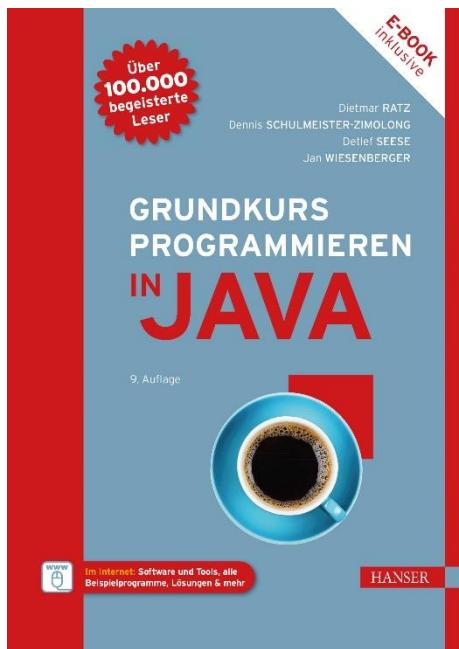


HANSER



Leseprobe

zu

Grundkurs Programmieren in Java

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

Print-ISBN: 978-3-446-48122-0

E-Book-ISBN: 978-3-446-48123-7

E-Pub-ISBN: 978-3-446-48294-4

Weitere Informationen und Bestellungen unter
<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446481220>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhaltsverzeichnis

Vorwort	17
Einleitung	19
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
Über die Autoren	22
I Einstieg in das Programmieren in Java	23
1 Einige Grundbegriffe aus der Welt des Programmierens	25
1.1 Computer, Software, Informatik und das Internet	25
1.2 Was heißt Programmieren?	27
1.3 Welche Werkzeuge brauchen wir?	30
2 Aller Anfang ist schwer	33
2.1 Installation der Entwicklungswerkzeuge	33
2.1.1 Rundum sorglos mit Eclipse	33
2.1.2 Traditionelle JDK-Installation	36
2.2 Mein erstes Programm	37
2.2.1 Quellcode eingeben, übersetzen und ausführen	37
2.2.2 Das Programmgerüst	40
2.2.3 Verwendung von Variablen	41
2.2.4 Formeln, Ausdrücke, Zuweisungen	41
2.2.5 „Auf den Schirm!“	42
2.2.6 Die Kurzversion zum Vergleich	43
2.3 Übungsaufgaben	44
3 Grundlagen der Programmierung in Java	45
3.1 Grundelemente eines Java-Programms	45
3.1.1 Kommentare	47

3.1.2	Bezeichner und Namen	49
3.1.3	Literale	50
3.1.4	Reservierte Wörter, Schlüsselwörter	50
3.1.5	Trennzeichen	51
3.1.6	Interpunktionszeichen	52
3.1.7	Operatorsymbole	53
3.1.8	import -Anweisungen	53
3.1.9	Zusammenfassung	54
3.1.10	Übungsaufgaben	54
3.2	Erste Schritte in Java	55
3.2.1	Grundstruktur eines Java-Programms	56
3.2.2	Ausgaben auf der Konsole	57
3.2.3	Eingaben von der Konsole	58
3.2.4	Schöner programmieren in Java	58
3.2.5	Zusammenfassung	59
3.2.6	Übungsaufgaben	60
3.3	Einfache Datentypen	60
3.3.1	Ganzzahlige Datentypen	61
3.3.1.1	Literalkonstanten in anderen Zahlensystemen	62
3.3.1.2	Unterstrich als Trennzeichen in Literalkonstanten	63
3.3.2	Gleitkommatypen	64
3.3.3	Der Datentyp char für Zeichen	66
3.3.4	Der Datentyp String für Zeichenketten	67
3.3.5	Der Datentyp boolean für Wahrheitswerte	67
3.3.6	Implizite und explizite Typumwandlungen	68
3.3.7	Zusammenfassung	69
3.3.8	Übungsaufgaben	69
3.4	Der Umgang mit einfachen Datentypen	70
3.4.1	Variablen	70
3.4.2	Operatoren und Ausdrücke	74
3.4.2.1	Arithmetische Operatoren	76
3.4.2.2	Bitoperatoren	78
3.4.2.3	Zuweisungsoperator	80
3.4.2.4	Vergleichsoperatoren und logische Operatoren	81
3.4.2.5	Inkrement- und Dekrementoperatoren	83
3.4.2.6	Priorität und Auswertungsreihenfolge der Operatoren	84
3.4.3	Allgemeine Ausdrücke	85
3.4.3.1	Reihenfolge der Operationen in Ausdrücken	85
3.4.3.2	Potenzielle Probleme bei der Auswertung von Ausdrücken	86
3.4.4	Ein- und Ausgabe	87
3.4.4.1	Statischer Import der IOTools-Methoden	89
3.4.5	Zusammenfassung	90

3.4.6 Übungsaufgaben	90
3.5 Anweisungen und Ablaufsteuerung	94
3.5.1 Anweisungen	94
3.5.2 Blöcke und ihre Struktur	94
3.5.3 Entscheidungsanweisungen	95
3.5.3.1 Die if -Anweisung	95
3.5.3.2 Die switch -Anweisung	98
3.5.3.3 Die vereinfachte switch -Anweisung	102
3.5.3.4 Switch-Ausdrücke	104
3.5.4 Wiederholungsanweisungen, Schleifen	105
3.5.4.1 Die for -Anweisung	106
3.5.4.2 Vereinfachte for -Schleifen-Notation	107
3.5.4.3 Die while -Anweisung	107
3.5.4.4 Die do -Anweisung	108
3.5.4.5 Endlosschleifen	109
3.5.5 Sprungbefehle und markierte Anweisungen	110
3.5.6 Zusammenfassung	112
3.5.7 Übungsaufgaben	112
4 Referenzdatentypen	121
4.1 Felder (Arrays)	123
4.1.1 Was sind Felder?	125
4.1.2 Deklaration, Erzeugung und Initialisierung von Feldern	127
4.1.3 Felder unbekannter Länge	129
4.1.4 Referenzen	131
4.1.5 Ein besserer Terminkalender	136
4.1.6 Mehrdimensionale Felder	138
4.1.7 Mehrdimensionale Felder unterschiedlicher Länge	142
4.1.8 Vorsicht, Falle: Kopieren von mehrdimensionalen Feldern	144
4.1.9 Vereinfachte for -Schleifen-Notation	145
4.1.10 Zusammenfassung	146
4.1.11 Übungsaufgaben	147
4.2 Klassen	150
4.2.1 Willkommen in der ersten Klasse!	151
4.2.2 Komponentenzugriff bei Objekten	155
4.2.3 Ein erstes Adressbuch	155
4.2.4 Klassen als Referenzdatentyp	157
4.2.5 Felder von Klassen	160
4.2.6 Vorsicht, Falle: Kopieren von geschachtelten Referenzdatentypen	163
4.2.7 Zusammenfassung	164
4.2.8 Übungsaufgaben	164
5 Methoden, Unterprogramme	167
5.1 Methoden	168

5.1.1	Was sind Methoden?	168
5.1.2	Deklaration von Methoden	169
5.1.3	Parameterübergabe und Ergebnisrückgabe	170
5.1.4	Aufruf von Methoden	172
5.1.5	Überladen von Methoden	174
5.1.6	Variable Argumentanzahl bei Methoden	175
5.1.7	Vorsicht, Falle: Referenzen als Parameter	176
5.1.8	Sichtbarkeit und Verdecken von Variablen	178
5.1.9	Zusammenfassung	180
5.1.10	Übungsaufgaben	180
5.2	Rekursiv definierte Methoden	181
5.2.1	Motivation	181
5.2.2	Gute und schlechte Beispiele für rekursive Methoden	183
5.2.3	Zusammenfassung	186
5.3	Die Methode <code>main</code>	186
5.3.1	Kommandozeilenparameter	187
5.3.2	Anwendung der vereinfachten <code>for</code> -Schleifen-Notation	188
5.3.3	Zusammenfassung	189
5.3.4	Übungsaufgaben	189
5.4	Methoden aus anderen Klassen aufrufen	191
5.4.1	Klassenmethoden	191
5.4.2	Die Methoden der Klasse <code>Math</code>	192
5.4.3	Statischer Import	194
5.5	Methoden von Objekten aufrufen	195
5.5.1	Instanzmethoden	195
5.5.2	Die Methoden der Klasse <code>String</code>	196
5.6	Übungsaufgaben	198
II	Objektorientiertes Programmieren in Java	203
6	Die objektorientierte Philosophie	205
6.1	Die Welt, in der wir leben	205
6.2	Programmierparadigmen – Objektorientierung im Vergleich	206
6.3	Die vier Grundpfeiler objektorientierter Programmierung	208
6.3.1	Generalisierung	208
6.3.2	Vererbung	210
6.3.3	Kapselung	213
6.3.4	Polymorphie	214
6.3.5	Weitere wichtige Grundbegriffe	215
6.4	Modellbildung – von der realen Welt in den Computer	216
6.4.1	Grafisches Modellieren mit UML	216
6.4.2	Entwurfsmuster	217
6.5	Zusammenfassung	218
6.6	Übungsaufgaben	219

7	Der grundlegende Umgang mit Klassen	221
7.1	Vom Referenzdatentyp zur Objektorientierung	221
7.2	Instanzmethoden	223
7.2.1	Zugriffsrechte	223
7.2.2	Was sind Instanzmethoden?	224
7.2.3	Instanzmethoden zur Validierung von Eingaben	227
7.2.4	Instanzmethoden als erweiterte Funktionalität	228
7.3	Statische Komponenten einer Klasse	229
7.3.1	Klassenvariablen und -methoden	230
7.3.2	Klassenkonstanten	232
7.4	Instanziierung und Initialisierung	235
7.4.1	Konstruktoren	235
7.4.2	Überladen von Konstruktoren	237
7.4.3	Der statische Initialisierer	239
7.4.4	Der Mechanismus der Objekterzeugung	241
7.5	Zusammenfassung	246
7.6	Übungsaufgaben	247
8	Vererbung und Polymorphie	267
8.1	Wozu braucht man Vererbung?	267
8.1.1	Aufgabenstellung	267
8.1.2	Analyse des Problems	268
8.1.3	Ein erster Ansatz	268
8.1.4	Eine Klasse für sich	269
8.1.5	Stärken der Vererbung	270
8.1.6	Vererbung verhindern durch final	273
8.1.7	Übungsaufgaben	274
8.2	Die super -Referenz	275
8.3	Überschreiben von Methoden und Variablen	277
8.3.1	Dynamisches Binden	277
8.3.2	Überschreiben von Methoden verhindern durch final	279
8.4	Die Klasse Object und der Umgang mit instanceof und @Override	279
8.4.1	Methoden der Klasse Object sinnvoll überschreiben	280
8.4.2	Hilfe beim Überschreiben: Die Annotation @Override	283
8.4.3	Der Operator instanceof und das Pattern-Matching	286
8.4.4	Das Pattern-Matching für switch	288
8.5	Übungsaufgaben	290
8.6	Abstrakte Klassen und Interfaces	291
8.6.1	Einsatzszenarien am Beispiel	291
8.6.2	Abstrakte Klassen im Detail	294
8.6.3	Interfaces im Detail	297
8.7	Interfaces mit Default-Methoden und statischen Methoden	301
8.7.1	Deklaration von Default-Methoden	301

