- Skriptsprache 671
- sleep 550, 557
- SMTP 614
- Socket 618
  - Socket 619, 630
  - SocketChannel 605
- Software 26
- sort 422, 424, 647
- sorted 661
- SortedSet 416
- Sortieren 422
- Sourcecode 733
- SOUTH 455
- Speicher, externer 26
- Speicherkapazität 733
- Speicherzelle 26
- Sperre 563
- Spezialisierung 199
- Sprungbefehle 99
- sqrt 182
- Stage 682
- Standard Widget Toolkit 431
- Standard-Konstruktor 226
- Standardausgabe 394
- Standardwerte 232
- start 546–548, 556
- Starvation 558
- stateChanged 511
- static 221
- static-Block 229
- statische Importe 83, 183
- statische Initialisierer 229
- statische Komponenten 220
- statische Methoden 220, 283
- Ströme
  - gepufferte 589
- Stream 654, 659, 661
- stream 659
- Stream-Operationen 657
- Streams 654, 657
- StreamTokenizer 591
- String 176, 185, 369
  - Addition 72, 73
  - Konkatenation 72, 73
- StringBuffer 372
- StringTokenizer 426
- Stylesheet 686
- Subklassen 199, 257, 259
- subSet 417
- subtract 385, 390
- Suchen 422
- sum 662
- super 266
- Superklassen 199, 259
- Supplier 646
- Swing 431, 437, 441, 538
- SwingUtilities 519, 577
- switch 91
- SWT 431
- symbolische Konstanten 69, 222
- Synchronisation, Thread- 559
- Synchronisieren 733
- synchronisierte Methoden 562
- synchronized 562
- Syntax 41, 42, 733
  - Regel 41, 42
- System 587
- Systemsoftware 26
- Szenengraph 685
- Tabulatorzeichen 48
- tailSet 417
- Targets 552
- Tastatureingaben 717
- Tastaturfokus 461
- Tastaturkommandos 436
- TCP 614, 615, 733
- Telnet-Programm 623
- Terabyte 733
- Terminal 30
- terminieren 172
- ternäre Operatoren 49, 70
- test 645
- Texteditor 29, 30
- Textkomponenten 473, 476
- this 215
- Thread 548, 628

- Threads **545, 733**
  - Dämon- **558**
  - Deadlock **558**
  - Frames **571**
  - Gruppen **559**
  - Kommunikation **559**
  - Lebenszyklus **556**
  - Namen **556**
  - Scheduling **558**
  - Sicherheit **577**
  - Starvation **558**
  - Swing **571**
  - Synchronisation **559**
    - vorzeitig beenden **554**
- throw** **314, 322**
- Throwable **334**
- throws** **317**
- Tibibyte **733**
- Tiefenkopie **126, 134, 153, 154**
- Timeline **689**
- toArray** **412**
- Token **426**
- Toolbars **489**
- Tooltip **448, 734**
- TOP **458**
- Top-Level-Container **435, 437**
- toString** **219, 270, 373, 386, 392, 395, 400**
- transient** **600**
- Transmission Control Protocol **614**
- Transportschicht **614**
- TreeSet **417**
- Trennzeichen **47**
- triadische Operatoren **49, 70**
- true** **64**
- TRUNCATE\_EXISTING **608**
- try** **314**
  - mit Ressourcen **340, 603**
- TT\_EOF **591**
- TT\_EOL **591**
- TT\_NUMBER **591**
- TT\_WORD **591**
- Typ **36, 71**
  - Daten- **36**
  - Ergebnis- **72**
  - Rückgabe- **159, 160**
- Typ-Parameter **346, 351, 359**
- Typecast **64**
- Typkonvertierung **64**
  - automatische **64, 163, 262**
  - explizite **65**
  - implizite **64, 163, 262**
- Typsicherheit bei Collections **411**
- Typumwandlung **64**
  - automatische **64, 163, 262**
  - implizite **64, 163, 262**
- Typ-Variable **349**
- Typwandlung bei Wrapper-Klassen **380**
- Überladen **164**
  - von Konstruktoren **227, 228**
  - von Methoden **163**
- Überschreiben von Methoden **204, 266**
- Übersetzer **28, 734**
- UDP **614, 615, 734**
- UI **734**
- UI-Toolkit **431**
- UIManager **519**
- Umgebungsvariable **734**
- UML **206, 735**
- Umlaute **45**
- unäre Operatoren **49, 70**
- UnaryOperator **646**
- Unboxing **380**
- Und
  - bedingtes logisches **77**
  - logisches **74, 77**
- unendliche Streams **659, 660**
- Unicode **63, 735**
  - Schreibweise **63**
- Unified Modeling Language **206, 735**
- Uniform Resource Locator **631**
- Unit-Test **712**
- Unterprogramm **157**
- Unterstrich **45, 60**
- Update **735**
- URL **631, 735**
- URL **631**
- URLConnection **632**
- use cases **206**
- Use-Case-Diagramm **207**
- User Datagram Protocol **614**
- validate** **537**
- valueOf** **377**
- var** **70**
- variable Methodenargumente **424**
- Variablen **36, 67**