8.7.2	Deklaration von statischen Methoden	302
8.7.3	Auflösung von Namensgleichheiten bei Default-Methoden	302
8.7.4	Interfaces und abstrakte Klassen im Vergleich	304
8.8	Weiteres zum Thema Objektorientierung	305
8.8.1	Erstellen von Paketen	305
8.8.2	Zugriffsrechte	308
8.8.3	Innere Klassen	309
8.8.4	Anonyme Klassen	315
8.9	Zusammenfassung	318
8.10	Übungsaufgaben	318
9	Exceptions und Errors	331
9.1	Eine Einführung in Exceptions	332
9.1.1	Was ist eine Exception?	332
9.1.2	Übungsaufgaben	334
9.1.3	Abfangen von Exceptions	334
9.1.4	Ein Anwendungsbeispiel	335
9.1.5	Die <code>RuntimeException</code>	338
9.1.6	Übungsaufgaben	339
9.2	Exceptions für Fortgeschrittene	341
9.2.1	Definieren eigener Exceptions	341
9.2.2	Übungsaufgaben	343
9.2.3	Vererbung und Exceptions	343
9.2.4	Vorsicht, Falle!	347
9.2.5	Der <code>finally</code> -Block	349
9.2.6	Die Klassen <code>Throwable</code> und <code>Error</code>	353
9.2.7	Zusammenfassung	355
9.2.8	Übungsaufgaben	355
9.3	Assertions	356
9.3.1	Zusicherungen im Programmcode	356
9.3.2	Ausführen des Programmcodes	357
9.3.3	Zusammenfassung	358
9.4	Mehrere Ausnahmetypen in einem <code>catch</code> -Block	358
9.5	Ausblick: <code>try</code> -Block mit Ressourcen	360
10	Fortgeschrittene Themen der objektorientierten Programmierung	361
10.1	Aufzählungstypen	361
10.1.1	Deklaration eines Aufzählungstyps	362
10.1.2	Instanzmethoden der <code>enum</code> -Objekte	362
10.1.3	Selbstdefinierte Instanzmethoden für <code>enum</code> -Objekte	363
10.1.4	Übungsaufgaben	364
10.2	Generische Datentypen	367
10.2.1	Herkömmliche Generizität	367
10.2.2	Generizität durch Typ-Parameter	369
10.2.3	Einschränkungen der Typ-Parameter	371

10.2.4 Wildcards	373
10.2.5 Bounded Wildcards	375
10.2.6 Generische Methoden	377
10.2.7 Verkürzte Notation bei generischen Datentypen	379
10.2.8 Ausblick	382
10.2.9 Übungsaufgaben	382
10.3 Sortieren von Feldern und das Interface Comparable	387
10.3.1 Einsatz der Klasse Arrays	387
10.3.2 Implementierung des Interface Comparable	388
10.3.3 Übungsaufgaben	390
10.4 Versiegelte Klassen und Interfaces	390
10.4.1 Der Mechanismus der Versiegelung	391
10.4.2 Versiegelung am Beispiel von Klassen	392
10.4.3 Versiegelung am Beispiel mit einem Interface	395
10.4.4 Hilfreiche Konsequenzen für das Pattern-Matching	398
10.4.5 Übungsaufgaben	399
10.5 Records	400
10.5.1 Motivation für die Nutzung von Records	400
10.5.2 Records im Detail	401
10.5.2.1 Regeln zur Vererbung für Records	403
10.5.2.2 Mögliche Ergänzungen von Records	403
10.5.3 Records und das Pattern-Matching	405
10.5.4 Übungsaufgaben	406
11 Einige wichtige Hilfsklassen	407
11.1 Die Klasse StringBuffer	407
11.1.1 Arbeiten mit String-Objekten	407
11.1.2 Arbeiten mit StringBuffer-Objekten	410
11.1.3 Übungsaufgaben	412
11.2 Die Wrapper-Klassen (Hüll-Klassen)	413
11.2.1 Arbeiten mit „eingepackten“ Daten	413
11.2.2 Aufbau der Wrapper-Klassen	415
11.2.3 Ein Anwendungsbeispiel	417
11.2.4 Automatische Typwandlung für die Wrapper-Klassen	418
11.2.5 Übungsaufgaben	420
11.3 Die Klassen BigInteger und BigDecimal	420
11.3.1 Arbeiten mit langen Ganzzahlen	421
11.3.2 Aufbau der Klasse BigInteger	422
11.3.3 Übungsaufgaben	424
11.3.4 Arbeiten mit langen Gleitkommazahlen	425
11.3.5 Aufbau der Klasse BigDecimal	428
11.3.6 Viele Stellen von Nullstellen gefällig?	431
11.3.7 Übungsaufgaben	432
11.4 Die Klasse DecimalFormat	433

11.4.1 Standardausgaben in Java	433
11.4.2 Arbeiten mit Format-Objekten	434
11.4.3 Vereinfachte formatierte Ausgabe	436
11.4.4 Übungsaufgaben	437
11.5 Die Klassen Date und Calendar	437
11.5.1 Arbeiten mit „Zeitpunkten“	438
11.5.2 Auf die Plätze, fertig, los!	439
11.5.3 Spezielle Calendar-Klassen	440
11.5.4 Noch einmal: Zeitmessung	442
11.5.5 Übungsaufgaben	444
11.6 Die Klassen SimpleDateFormat und DateFormat	444
11.6.1 Arbeiten mit Format-Objekten für Datum/Zeit-Angaben .	444
11.6.2 Übungsaufgaben	449
11.7 Die Collection-Klassen	449
11.7.1 „Sammlungen“ von Objekten – der Aufbau des Interface Collection	449
11.7.2 „Sammlungen“ durchgehen – der Aufbau des Interface Iterator	452
11.7.3 Mengen	453
11.7.3.1 Das Interface Set	453
11.7.3.2 Die Klasse HashSet	453
11.7.3.3 Das Interface SortedSet	455
11.7.3.4 Die Klasse TreeSet	456
11.7.4 Listen	457
11.7.4.1 Das Interface List	458
11.7.4.2 Die Klassen ArrayList und LinkedList	458
11.7.4.3 Suchen und Sortieren – die Klassen Collections und Arrays	460
11.7.5 Verkürzte Notation bei Collection-Datentypen	463
11.7.6 Übungsaufgaben	464
11.7.7 Assoziative Sammlungen mit Maps	465
11.7.7.1 Das Interface Map	465
11.7.7.2 Die Klassen HashMap und TreeMap	466
11.7.8 Übungsaufgaben	469
11.8 Die Klasse StringTokenizer	469
11.8.1 Übungsaufgaben	473
III Grafische Oberflächen in Java	475
12 Aufbau grafischer Oberflächen in Frames – von AWT nach Swing	477
12.1 Grundsätzliches zum Aufbau grafischer Oberflächen	477
12.2 Ein einfaches Beispiel mit dem AWT	478
12.3 Let's swing now!	481
12.4 Etwas „Fill-in“ gefällig?	483

12.5 Die AWT- und Swing-Klassenbibliothek im Überblick	485
12.6 Übungsaufgaben	487
13 Swing-Komponenten	489
13.1 Die abstrakte Klasse Component	489
13.2 Die Klasse Container	490
13.3 Die abstrakte Klasse JComponent	491
13.4 Layout-Manager, Farben und Schriften	492
13.4.1 Die Klasse Color	493
13.4.2 Die Klasse Font	495
13.4.3 Layout-Manager	496
13.4.3.1 Die Klasse FlowLayout	497
13.4.3.2 Die Klasse BorderLayout	499
13.4.3.3 Die Klasse GridLayout	500
13.5 Einige Grundkomponenten	502
13.5.1 Die Klasse JLabel	503
13.5.2 Die abstrakte Klasse AbstractButton	505
13.5.3 Die Klasse JButton	505
13.5.4 Die Klasse JToggleButton	507
13.5.5 Die Klasse JCheckBox	508
13.5.6 Die Klassen JRadioButton und ButtonGroup	509
13.5.7 Die Klasse JComboBox	511
13.5.8 Die Klasse JList	514
13.5.9 Die abstrakte Klasse JTextComponent	517
13.5.10 Die Klassen JTextField und JPasswordField	517
13.5.11 Die Klasse JTextArea	520
13.5.12 Die Klasse JScrollPane	522
13.5.13 Die Klasse JPanel	524
13.6 Spezielle Container, Menüs und Toolbars	525
13.6.1 Die Klasse JFrame	526
13.6.2 Die Klasse JWindow	527
13.6.3 Die Klasse JDialog	527
13.6.4 Die Klasse JMenuBar	530
13.6.5 Die Klasse JToolBar	533
13.7 Übungsaufgaben	536
14 Ereignisverarbeitung	539
14.1 Zwei einfache Beispiele	540
14.1.1 Zufällige Grautöne als Hintergrund	540
14.1.2 Ein interaktiver Bilderrahmen	543
14.2 Programmiervarianten für die Ereignisverarbeitung	547
14.2.1 Innere Klasse als Listener-Klasse	547
14.2.2 Anonyme Klasse als Listener-Klasse	547
14.2.3 Container-Klasse als Listener-Klasse	548
14.2.4 Separate Klasse als Listener-Klasse	549

14.3	Event-Klassen und -Quellen	551
14.4	Listener-Interfaces und Adapter-Klassen	554
14.5	Listener-Registrierung bei den Event-Quellen	560
14.6	Auf die Plätze, fertig, los!	563
14.7	Übungsaufgaben	568
15	Einige Ergänzungen zu Swing-Komponenten	573
15.1	Zeichnen in Swing-Komponenten	573
15.1.1	Grafische Darstellung von Komponenten	573
15.1.2	Das Grafikkoordinatensystem	574
15.1.3	Die abstrakte Klasse <code>Graphics</code>	575
15.1.4	Ein einfaches Zeichenprogramm	578
15.1.5	Layoutveränderungen und der Einsatz von <code>revalidate</code> .	580
15.2	Noch mehr Swing gefällig?	582
15.3	Übungsaufgaben	583
IV	Nebenläufige und verteilte Anwendungen	587
16	Parallele Programmierung mit Threads	589
16.1	Ein einfaches Beispiel	589
16.2	Threads in Java	591
16.2.1	Die Klasse <code>Thread</code>	592
16.2.2	Das Interface <code>Runnable</code>	596
16.2.3	Threads vorzeitig beenden	598
16.3	Wissenswertes über Threads	600
16.3.1	Lebenszyklus eines Threads	600
16.3.2	Thread-Scheduling	602
16.3.3	Dämon-Threads und Thread-Gruppen	602
16.4	Thread-Synchronisation und -Kommunikation	603
16.4.1	Das Leser/Schreiber-Problem	604
16.4.2	Das Erzeuger/Verbraucher-Problem	607
16.5	Threads in Swing-Anwendungen	615
16.5.1	Auf die Plätze, fertig, los!	615
16.5.2	Spielereien	618
16.5.3	Swing-Komponenten sind nicht Thread-sicher	621
16.6	Übungsaufgaben	622
17	Ein- und Ausgabe über I/O-Streams	625
17.1	Grundsätzliches zu I/O-Streams in Java	626
17.2	Dateien und Verzeichnisse – die Klasse <code>File</code>	626
17.3	Ein- und Ausgabe über Character-Streams	629
17.3.1	Einfache Reader- und Writer-Klassen	630
17.3.2	Gepufferte Reader- und Writer-Klassen	633
17.3.3	Die Klasse <code>StreamTokenizer</code>	635

17.3.4 Die Klasse <code>PrintWriter</code>	636
17.3.5 Die Klassen <code>IOTools</code> und <code>Scanner</code>	638
17.3.5.1 Was machen eigentlich die <code>IOTools</code> ?	638
17.3.5.2 Konsoleneingabe über ein <code>Scanner</code> -Objekt	639
17.4 Ein- und Ausgabe über Byte-Streams	640
17.4.1 Einige <code>InputStream</code> - und <code>OutputStream</code> -Klassen	641
17.4.2 Die Serialisierung und Deserialisierung von Objekten	643
17.4.3 Die Klasse <code>PrintStream</code>	647
17.5 Streams im <code>try</code> -Block mit Ressourcen	648
17.6 Einige abschließende Bemerkungen	650
17.6.1 Das Paket <code>java.nio</code>	651
17.6.2 Das Paket <code>java.nio.file</code>	651
17.6.2.1 Das Interface <code>Path</code> und die Klasse <code>Paths</code>	652
17.6.2.2 Die Klasse <code>Files</code>	653
17.7 Übungsaufgaben	656
18 Client/Server-Programmierung in Netzwerken	659
18.1 Wissenswertes über Netzwerkkommunikation	660
18.1.1 Protokolle	660
18.1.2 IP-Adressen	662
18.1.3 Ports und Sockets	663
18.2 Client/Server-Programmierung	664
18.2.1 Die Klassen <code>ServerSocket</code> und <code>Socket</code>	665
18.2.2 Ein einfacher Server	667
18.2.3 Ein einfacher Client	670
18.2.4 Ein Server für mehrere Clients	671
18.2.5 Ein Mehrzweck-Client	674
18.3 Übungsaufgaben	677
V Funktionale Programmierung	681
19 Lambda-Ausdrücke, Streams und Pipeline-Operationen	683
19.1 Lambda-Ausdrücke	683
19.1.1 Lambda-Ausdrücke in Aktion – zwei Beispiele	684
19.1.2 Lambda-Ausdrücke im Detail	687
19.1.3 Lambda-Ausdrücke und funktionale Interfaces	689
19.1.4 Vordefinierte funktionale Interfaces	691
19.1.5 Anwendungen auf Datenstrukturen	693
19.1.6 Methodenreferenzen als Lambda-Ausdrücke	695
19.1.7 Zugriff auf Variablen aus der Umgebung innerhalb eines Lambda-Ausdrucks	698
19.1.8 Übungsaufgaben	699
19.2 Streams und Pipeline-Operationen	700
19.2.1 Streams in Aktion	701

19.2.2 Streams und Pipelines im Detail	703
19.2.3 Erzeugen von endlichen und unendlichen Streams	704
19.2.4 Die Stream-API	706
19.2.5 Übungsaufgaben	709
VI Abschluss, Ausblick und Anhang	711
20 Blick über den Tellerrand	713
20.1 JShell für kleine Skripte	714
20.2 Das Java-Modulsystem	718
20.3 Bühne frei für JavaFX	725
20.4 Beginn einer neuen Zeitrechnung	734
20.5 Webprogrammierung und verteilte Systeme	737
20.6 Zu guter Letzt	739
A Der Weg zum guten Programmierer	741
A.1 Die goldenen Regeln der Code-Formatierung	742
A.2 Die goldenen Regeln der Namensgebung	745
A.3 Zusammenfassung	747
B Ohne Werkzeug geht es nicht	749
B.1 Die API-Dokumentation zum Nachschlagen	750
B.2 Die IDE, dein Freund und Helfer	752
B.3 Alle Versionen stets im Griff	754
B.4 Testen bitte nicht vergessen	756
B.5 Der Automat kann es besser	758
C Die Klasse <code>IOTools</code> – Tastatureingaben in Java	761
C.1 Kurzbeschreibung	761
C.2 Anwendung der <code>IOTools</code> -Methoden	762
Glossar	765
Literaturverzeichnis	779
Stichwortverzeichnis	783

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 neunte, 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 Klatte, Niklas Kühl, Roland Küstermann, Jonas Lehner, Joachim Melcher, Cornelia Richter-von Hagen, Sebastian Ratz, Frank Schlottmann, Oliver Schöll, Lukas Struppek, Janna Ulrich, Leonard von Hagen und Lucian 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 „Programmierung I und II“, „Programmierung kommerzieller Systeme“, „Moderne Programmierkonzepte“, „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 Brigitte Bauer-Schiewek, Kristin Rothe und Irene Weilhart vom Carl Hanser Verlag für die gewohnt gute Zusammenarbeit.

Einleitung

Was eint Shakespeare, Goethe, Tolstoi, Fontane und viele mehr? Sie alle haben Weltliteratur hervorgebracht und dabei auch über tragische Liebesbeziehungen geschrieben. Romeo und Julia, Die Leiden des jungen Werthers, Anna Karenina oder Effi Briest sind wohl vielen ein Begriff. Und wer selbst schon einmal unglücklich verliebt war, kann die beschriebenen Geschichten umso besser nachvollziehen. Doch was hat das alles mit diesem Buch zu tun und wie kann es Ihnen in einer solchen Situation helfen? Die traurige Wahrheit ist: Die Gemeinsamkeiten enden beim Wort Buch, und auf schwierige Lebensfragen haben wir leider auch keine Antwort. Auch hilft Ihnen dieses Buch in der Regel nicht, Ihren Schwarm zu beeindrucken oder unliebsamen Wettbewerb loszuwerden. Aber im Ernst: Wozu ist das Buch dann zu gebrauchen? Die folgenden Seiten verraten es Ihnen.

Java – mehr als nur kalter Kaffee?

Seit dem Einzug von Internet und World Wide Web in das öffentliche Leben surfen, mailen und chatten Milliarden von Menschen täglich in der virtuellen Welt. Und seitdem wir mit dem Smartphone auch unterwegs dauerhaft online sein können, ist ein Leben ohne das Internet in vielen Bereichen kaum vorstellbar geworden. Wer nicht mitmacht, gilt als altmodisch oder gar abgehängt. Interessanterweise setzte diese Entwicklung aber nicht erst im letzten Jahrzehnt ein, wie man zunächst denken könnte, sondern schon sehr viel früher, als Anfang der 1990er-Jahre das World Wide Web das Licht der Welt erblickte und selbst so ehrwürdige Organisationen wie Pizza Hut oder der Vatikan auf einmal im Netz der Netze vertreten waren. Oder wussten Sie, dass man bereits seit 1994 Pizza online bestellen kann und 1995 die Webseite des Vatikans online ging?¹

Im selben Jahr erschien auch Java, das dem damaligen Zeitgeist entsprechend als *Programmiersprache des Internets* vermarktet wurde. Schnell entwickelte sich Java deshalb zu einer der populärsten Programmiersprachen überhaupt, die Jahr für

¹ Zur Sicherheit wollen wir hier allerdings darauf hinweisen, dass das Internet und das World Wide Web zwei unterschiedliche Dinge sind. Das Internet, dessen Ursprünge sich bis ins Jahr 1969 zurückverfolgen lassen, bildet das weltumspannende Netz, über welches so unterschiedliche Dienste wie E-Mail, Video-Streaming oder eben das World Wide Web – also das Betrachten von Webseiten im Browser – bezogen werden können.

Jahr weit oben im TIOBE-Index liegt. Und das, obwohl das Sprachkonzept eigentlich nur wenig Neues bietet. Viel mehr orientierte man sich an der mehr als ein Jahrzehnt älteren Programmiersprache C++, vereinfachte diese so sehr, dass sie auch von weniger geübten Programmierer/-innen genutzt werden konnte, und legte darüber hinaus eine Klassenbibliothek bei, die keine Wünsche offen lässt.

Über die Jahre haben sich Programmiersprache und Klassenbibliothek natürlich gewaltig weiterentwickelt und zu einer vollwertigen Konkurrenz zu den anderen gängigen Sprachen gemausert. Datenbank- oder Netzwerkzugriffe, anspruchsvolle Grafikanwendungen, Spieleprogrammierung – alles ist möglich. Und gerade im 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 nutzen. Doch auch jede andere Art von Anwendung ist möglich, inklusive ausgereifter grafischer Benutzeroberflächen – und das sogar plattformunabhängig für alle gängigen Betriebssysteme.

Dies erklärt das große Interesse, das der Sprache Java seither entgegengebracht wird. Bedenkt man die Anzahl von Buchveröffentlichungen, Zeitschriftenbeiträgen, Webseiten, Foren, Blogs und Social Media Posts zum Thema, so wird der erfolgreiche Weg, den die Sprache Java hinter sich hat, offensichtlich. Aber 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.

Java für Anfänger – das Konzept dieses Buches

Wie schreibt man ein Buch über das Programmieren für absolute Neulinge, wenn man selbst seit vielen Jahren programmiert? Folgende Antwort haben wir darauf gefunden: Das Buch soll die grundlegenden Konzepte der Programmierung sowie die Programmiersprache Java möglichst vollständig und korrekt beschreiben, diese aber leicht verständlich vermitteln. Maßstab für die Qualität des Buches ist deshalb, dass es sich optimal zum Selbststudium sowie als Begleitmaterial für Vorlesungen im Bachelor-Studium einsetzen lässt, ohne Vorkenntnisse in den Bereichen Programmieren, Programmiersprachen oder Informatik vorauszusetzen. Vor allem unsere langjährige Erfahrung in der studentischen Programmierausbildung in einführenden und weiterführenden Kursen des Instituts AIFB² am KIT³ sowie des Zentrums für Wirtschaftsinformatik an der DHBW⁴ Karlsruhe sollen Ihnen hier zu Gute kommen.

² Institut für Angewandte Informatik und Formale Beschreibungsverfahren

³ Karlsruher Institut für Technologie

⁴ Duale Hochschule Baden-Württemberg

Beispielsweise gibt es gewisse Erfahrungswerte darüber, welche Themen gerade Neulingen besondere Probleme bereiteten. Daher röhrt der Entschluss, nicht sofort in fortgeschrittene Themen wie die für Java so charakteristische objektorientierte Programmierung einzusteigen, sondern erst einen grundlegenden Wortschatz zu erarbeiten und ein Verständnis dafür zu entwickeln, wie sich Programmieren in Java „anfühlt“. Vergleichen Sie das mit dem Erlernen einer gesprochenen Sprache wie zum Beispiel Spanisch. Auch hier muss man anhand vieler praktischer Beispiele zunächst die Sprache kennenlernen und durch regelmäßige Übung festigen. Dadurch entsteht eine gewisse Routine und Intuition, die es dann auch ermöglicht, tiefer in die Regeln guten Schreibstils oder die Besonderheiten der spanischen Lyrik einzusteigen.

Im ersten Teil des Buches geht es daher überwiegend darum, zunächst einmal die grundlegenden Konzepte der Programmierung zu verinnerlichen und ganz allgemein „algorithmisches Denken“ zu lernen. Die darauffolgenden Teile II bis VI bauen dann darauf auf und führen Sie immer mehr in fortgeschrittene Themen wie Objektorientierung, funktionale Programmierung oder die Entwicklung grafischer Oberflächen ein. Alle Kapitel sind hierfür mit Übungsaufgaben ausgestattet, die Sie zum besseren Verständnis bearbeiten sollten. Denn: *Man lernt eine Sprache nur, indem man sie spricht!*

Selbstredend können und wollen wir nicht auf jedes noch so kleine Detail eingehen, auch wenn Sie sicher feststellen werden, dass wir die Themen alles andere als oberflächlich behandeln. Wir möchten Sie daher bereits an dieser Stelle dazu ermutigen, regelmäßig einen Blick in die sogenannte API-Spezifikation⁵ der Klassenbibliothek [42] zu werfen und sich auch mit anderen Informationsquellen im Internet vertraut zu machen – nicht zuletzt, weil wir im „Programmieralltag“ von einem routinierten Umgang mit diesen Werkzeugen nur profitieren können. Eine kleine Starthilfe hierzu bietet das 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/>

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, eine Liste eventueller Feh-

⁵ API steht für Application Programming Interface, die Programmierschnittstelle für eine Klasse, ein Paket oder eine ganze Klassenbibliothek.

ler 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 **Maschinenschrift** 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 Maschinenschrift** 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. Im Stichwortverzeichnis verweisen fettgedruckte Seitenzahlen auf die Stellen im Buch, an denen die jeweiligen Begriffe eingeführt bzw. definiert werden.

Über die Autoren

- *Prof. Dr. Dietmar Ratz* ist Studiengangsleiter Wirtschaftsinformatik an der Dualen Hochschule Baden-Württemberg (DHBW) Karlsruhe und lehrt auch am Karlsruher Institut für Technologie (KIT).
- *Dipl.-Wirtsch.-Inf. (DH) Dennis Schulmeister-Zimolong* arbeitet als akademischer Mitarbeiter an der Dualen Hochschule Baden-Württemberg (DHBW) Karlsruhe sowie als Produktmanager bei der SOA People AG, Karlsruhe.
- *Prof. Dr. Detlef Seese* ist ehemaliger Professor für Angewandte Informatik am Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) des Karlsruher Instituts für Technologie (KIT).
- *Dipl.-Wi.-Ing. Jan Wiesenberger* ist geschäftsführender Vorstand des FZI Forschungszentrum Informatik in Karlsruhe und Hauptgesellschafter des IT-Dienstleisters m+ps.