- Deklaration 36, 68
- Gültigkeitsbereich 89
- indizierte 115
- Initialisierung 69
- Instanz- 142
- Klassen- 142, 220
- Name 36, 68
- Variablendeklaration 36
- VBox 684, 686
- Verdecken 168
- vereinfachte Eingabe 595
- Vererbung 200, 257, 261
- Vergleichsoperatoren 76
- Verkettung, String- 370
- vernetzt 27
- Versionsverwaltung 710
- Verteilungsdiagramm 207
- VERTICAL 489
- VERTICAL\_SCROLLBAR\_ALWAYS 478
- VERTICAL\_SCROLLBAR\_AS\_NEEDED 478
- VERTICAL\_SCROLLBAR\_NEVER 478
- Verzeichnis 26, 582
- VirtualMachineError 334
- virtuelle Maschine 29
- void 159, 160, 176
- Vorzeichen 58
  
- Wahrheitswert 64
- wait 557, 566
- walk 659
- web 27
- Webcontainer 694
- Werkzeugeleisten 489
- Wertaufruf 162
- Wertebereich 57
- WEST 455
- while 97
- WHITE 449
- Wiederholungsanweisungen 95
- Wildcards 353
  - Bounded 354
- windowActivated 513
- WindowAdapter 515
- windowClosed 513
- windowClosing 513
- windowDeactivated 513
- windowDeiconified 513
- WindowEvent 509
- WindowFocusListener 513
- windowGainedFocus 513
- windowIconified 513
- WindowListener 513
- windowLostFocus 513
- windowOpened 513
- windowStateChanged 513
- WindowStateListener 513
- Workaround 736
- Wortschatz 736
- Wortsymbole 47
- Wrapper-Klassen 273, 375
  - Autoboxing 380
  - Autounboxing 380
  - Boxing 380
  - Typwandlung 380
  - Unboxing 380
- WRITE 608
- write 585, 586, 597
- writeBoolean 598
- writeByte 598
- writeChar 598
- writeDouble 598
- writeFloat 598
- writeInt 598
- writeLong 598
- writeObject 599
- Writer 582, 585
- writeShort 598
  
- XML 736
  
- YEAR 402
- YELLOW 449
- yield 549, 557
  
- Zahlen
  - binäre 59
  - ganze 35
  - Gleitkomma- 35, 62
  - hexadezimale 59
  - negative 58
  - oktale 59
- Zeichen 63
- Zeichenkettenliterale 369
- Zeichenströme 582
- Zeichnen 529
- Zeile 129
- Zeilenendezeichen 48

Zeilenvorschub 37  
Zeitangaben 399, 401, 405  
Zeitpunkte 399  
Zeitscheibenverfahren 558  
Zentraleinheit 26  
Zielprogramm 28  
Ziffern 45  
ZipInputStream 605  
ZipOutputStream 605  
ZonedDateTime 691

Zugriffsmethoden 213  
Zugriffsrechte 213, 289  
Zusicherungen 336  
Zuweisung 68  
zuweisungskompatibel 160  
Zuweisungsoperator 68, 76  
Zweierkomplement 58  
zweistellige Operatoren 49, 70  
Zwischenoperationen 658, 661