Kapitel 2

Aller Anfang ist schwer

Diese sprichwörtliche Feststellung gilt naturgemäß auch für das Erlernen einer Programmiersprache. Die Ursache dieser Anlaufschwierigkeiten liegt möglicherweise darin, dass selbst eine noch so einfach gehaltene Einführung ins Programmieren stets ein gewisses Mindestmaß an Formalismus benötigt, um bestimmte Sachverhalte korrekt wiederzugeben. Einsteiger werden dadurch leicht abgeschreckt und benötigen einige Zeit, um das Ganze zu verdauen. Aber keine Bange: So manche Hürde ist einfacher zu nehmen, als es zunächst den Anschein hat. Sollten Sie also das eine oder andere Detail in unserem ersten Beispiel nicht auf Anhieb verstehen, ist das kein Grund zur Besorgnis. Wir gehen in den folgenden Kapiteln auf jeden der hier beschriebenen Punkte nochmals näher ein. Doch zunächst wollen wir erst einmal die Werkzeuge installieren, die wir zum Programmieren benötigen.

2.1 Installation der Entwicklungswerkzeuge

2.1.1 Rundum sorglos mit Eclipse

In Kapitel 1.3 haben wir schon einen kurzen Blick auf die wichtigsten Werkzeuge für die Programmierung in Java geworfen und dabei gesehen, dass je nach persönlicher Vorliebe zwei Varianten in Frage kommen: Die Nutzung eines einfachen Texteditors zusammen mit dem Java Development Kit oder die Verwendung einer integrierten Entwicklungsumgebung wie Eclipse. Vergleichbar in der physischen Welt mit einem Ausritt zu Pferd, wobei hier vor allem der direkte Kontakt zum Tier im Vordergrund steht,¹ oder einer Fahrt mit der gepolsterten Pferdekutsche zugunsten einer komfortableren Reise.

¹ Anders als beim Java-Compiler handelt es sich bei einem Pferd bekanntlich um ein fühlendes Wesen, auf das man sich aktiv einlassen muss, weshalb sich Mensch und Tier erst aneinander gewöhnen müssen. Letzteres ist bei Java sicher ähnlich, obwohl Programmierneulinge den Java-Compiler zuweilen eher mit einem störrischen Esel als mit einem eleganten Pferd vergleichen würden.

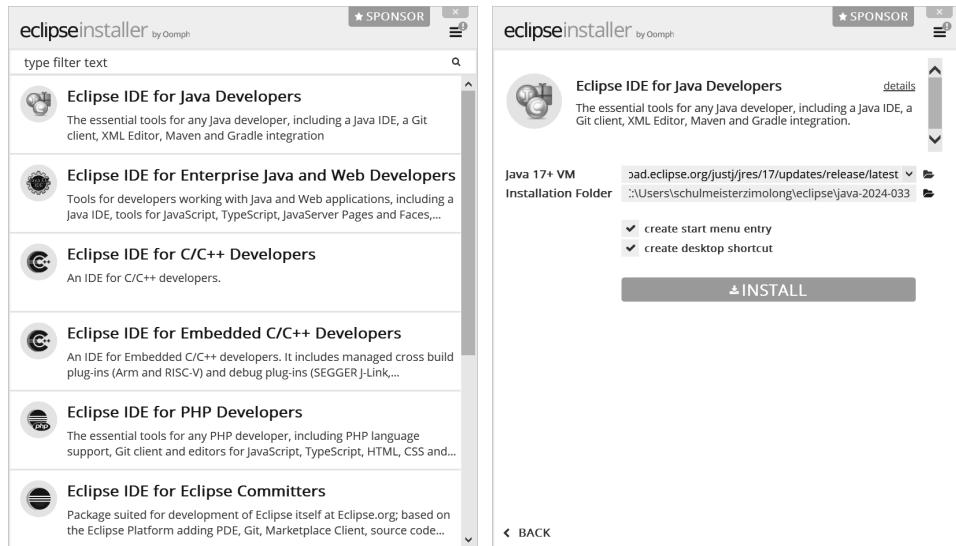


Abbildung 2.1: Die ersten beiden Schritte während der Installation von Eclipse

Innerhalb der Java-Community geht die Tendenz mehr in Richtung der Entwicklungsumgebungen, wobei Programmierneulinge und Profis gleichermaßen deren Vorteile zu schätzen wissen. Im Fall von Eclipse kommt allerdings noch ein Vorteil hinzu: Kauft man die Kutsche, ist das Pferd im Preis inbegriffen. Das Installationsprogramm installiert bei Bedarf auch gleich das Java Development Kit, sodass neben Eclipse keine weiteren Programme heruntergeladen und eingerichtet werden müssen. Sie müssen lediglich von der Webseite [30] das Installationsprogramm herunterladen und darin folgende Punkte auswählen:

- **Zu installierende Variante:** *Eclipse IDE for Java Developers*
- **Im zweiten Schritt:** Bei Java VM die neuste Version auswählen

Daraufhin lädt das Programm alle weiteren benötigten Pakete aus dem Internet und installiert diese auf dem eigenen Rechner. Nach ein paar Minuten und den wirklich immer auftretenden Hinweisen, dass die Installation gerade wesentlich langsamer als üblich laufe, lässt sich Eclipse auch schon starten, sodass Sie direkt loslegen können.

Innerhalb von Eclipse müssen Sie, wie in jeder integrierten Entwicklungsumgebung, erst ein Projekt anlegen, bevor Sie anfangen können zu programmieren.² Hierfür klicken Sie einfach auf *File → New → Java Project* und achten darauf, im daraufhin erscheinenden Fenster folgende Werte korrekt zu setzen:

² Bei größeren Entwicklungen empfiehlt es sich, für jedes Programm ein eigenes Projekt anzulegen. Für die Übungsaufgaben im Buch ist es jedoch einfacher, alles in einem Projekt zu sammeln.

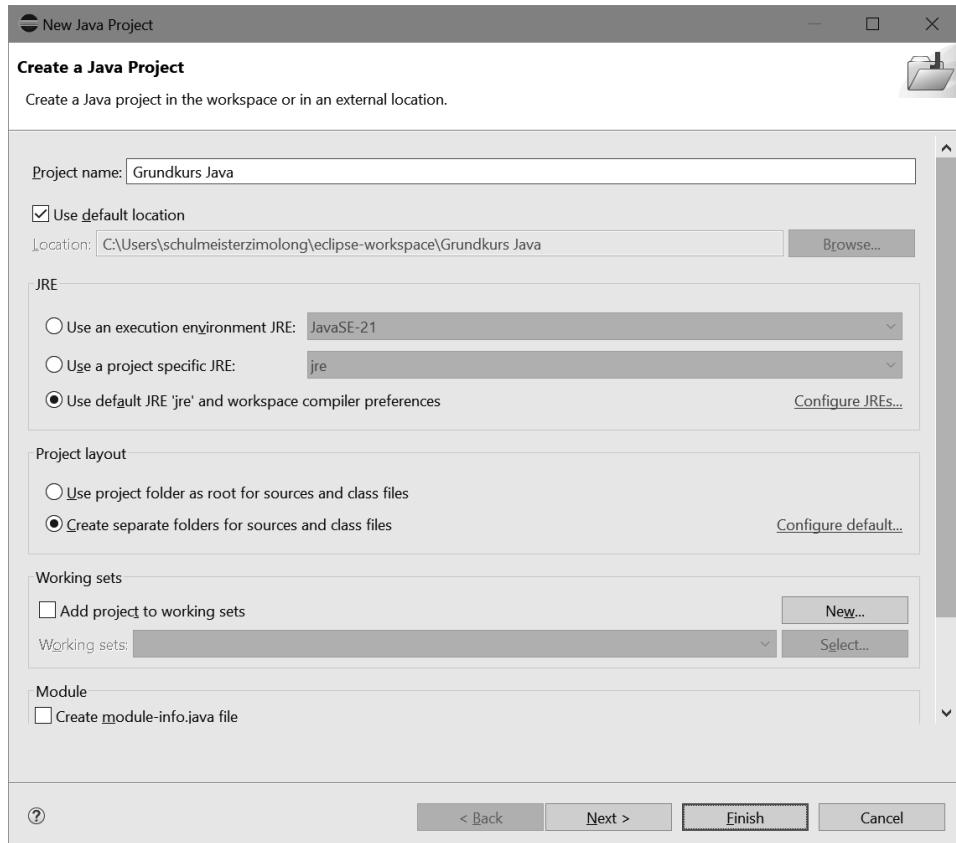


Abbildung 2.2: Anlegen eines neuen Java-Projekts in Eclipse

- **Project name:** Ein beliebiger, sprechender Name wie `Grundkurs Java`
- **JRE:** Hier die dritte Option *Use default JRE and workspace compiler preferences* auswählen, sofern Sie kein separates JDK installiert haben. Andernfalls lassen sich die Quellcodes später nicht compilieren, und Eclipse kennzeichnet viele Stellen im Quellcode als Fehler, die gar keine sind.
- **Module:** Die Checkbox unten bei *Create module-info.java file* sollte nicht gesetzt sein. Denn sonst müssen Sie sich erst mit Paketen und Modulen (vgl. Kapitel 8.8.1 und 20.2) auseinandersetzen. Gerade am Anfang ist das eher hinderlich.

Insgesamt sollte das Fenster wie in Abbildung 2.2 gezeigt aussehen. Anschließend können Sie mit *File → New → Class* neue Quellcodedateien anlegen.³

³ Stören Sie sich hier nicht daran, dass Sie eine neue „Klasse“ anlegen müssen, wenn Sie in Wirklichkeit nur eine Quelldatei benötigen. Wie wir bald sehen werden, beinhaltet in Java jede `.java`-Datei eine sogenannte Klasse mit demselben Namen wie die Datei (ohne Endung).

An vielen Stellen verwenden wir darüber hinaus die in Anhang C beschriebene Klasse `IOTools`, um Tastatureingaben einfacher programmieren zu können. Diese sollten Sie daher gleich von Anfang an dem Eclipse-Projekt hinzufügen, indem Sie sich von unserer Webseite [46] die unter *Tools und Anleitungen* bereitgestellte ZIP-Datei herunterladen und ihre Inhalte, wie auf der Webseite ausführlich beschrieben, in das `src`-Verzeichnis des Projekts hinein kopieren.

2.1.2 Traditionelle JDK-Installation

Falls Sie die direkte Verwendung des Java-Compilers und Interpreters innerhalb der Kommandozeile einer großen IDE vorziehen, lässt sich dies ebenfalls leicht bewerkstelligen. Laden Sie einfach auf der Oracle-Webseite [45] eines der verfügbaren Installationspakete („Installer“) herunter und führen Sie dieses aus. Alternativ können Sie sich von der OpenJDK-Webseite [40] eine um alle nicht-quelloffenen Komponenten bereinigte Version des Java Development Kit herunterladen und diese in einem beliebigen Verzeichnis auf Ihrem Rechner entpacken. Allerdings müssen Sie dann dafür Sorge tragen, dass die `PATH`-Umgebungsvariable des Betriebssystems um das `bin`-Unterverzeichnis des JDKs ergänzt wird. Denn sonst können Sie die beiden Programme `javac` und `java` hinterher gar nicht ausführen, weil sie von der Kommandozeile schlichtweg nicht gefunden werden. Zum Test, ob die Installation vollständig funktioniert hat, öffnen Sie daher bitte ein neues Konsolenfenster⁴ und geben dort das Kommando `java -version` ein. Daraufhin sollte folgende Meldung mit der verwendeten Java-Version erscheinen:

Konsole

```
openjdk version "22" 2024-03-19
```

Der Befehl `javac -version` sollte ebenfalls die aktuelle Java-Version zeigen. Die angezeigte Version kann in beiden Fällen bei Ihnen daher auch neuer sein.

Konsole

```
javac 22
```

Haben Sie dies geschafft, sollten Sie ein neues Verzeichnis für die Beispiele und Übungen des Buchs anlegen und darin auch die Klasse `IOTools` von unserer Webseite [46] aufnehmen. Die Verzeichnisstruktur sollte insgesamt so aussehen, wobei Sie das Hauptverzeichnis auch anderes benennen können:

```
Grundkurs Java ..... Verzeichnis für die Übungen und Beispiele
└── Prog1Tools ..... Paket Prog1Tools aus der ZIP-Datei
    └── IOTools.java ..... Klasse IOTools aus der ZIP-Datei
    └── IOToolsDemo.java ..... Klasse IOToolsDemo aus der ZIP-Datei
    └── Eigene Quellcodedateien
```

Jetzt benötigen Sie lediglich noch einen guten Texteditor, der zumindest rudimentäre Funktionen wie Syntax Highlighting bietet, und schon kann es losgehen.

⁴ Zum Beispiel unter Windows, indem Sie im Startmenü nach „Eingabeaufforderung“ suchen.

2.2 Mein erstes Programm

Wenn man an praktische Anwendungen von Computern denkt, fallen einem sicher zuerst einfache Berechnungen ein. Na ja, oder eben nicht. Vielleicht haben Sie eher an Spiele gedacht, was durchaus okay wäre. Denn natürlich gibt es auch in Java programmierte Spiele wie z. B. Minecraft oder nahezu alle Spiele für die allgegenwärtigen Android-Geräte. Als Einstiegsbeispiel wäre das aber doch zu groß dimensioniert. Anfangen wollen wir deshalb mit etwas viel Einfacherem: Einem Programm, das drei plus vier rechnet und auf dem Bildschirm ausgibt:⁵

```
1 public class Berechnung {  
2     public static void main(String[] args) {  
3         int i;  
4         i = 3 + 4;  
5         System.out.println(i);  
6     }  
7 }
```

Sicher werden Sie jetzt denken, dass das für so eine einfache Aufgabe ganz schön viel Quellcode ist. Und tatsächlich haben Sie Recht, da eigentlich nur die Zeilen drei bis fünf, die man obendrein auch noch zu einer Zeile zusammenfassen könnte, für das eigentliche Problem relevant sind. Anhand dieser Version können wir aber einige Dinge zeigen, die wir später immer wieder benötigen werden. Bevor wir uns diese gleich näher anschauen, nehmen Sie sich daher die Zeit und schreiben den Quellcode ab. Zunächst wollen wir das Programm tatsächlich laufen lassen, bevor wir es weiter analysieren. Die erste Übungsaufgabe haben Sie damit auch gleich bearbeitet. Wie praktisch!

2.2.1 Quellcode eingeben, übersetzen und ausführen

Gleich am Anfang stellen sich schon die ersten Fragen: Wie legt man überhaupt eine neue Quellcodedatei an und welchen Namen soll sie haben? Wäre dies ein Buch für irgendeine andere Programmiersprache, hieße die Antwort oft, dass Sie den Dateinamen lediglich selbst verstehen müssen. In Java gibt es hingegen eine feste Regel, die verlangt, dass die Datei exakt den Namen `Berechnung.java` tragen muss. Denn der Name einer Quellcodedatei muss immer dem Namen der darin enthaltenen Klasse entsprechen, womit die Zeile, die mit `public class` beginnt, gemeint ist.

In Eclipse klicken Sie hierfür im *Package Explorer* am linken Bildschirmrand mit der rechten Maustaste auf das `src`-Verzeichnis und wählen den Menüpunkt *New → Class* aus. Daraufhin öffnet sich das in Abbildung 2.3 gezeigte Fenster, wo Sie nichts weiter tun müssen, als bei *Name* den Wert `Berechnung` einzutragen und auf *Finish* zu klicken. Falls Sie dabei die Warnung stört, dass die Nutzung des **default-Pakets** (englisch: `default package`) vermieden werden sollte, ignorieren Sie diese einfach. Sie wird uns noch bis Kapitel 8.8.1 begleiten.

⁵ Wie könnten wir nur weiter von einem tollen neuen Spiel entfernt sein?

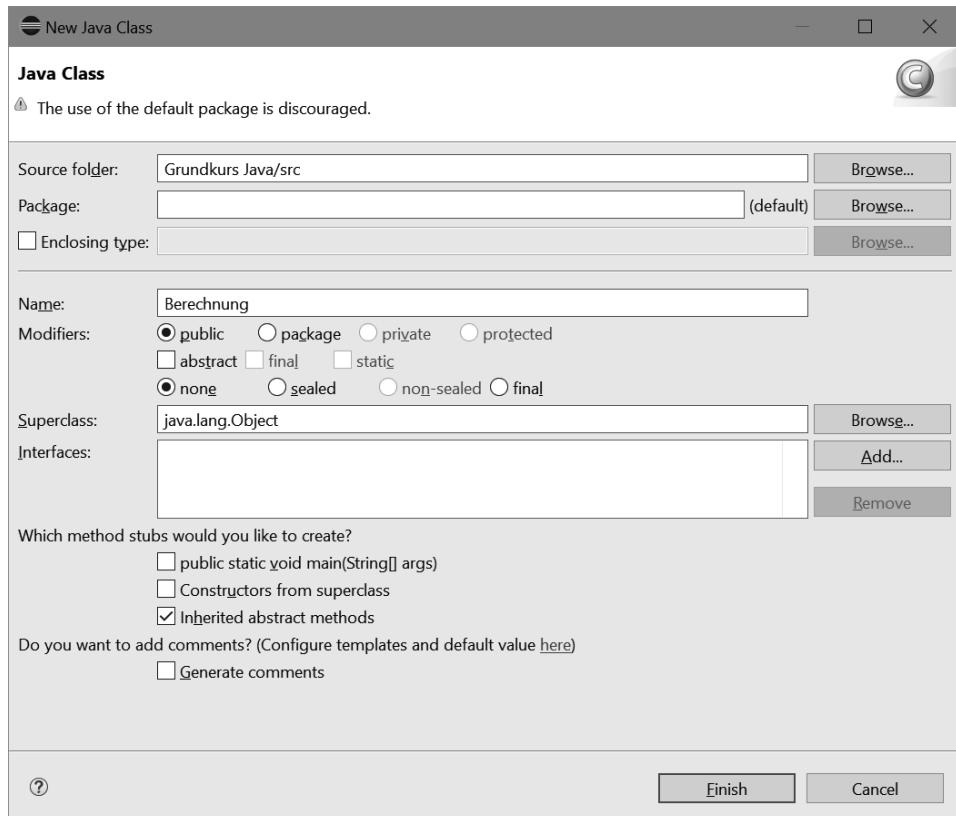


Abbildung 2.3: Anlegen einer Klasse in Eclipse

Arbeiten Sie stattdessen auf der Kommandozeile, legen Sie eine Textdatei mit den Namen `Berechnung.java` an. Beachten Sie allerdings, dass Windows gerne ein unsichtbares `.txt` an den Dateinamen anhängt, das wir nicht gebrauchen können. Um dies zu vermeiden müssen Sie die Dateiendungen im Windows Explorer sichtbar machen, was in jeder Windows-Version leider unterschiedlich erfolgt. Nun können Sie den Quellcode abschreiben. Viel schiefgehen kann dabei nicht, auch wenn die vielen Sonderzeichen sicher etwas gewöhnungsbedürftig sind. Haben wir hier doch gleich das volle Programm (im wahrsten Sinne des Wortes) mit Semikolon, runden Klammern, eckigen Klammern und geschweiften Klammern. Ein Trick, der sich im Alltag daher bewährt hat, ist, nach einer öffnenden Klammer immer gleich die schließende Klammer zu schreiben, wenn sie vom Editor nicht ohnehin automatisch eingefügt wird, um dann den benötigten Inhalt dazwischen zu schreiben. Dann sollten zumindest die Klammern keine großen Probleme machen. Besonders praktisch dabei ist, dass viele Codeeditoren die jeweils andere Klammer optisch hervorheben, wenn der Cursor auf einer Klammer steht,

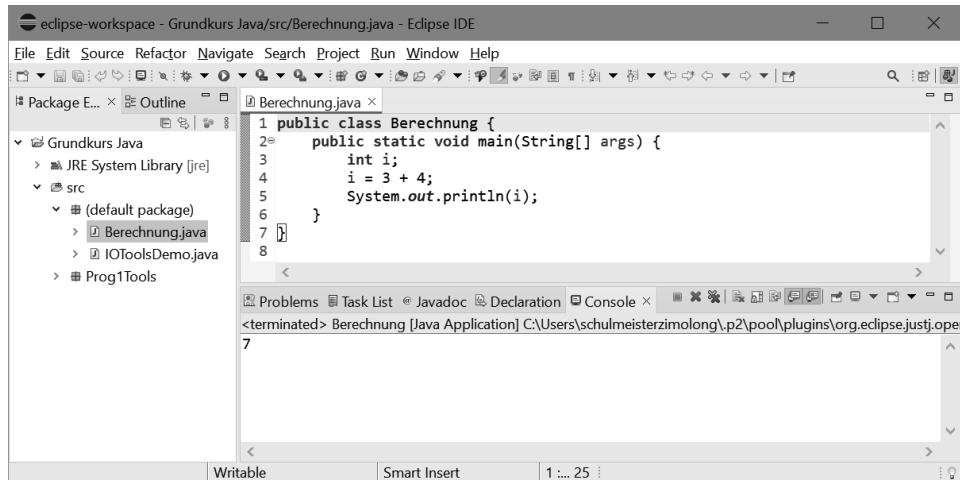


Abbildung 2.4: Quellcode und Konsolenausgabe in Eclipse

oder die Klammerpaare zur besseren Unterscheidung unterschiedlich einfärben. Achten Sie auf solche Details, die bei der Vermeidung von Eingabefehlern helfen sollen, wobei hierzu auch konsistente Einrückungen gehören. Bei näherer Betrachtung werden Sie nämlich feststellen, dass zwischen den Klammern stehende, mehrzeilige Abschnitte immer um eine feste Breite eingerückt sind, wofür Sie sowohl das Tabulator-Zeichen als auch normale Leerzeichen verwenden können.⁶ Anders als die Klammern werden Zeilenumbrüche und Einrückungen zwar vom Compiler ignoriert, sie helfen aber ebenfalls, Eingabefehler früh zu erkennen und den Code leichter nachvollziehen zu können.

Im letzten Schritt muss der Quellcode nur noch übersetzt und ausgeführt werden. Eclipse-Anwender/-innen haben es hier wie immer einfacher. Denn tatsächlich genügt ein einfacher Klick auf den Button mit dem weißen Dreieck (>) im grünen Kreis,⁷ um beide Schritte auf einmal auszuführen. Das Ergebnis sollte dementsprechend wie in Abbildung 2.4 aussehen. Konsolen-Nerds haben dafür mehr Spaß, müssen sie doch erst in das richtige Verzeichnis wechseln⁸ und dann die beiden Befehle `javac Berechnung.java` zum Compilieren sowie `java Berechnung` zum Ausführen des Programms eingeben. Am Ende muss natürlich dasselbe Ergebnis wie in Eclipse erscheinen.

⁶ Drücken Sie zum Einrücken einfach immer die Tabulator-Taste. Je nach Einstellung fügt der Editor dann entweder ein Tabulator-Zeichen oder mehrere Leerzeichen ein.

⁷ Profis erkennen hier natürlich das Play-Symbol, das für die Ausführung des Programms steht.

⁸ Tipp: Unter Windows können Sie im Explorer `cmd` in das Adressfeld schreiben und `ENTER` drücken, um das aktuelle Verzeichnis in einem neuen Konsolenfenster zu öffnen. Andernfalls müssen Sie sich mit dem `cd`-Befehl so lange durchhangeln, bis vor dem Cursor der richtige Verzeichnisname steht.

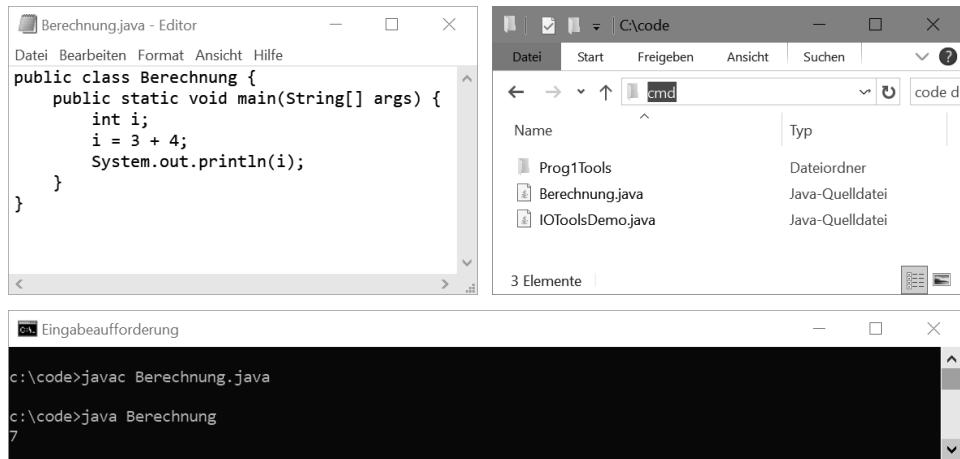


Abbildung 2.5: Eingabe, Übersetzung und Ausführung mit dem JDK im Konsolenfenster

2.2.2 Das Programmgerüst

Nach diesem ersten Erfolgserlebnis wollen wir natürlich verstehen, wie unser kleines Programm funktioniert. Fangen wir damit an, warum dafür überhaupt so viele Zeilen erforderlich sind. Zwar wurde schon gesagt, dass sich die Zeilen drei bis fünf mit der Berechnung und Bildschirmausgabe auch zu einer Zeile hätten zusammenfassen lassen. Dann bleiben aber immer noch vier Zeilen, die zwar einen festen Aufbau zu haben scheinen, deren Sinn sich jedoch nicht so recht erschließt. Am besten merken Sie sich einfach, dass es sich dabei um das Grundgerüst handelt, das jedes Java-Programm besitzt. Es sind sozusagen vier Zeilen, die jedes Programm umrahmen, wobei wir dazwischen die eigentlichen Befehle für den Computer notieren.⁹

```
public class Berechnung {
    public static void main(String[] args) {
        // Hier gehören die auszuführenden Befehle hin
    }
}
```

Wollen wir diese Zeilen genauer verstehen, fangen wir am besten bei den geschweiften Klammern an. Dazu ist es wichtig zu wissen, dass Java zur Strukturierung sogenannte **Blöcke** vorsieht, die immer von geschweiften Klammern umschlossen werden. Somit können wir zwei ineinander geschachtelte Blöcke ausmachen:

1. **Die Klasse:** Sie wird durch die Wörter **public class** `Berechnung` definiert und bildet den äußeren Rahmen des Programms. Ihr Name kann tat-

⁹ Stellen Sie sich das ähnlich wie bei einem Märchen vor, das ja auch oft mit „Es war einmal ...“ beginnt und mit „... und wenn sie nicht gestorben sind, dann leben sie noch heute“ endet. Oder, aus Laiensicht wohl zutreffender, wie ein Zauberspruch der Form „Abrakadabra ... Simsalabim“.

sächlich frei gewählt werden, sollte aber den Java-Konventionen entsprechend jedes Wort mit einem Großbuchstaben beginnen lassen. Später werden wir sehen, dass Java-Programme üblicherweise aus einer Vielzahl von Klassen bestehen. Im Moment können wir die Klasse aber einfach mit dem Programm selbst gleichsetzen.

2. **Die Hauptmethode:** Klassen sind zwar eine wichtige Einheit, in die ein Java-Programm zerlegt werden kann. Um aber wirklich etwas Sinnvolles tun zu können, benötigen sie in der Regel mindestens eine sogenannte Methode, in der die Anweisungen zur Ausführung eines Algorithmus stehen. Diese wird hier durch `public static void main(String[] args)` deklariert, wobei die Schreibweise signalisiert, dass genau diese Methode den Start des Programms markiert und somit immer zuerst ausgeführt werden muss. Sie wird deshalb Hauptmethode oder **main-Methode** genannt.

Die Schreibweise für beides ist aufgrund der vielen **Schlüsselwörter** wie `public`, `static` oder `void` etwas ausschweifend. Natürlich werden wir die Bedeutung davon später noch erklären. Anfangs müssen wir uns aber leider damit zufriedengeben, die Schreibweise einfach auswendig zu lernen.

2.2.3 Verwendung von Variablen

Kommen wir an dieser Stelle nun endlich zu den drei Zeilen mit der eigentlichen Berechnung und Bildschirmausgabe

```
int i;  
i = 3 + 4;  
System.out.println(i);
```

und schauen uns hier insbesondere die erste Zeile genauer an. Sie lautet kurz und knapp `int i;`, was eine **Variablen-deklaration** genannt wird. Hierbei handelt es sich wie bei den anderen beiden Zeilen um eine **Anweisung**, weshalb die Zeile mit einem Semikolon abgeschlossen werden muss. Der Zweck dieser Anweisung ist, eine Variable mit dem zugegeben etwas kurzen Namen `i` zu definieren, um einen Bereich im Hauptspeicher zu reservieren, der später das Ergebnis der Berechnung aufnehmen kann.

Das Schlüsselwort `int` besagt dabei, dass es sich um eine **ganzzahlige** (englisch: `integer`) Variable handelt, die Zahlen im Bereich von ca. $\pm 2,1$ Milliarden umfasst. Später werden wir noch viele weitere Datentypen kennenlernen, u. a. für kleinere oder auch größere Zahlenbereiche. `int` ist jedoch oft eine gute Wahl.

2.2.4 Formeln, Ausdrücke, Zuweisungen

Die nächste Zeile beinhaltet die Berechnung des gewünschten Ergebnisses:

```
i = 3 + 4;
```

Ein wenig mathematisches Grundverständnis vorausgesetzt, sollte diese Zeile nicht allzu schwer zu verstehen sein. Das Semikolon verrät uns allerdings, dass

es sich auch hier um eine Anweisung handelt, die vielleicht doch einen zweiten Blick wert ist. Denn insbesondere das Gleichheitszeichen bedeutet möglicherweise etwas anderes, als Sie vermuten. Es handelt sich hierbei nämlich nicht um ein mathematisches Gleichheitszeichen, das eine formale Gleichheit ausdrückt, sondern um eine **Zuweisung**. Diese liest sich wie folgt:

1. Berechne das Ergebnis des Ausdrucks $3 + 4$.
2. Lege das Ergebnis in der Variable `i` ab.

Das ist vor allem deshalb wichtig, weil wir hier keinen mathematischen Term haben, den wir umstellen können. Folgende Zeilen sind daher zwar mathematisch korrekt, für den Compiler aber dennoch falsch:

```
i - 3 = 4;
i - 3 - 4 = 0;
3 + 4 = i;
```

Sie alle verletzten die Regeln für Zuweisungen, die vereinfacht gesagt lauten:

- Ein einzelnes Gleichheitszeichen steht immer für eine Zuweisung.
- Links vom Gleichheitszeichen muss eine Variable stehen.
- Rechts vom Gleichheitszeichen muss ein **Ausdruck** stehen.
- Ein Ausdruck führt zur Laufzeit immer zu einem eindeutigen Ergebnis.
- Das Ergebnis des Ausdrucks wird in der Speicherzelle der Variable abgelegt.

Daraus folgt, dass mathematische Formeln gültige Ausdrücke sind und daher von Java ausgewertet und berechnet werden können. Dies ist kaum verwunderlich, wenn man bedenkt, dass sich die Informatik als eigenständige Disziplin aus der Mathematik herausgebildet hat.

2.2.5 „Auf den Schirm!“

Mit diesem Vorwissen dürfte die dritte Zeile nun kein allzu großes Problem sein:

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

Wir erkennen wieder das Semikolon, das auch diese Zeile als Anweisung kenntlich macht, sowie den Ausdruck `i`, der den zuvor in der gleichnamigen Variable abgelegten Wert als Ergebnis liefert. Der allgemeinen Logik nach muss `System.out.println()` demnach eine Anweisung sein, die in ihren runden Klammern einen Ausdruck erwartet und dessen Ergebnis anzeigt:

Konsole

7

Super, alles richtig! Doch um was für einen Ausdruck handelt es sich dabei genau? Und warum wird er genau so und nicht irgendwie anders geschrieben? Wie wir

später noch sehen werden, stehen die runden Klammern hinter einem Bezeichner¹⁰ für einen **Methodenaufruf**, was bedeutet, dass hier an eine andere Stelle im Programm abgesprungen und zunächst diese ausgeführt wird, bevor das Programm an der Stelle nach dem Methodenaufruf fortfährt. Innerhalb der runden Klammern stehen durch Komma getrennte Ausdrücke, die der **Methode** als **Parameter** übergeben werden. Die Idee dahinter ist, dass eine Methode einen Algorithmus kapselt, hierfür eine bestimmte Anzahl Werte als Eingangsparameter benötigt und als Ergebnis einen neuen Wert liefern kann.¹¹

Allerdings haben wir so eine Methode doch gar nicht ausprogrammiert. Und zum gegenwärtigen Zeitpunkt wüssten wir auch nicht einmal, wie wir das tun sollten. Müssen wir aber auch gar nicht, weil `System.out.println` in der Klassenbibliothek von Java enthalten ist und dem Compiler somit schon fix und fertig vorliegt. Wann immer wir einen Text auf der Konsole ausgeben wollen, können wir sie verwenden. Mehr müssen wir an dieser Stelle noch nicht verstehen.

2.2.6 Die Kurzversion zum Vergleich

Da es inzwischen öfters erwähnt wurde, wollen wir Ihnen die Kurzversion des Programms, in der die Zeilen drei bis fünf zusammengefasst wurden, nicht vorenthalten. Sie lautet wie folgt:

```
1 public class Berechnung {  
2     public static void main(String[] args) {  
3         System.out.println(3 + 4);  
4     }  
5 }
```

Übrig geblieben sind `System.out.println(...)` für die Bildschirmausgabe sowie der Ausdruck `3 + 4` zur Berechnung des darzustellenden Werts. Weggefallen sind hingegen die Variable `i` und dementsprechend die Zuweisung des Rechenergebnisses an diese.

Möglich wird dies, weil `System.out.println` zwischen den runden Klammern einen beliebigen Ausdruck erwartet, der sich in eine Textform umwandeln lässt, wobei sowohl die Berechnung `3 + 4` als auch die bloße Nennung der Variablen `i` diese Bedingung erfüllen. Im ersten Fall kümmert sich Java selbstständig darum, das Rechenergebnis im Hauptspeicher abzulegen, falls es zur Ausführung der sie umgebenden Anweisung benötigt wird.¹² Im zweiten Fall führt die Auswertung des Ausdrucks `i` dazu, dass der Wert der Variablen gelesen und als Zwischenergebnis eingesetzt wird.

¹⁰ Bezeichner sind Wörter, die nicht zum Sprachumfang von Java gehören, also keine reservierten Schlüsselwörter sind. Es handelt sich dabei um Namen, die wir unseren Klassen, Methoden, Variablen und anderen Dingen geben, damit wir uns später auf diese beziehen können.

¹¹ Ja richtig: Methodenaufrufe sind Ausdrücke und können daher an jeder Stelle verwendet werden, an der ein Ausdruck erwartet wird.

¹² Tatsächlich kann der Compiler hier sehr wahrscheinlich auf die Ablage im Hauptspeicher verzichten, da sowohl die Operanden als auch das Ergebnis der Berechnung in den sogenannten Registern des Prozessors abgelegt werden können. Streng genommen könnte er sogar auf die gesamte Berechnung verzichten und nur das Ergebnis übernehmen, weil sich dieses ja niemals ändern wird.

2.3 Übungsaufgaben

Aufgabe 2.1

Falls Sie es bis hierhin noch nicht getan haben, folgen Sie den Anweisungen weiter vorne im Buch zur Installation von Java und ggf. Eclipse. Legen Sie dann ein neues Projekt an und schreiben das im vorherigen Teilkapitel besprochene Programm ab, übersetzen es und lassen es laufen. Versuchen Sie, etwas Routine hierbei zu bekommen. Es wird ja bei Weitem nicht das letzte Programm sein, das Sie im Rahmen des Buches eingeben und ausführen müssen.

Aufgabe 2.2

Legen Sie eine neue Klasse an und übernehmen Sie folgenden Quellcode hierin. Lassen Sie auch dieses Programm zunächst laufen, bevor Sie mit der nächsten Aufgabe weitermachen.

```
1 public class Uebung {  
2     public static void main(String[] args) {  
3         System.out.println("Guten Tag!");  
4         System.out.println("Mein Name ist Puter, Komm-Puter.");  
5     }  
6 }
```

Aufgabe 2.3

Was passiert, wenn Sie im vorigen Programm in Zeile 3 das Semikolon entfernen? Was passiert, wenn Sie statt einem zwei Semikolons einfügen? Warum ist das eine ein Fehler und das andere nicht? Probieren Sie es aus.

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-

Student	
name:	String
nummer:	int

Abbildung 7.1: Die Klasse Student, erste Version

schlossen, die Klasse `Student` zu nennen. Natürlich soll diese Klasse aber sowohl weibliche als auch männliche Student(inn)en modellieren.¹

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.² 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

¹ 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.

² 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!

Stattdessen setzen wir die Zugriffsrechte nun auf **private**, dem genauen Gegen teil zum öffentlichen Zugriff: Während bei öffentlichem Zugriff jede Klasse auf die Variablen Zugriff hat, kann bei privatem Zugriff außer der eigenen Klasse *keine* andere Klasse auf die Variablen zugreifen, nicht einmal eigene Subklassen. Da nur die Klasse Zugriff hat, in der die Instanzvariablen definiert sind, handelt sich hierbei also wirklich um ihre *privaten* Variablen, die nur dieser Klasse „gehören“, die aber trotzdem an Subklassen vererbt werden.

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

Student	
-name:	String
-nummer:	int

Abbildung 7.2: Die Klasse Student, zweite Version

Die 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 */
5      private String name;

```

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

```

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 Compilieren 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 169 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:

Student	
-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» . «VARIABLENNNAME»

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

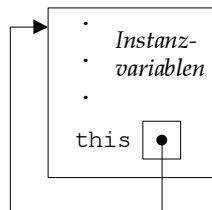


Abbildung 7.4: Die `this`-Referenz

■ 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.⁴ Ausformuliert lautet das wie folgt:

```
/** Gib die Matrikelnummer des Studenten als Integer zurück */
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 compilieren. Der Grund dafür liegt darin, dass der Compiler 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;
}
```

⁴ 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 binden⁵ – 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 != 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 */
```

⁵ 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 gültig
        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

Konsole

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`. Instanziieren 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 Objekt⁶ 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 zuordnen zu können, hat uns Perspektiven erschlossen, die wir mit unseren bisherigen Programmiererfahrungen nicht sahen.

⁶ 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 instanzierten 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:⁷

⁷ Der initiale Wert könnte an dieser Stelle auch wegfallen.

Syntaxregel

```
private static «TYP» «VARIABLENNNAME» = «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++; // erhöhe 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.

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

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 einzuziehen, 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 π (gesprochen „pi“, etwa 3.14...) benennen will. Hierbei gilt ja als Konvention, dass wir Konstanten in unseren Programmen immer groß schreiben. Im Falle von π 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 *Studiengang* an einer Hochschule einschreibt, wird dieses Fach in den Systemen vieler Hochschulverwaltungen mit einer bestimmten Nummer identifiziert.

Tabelle 7.1: Zuordnung Studienfach – Verwaltungsnummer

Studiengang	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 231 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();
```

instanziert, wobei der **new**-Operator angewendet wurde, entsprechend der bereits auf Seite 154 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 Rückgabetyp (nicht einmal **void**). Die Definition des Konstruktors erfolgt nach dem Schema:⁸

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 verbinden 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 Compiler automatisch erzeugt. Er wird vom System aufgerufen, wenn wir z. B. mit

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

ein Objekt instanziieren. 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();
}
```

⁸ 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 231 angemahnte Abwärtskompatibilität⁹ 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++;
    geburtsjahr = geburtsjahr;
```

⁹ 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 aufführen. 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 repräsentiert
   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 Systemstart¹⁰ automatisch zu initialisieren. Hierfür verwenden wir den so genannten **statischen Initialisierer**, umgangssprachlich oft einfach **static**-Block genannt.¹¹ Statische Initialisierer werden nach folgender Regel erschaffen:

Syntaxregel

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

¹⁰ Genauer gesagt, wenn wir die Klasse zum ersten Mal verwenden.

¹¹ 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 verstößen. 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, verstößen? 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 manipuliert¹² 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 Konstruktoren 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 Studentenname bei der Initialisierung dann gesetzt? Ist er dann „Namenlos“ oder ein „DummyStudent“?

¹² 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.

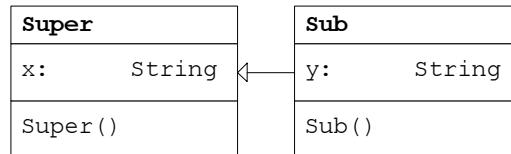


Abbildung 7.6: Beispielklassen für Abschnitt 7.4.4

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 Konstruktors¹³ 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

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

Tabelle 7.2: Default-Werte von Instanzvariablen

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

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 öffentliche 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    }
```

```

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 öffentliche 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 Instanziierungsvorgang 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.

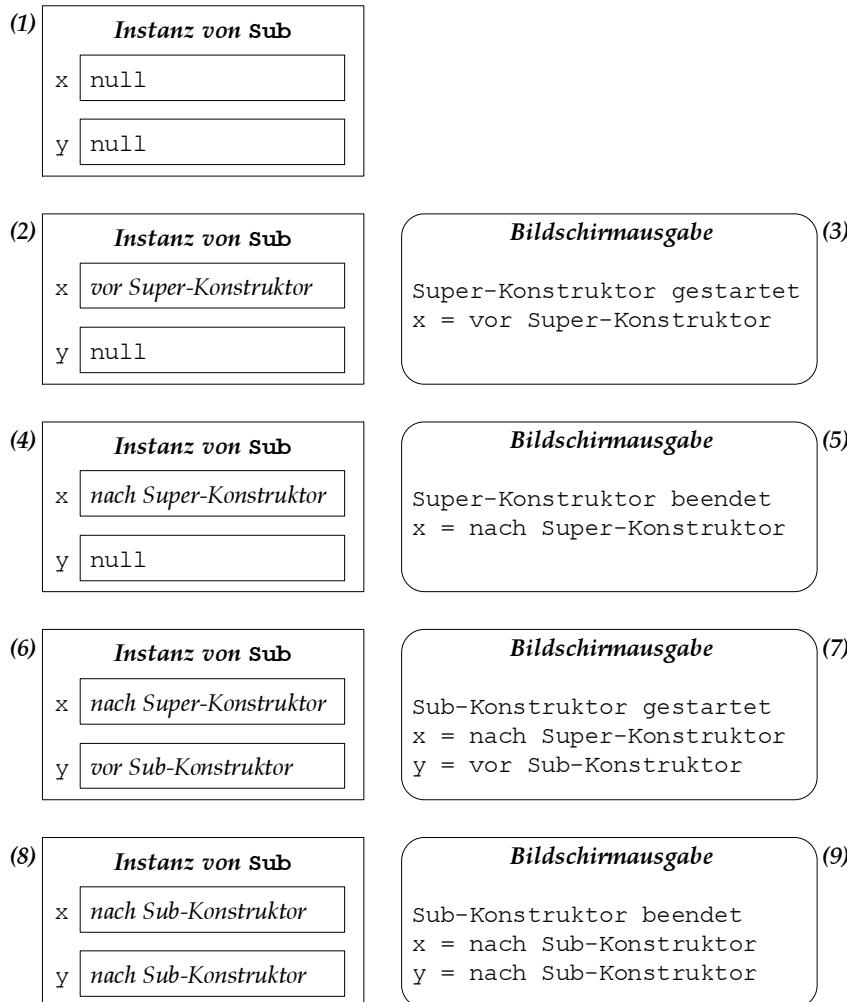


Abbildung 7.7: Instanziierungsprozess von Sub- und Superklasse

7. 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“.
8. Zuletzt werden die Variablen **x** und **y** wiederum auf einen neuen Wert gesetzt (Zeile 11 und 12 im Programmtext der Klasse Sub).
9. 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 Instanziierungsprozesses 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 Instanziierungsprozess 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“ Σ 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 Σ 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 Einziffern 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 ... über- schrieben 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 ... auf- gerufen.
- 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 = l;  
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 += l;  
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     }  
39 }
```

```

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 Instantiierung 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  /** Konstruktor */
6  public Fehler1(String name) {
7      name = name;
8  }
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) ganzzählige 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 lesernder 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).

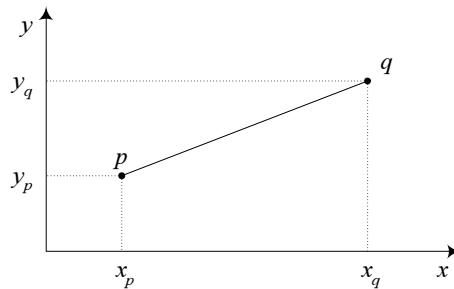


Abbildung 7.8: Definition einer Strecke

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:

- a) 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 stattet 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 stattet 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     }
```

```

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 }
```

- a) 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).
- b) Geben Sie an, was das Programm ausgibt.
- c) 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 }
```

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

Patient maier;
maier = new Patient();
```
- c) 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 fährt auf 2 Reifen mit je 4.5 bar
Autol steht auf 4 Reifen mit je 1.9 bar
Zustand 2:
Fahrrad1 fährt auf 2 Reifen mit je 4.5 bar
Autol fährt auf 4 Reifen mit je 1.9 bar
Zustand 3:
Fahrrad1 steht auf 2 Reifen mit je 4.5 bar
Autol fährt auf 4 Reifen mit je 1.9 bar
Zustand 4:
Fahrrad1 steht auf 2 Reifen mit je 4.5 bar
Autol 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.

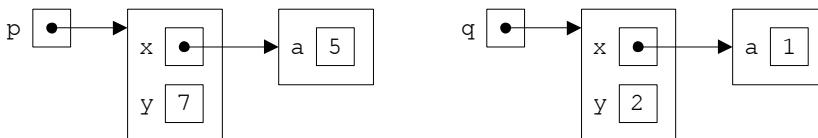


Abbildung 7.9: Ausgangsbild Aufgabe 7.15

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 Trinkglasses durch

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

berechnet werden.

- 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

Fettgedruckte Seitenzahlen verweisen auf die Stellen im Buch, an denen die jeweiligen Begriffe eingeführt bzw. definiert werden.

|     |             |                             |                                                  |
|-----|-------------|-----------------------------|--------------------------------------------------|
|     | 79, 82, 359 | &&                          | 82                                               |
|     | 82          | ^                           | 79                                               |
| =   | 81          | ~                           | 79                                               |
| *   | 76          |                             |                                                  |
| */  | 48          |                             |                                                  |
| *=  | 81          |                             |                                                  |
| +   | 53          |                             |                                                  |
| ++  | 53          |                             |                                                  |
| +=  | 81          |                             |                                                  |
| -   | 53          |                             |                                                  |
| --  | 53          |                             |                                                  |
| --= | 81          |                             |                                                  |
| ->  | 687         |                             |                                                  |
| /   | 76          |                             |                                                  |
| /*  | 48          |                             |                                                  |
| /** | 48          |                             |                                                  |
| //  | 47          |                             |                                                  |
| /=  | 81          |                             |                                                  |
| ::  | 696         |                             |                                                  |
| <   | 81          |                             |                                                  |
| <<  | 79          |                             |                                                  |
| <=  | 81          |                             |                                                  |
| <>  | 380, 463    |                             |                                                  |
| ==  | 53, 81      |                             |                                                  |
| >   | 81          |                             |                                                  |
| >=  | 81          |                             |                                                  |
| >>  | 79          |                             |                                                  |
| ? : | 53, 82      |                             |                                                  |
| %   | 76          |                             |                                                  |
| %=  | 81          |                             |                                                  |
| &   | 79, 82      |                             |                                                  |
| &=  | 81          |                             |                                                  |
|     |             | Ablaufsteuerung             | 94                                               |
|     |             | abs                         | 424, 430                                         |
|     |             | Abschlussoperationen        | 704, 707, 708                                    |
|     |             | <b>abstract</b>             | 268                                              |
|     |             | Abstract Window Toolkit     | 485                                              |
|     |             | AbstractButton              | 505                                              |
|     |             | abstrakte Klassen           | 268, 291, 292, 294                               |
|     |             | Absturz                     | 765                                              |
|     |             | accept                      | 665, 691                                         |
|     |             | ActionEvent                 | 552, 553                                         |
|     |             | ActionListener              | 541, 555                                         |
|     |             | actionPerformed             | 541, 555                                         |
|     |             | Adapter-Klassen             | 558                                              |
|     |             | add                         | 423, 428, 441, 450, 458, 485, 490, 491, 510, 531 |
|     |             | addActionListener           | 542                                              |
|     |             | addAll                      | 450                                              |
|     |             | addItem                     | 512                                              |
|     |             | addSeparator                | 531                                              |
|     |             | Adresse                     | 71, 122                                          |
|     |             | after                       | 438, 440                                         |
|     |             | Aggregation                 | 216                                              |
|     |             | aktueller Parameter         | 172                                              |
|     |             | algorithmische Beschreibung | 29                                               |
|     |             | Algorithmus                 | 26, 765                                          |
|     |             | Alias-Namen                 | 662                                              |
|     |             | allMatch                    | 707                                              |
|     |             | ALT_MASK                    | 533                                              |

ancestorAdded 556  
AncestorEvent 552  
AncestorListener 556  
ancestorMoved 557  
ancestorRemoved 557  
anonyme Klasse 315, 547  
ant 759  
Anteil, ganzzahliger 64  
Anweisung 41  
Anweisungen 94  
    Ausdrucks- 94  
    **break** 110  
    **case** 98  
    **continue** 111  
    **default** 98  
    **do** 108  
    Entscheidungs- 95  
    **for** 106  
    **if** 95  
    **import** 53  
    leere 94  
    markierte 110  
    **return** 112, 170  
    **switch** 98, 102  
    **while** 107  
        Wiederholungs- 105  
Anwendungsfälle 216  
Anwendungsschicht 660  
Anwendungssoftware 26  
anyMatch 707  
API 765  
API-Dokumentation 750  
API-Spezifikation 192  
APPEND 654  
append 410  
Application 727  
Applikation 55, 765  
Applikationsserver 738  
apply 691, 692  
Arbeitsspeicher 26, 122, 765  
args 169, 186  
Argument, formales 169  
Argumentliste 169  
ArithmeticException 332  
Arithmetische Operatoren 76  
arraycopy 135, 177  
ArrayList 458  
Arrays 462  
asList 463  
Assemblersprache 27  
**assert** 356  
AssertionError 356  
Assertions 356  
assoziative Datenstruktur 465  
ATOMIC\_MOVE 654  
Aufruf von Methoden 172  
Aufzählungstypen 361  
Ausdrücke 42, 74, 85  
    konstante 78  
    Lambda- 683, 687  
    Switch- 104  
    Wächter- 289  
Ausdrucksanweisung 94  
Ausgabe 57, 433, 434, 625, 629, 640, 650  
    formatierte 436  
Ausgabegeräte 26  
Ausnahme 332, 358  
Aussage, logische 67  
Auswahllisten 511, 514  
Auswertungsreihenfolge 84  
Autoboxing 418  
AutoCloseable 360, 650  
Automat 25  
automatische Typkonvertierung 68, 173, 272  
automatische Typumwandlung 68, 173, 272  
Autounboxing 418  
average 708  
AWT 485  
AWTEvent 552  
Basis-Container 479, 481  
Baukastenprinzip 477  
Beautifier 97  
bedingtes logisches Oder 81  
bedingtes logisches Und 81  
Beenden 480  
before 438, 440  
Behälter 477  
Benutzungsschnittstelle, grafische 477  
Berechnungen 74  
Betriebssystem 766  
Bezeichner 49  
BiConsumer 691  
BiFunction 691  
BigDecimal 428

BigInteger 422  
Bildlaufleisten 522  
binäre Operatoren 53, 75  
binäre Zahlen 62, 766  
Binärfolge 61  
BinaryOperator 692  
binarySearch 461, 462  
Binden  
  dynamisches 278  
BiPredicate 691  
Bit 78, 776  
Bitoperatoren 78  
BLACK 493  
Block 40, 52, 56, 94  
  **static**- 239  
  Struktur 94  
BLUE 493  
BOLD 495  
Boolean 414  
**boolean** 67  
booleanValue 416  
BorderLayout 499  
BOTTOM 502  
Bounded Wildcards 375  
Boxing 418  
**break** 98, 110  
Browser 766  
Buchstaben 49  
Buffer 651  
BufferedInputStream 643  
BufferedOutputStream 643  
BufferedReader 633, 705  
BufferedWriter 633  
Bug 766  
Bugfix 766  
ButtonGroup 509  
Byte 78, 775  
Byte 414  
**byte** 61  
Byte Streams 626  
Byte-Ströme 626  
Bytecode 28, 30, 766  
byteValue 416  
  
Calendar 440  
Call by reference 176  
Call by value 172  
canRead 627  
canWrite 627  
CaretEvent 553  
CaretListener 555  
caretUpdate 555  
**case** 98  
Cast 68  
**catch** 334, 335  
  mehrfaches 359  
CEILING 429  
CENTER 497, 499, 502, 518  
ChangeEvent 552, 553  
ChangeListener 555  
Channel 651  
**char** 66  
Character 414  
Character Streams 626  
charValue 416  
CheckedInputStream 650  
CheckedOutputStream 650  
clamp 194  
**class** 40  
clear 451, 465  
clearSelection 514  
Client 664, 767  
Client-Host 664  
Client-Rechner 664  
Client/Server-Programmierung 659  
clone 136  
close 629, 641, 650, 666  
CMD 30  
Code Formatter 97  
Codeblock 56  
**Codierung** 29  
Collection 449, 450, 704  
Collections 460  
Color 493  
Comparable 388, 455  
Comparator 694  
compare 694  
compareTo 415, 424, 430, 438, 455  
Compiler 28, 30, 767  
Component 489  
componentAdded 556  
ComponentEvent 552  
componentHidden 556  
ComponentListener 556  
componentMoved 556  
componentRemoved 556

componentResized 556  
componentShown 556  
Computer 25, 767  
Computersystem 26  
Consumer 691  
Container 477, 478, 524, 526, 527  
Container 484, 490  
ContainerEvent 552  
ContainerListener 556  
contains 451  
containsAll 451  
containsKey 465  
containsValue 465  
Content-Pane 484  
**continue** 111  
copy 517, 518, 653  
COPY\_ATTRIBUTES 653  
count 707  
countTokens 470  
CREATE 654  
createNewFile 627  
CTRL\_MASK 533  
currentThread 593  
cut 517, 518  
CYAN 493

Dämon-Threads 602  
DARK\_GRAY 493  
Data Binding 734  
data hiding 213, 223  
DataInputStream 641, 642  
DataOutputStream 641, 642  
Date 438  
Date/Time-API 714, 735  
DateFormat 444  
Datei 26, 626, 767  
    Namen 37  
    Namen-Erweiterung 26, 37  
Datenbank 767  
Datenkapselung 153, 213  
Datenstruktur  
    assoziative 465  
    dynamische 450  
Datentypen 41, 60, 767  
    einfache 60  
    elementar 121  
    ganzzahlige 61  
    generische 367  
Gleitkomma- 64  
komplex 121, 154  
Referenz- 121, 154  
Datumsangaben 437, 440, 444  
DAY\_OF\_MONTH 441  
DAY\_OF\_YEAR 441  
Deadlock 602, 767  
DecimalFormat 434  
**default** 98, 289, 301  
Default-Konstruktor 236  
Default-Methoden 301  
Default-Werte 242  
DeflaterOutputStream 651  
Deklaration 72  
    von Methoden 169  
    von Variablen 41  
Dekrementoperator 83  
Delegation Event Model 540  
delete 412, 627  
DELETE\_ON\_CLOSE 654  
deleteCharAt 412  
deleteIfExists 653  
Deserialisierung 643  
Diamond-Operator 380, 463  
Dienst 664  
directory 26  
dispose 526–528  
DISPOSE\_ON\_CLOSE 526  
distinct 707  
divide 423, 428  
Division 64  
DNS 662, 768  
**do** 108  
DO\_NOTHING\_ON\_CLOSE 526  
Domain Name Service 662  
Domain-Namen 662  
Doppelklicken 768  
Double 414  
**double** 65  
DoubleBuffer 651  
doubleValue 416  
DOWN 429  
Download 768  
drawArc 576  
drawLine 575  
drawOval 575  
drawPolygon 575  
drawPolyline 575

drawRect 575  
drawString 576  
dreistellige Operatoren 53, 75  
Duration 736  
dyadische Operatoren 53, 75  
dynamische Datenstruktur 450  
dynamisches Binden 278

E/A-Ströme 625  
EAST 499  
Editor 29, 768  
effektiv final 699  
einfache Datentypen 60  
Eingabe 625, 629, 640, 650  
Eingabeaufforderung 30  
Eingabegeräte 26  
Eingabestrom 761  
einstellige Operatoren 53, 75  
EJB-Container 738  
**else** 95  
Elternklasse 269  
Enterprise Java Beans 739  
Entscheidungsanweisung 95  
Entwicklungsumgebung 768  
Entwurfsmuster 217  
**enum** 362  
equals 281, 389, 415, 424, 430, 438, 455  
equals-Vertrag 281  
erben 210  
Ereignis 478, 539  
Ereignisempfänger 540  
Ereignisquellen 540  
Ereignisverarbeitung 539, 547  
Ergebnisrückgabe 170  
Ergebnistyp 76, 169  
Error 353  
Error message 332  
erweitern 211  
Erweiterung 209  
    Dateinamen- 26, 37  
Erzeuger/Verbraucher-Problem 604  
Escape-Sequenzen 66  
Ethernet 660  
Event 539  
Event-Dispatching-Thread 621, 727  
EventListener 554  
EventObject 551  
Exception 358

Exception 332, 338  
exists 627  
exit 603  
EXIT\_ON\_CLOSE 481, 526  
exklusives Oder 79  
explizite Typkonvertierung 69  
Exponentenschreibweise 65  
Exponentialschreibweise 65  
**extends** 212, 269  
externer Speicher 26, 768

**false** 67  
Farben 478, 493  
FDDI 660  
Fehler 30  
    fachlicher 756  
    Regression 756  
    semantischer 756  
    syntaktischer 756  
Fehlermeldung 52, 62, 332  
Felder 123, 125  
    flache Kopie 136, 144, 163  
    Index 125  
    Initialisierer 129  
    Komponenten 125  
    Kopie 177  
    Länge 129  
    mehrdimensionale 138, 142  
    Referenzkopie 135, 144, 163  
    Tiefenkopie 136, 144, 163, 164  
    von Feldern 139  
    Zeile 139  
Feldinitialisierer 129  
Feldkomponenten 125  
Feldlänge 129  
Fenster 479, 481  
Fiber Distributed Data Interface 660  
File 626  
**file** 26  
File Transfer Protocol 660  
FileChannel 651  
FileInputStream 641  
FileOutputStream 641  
FileReader 336, 630  
Files 653, 705  
FileWriter 630  
fillArc 576  
fillOval 576

fillPolygon 576  
fillRect 576  
filter 707  
**final** 73, 232, 273, 279  
final, effektiv 699  
**finally** 351  
find 705  
first 456  
flache Kopie 136, 144, 163  
Float 414  
**float** 65  
FloatBuffer 651  
floating point numbers 65  
floatValue 416  
FLOOR 429  
FlowLayout 497  
flush 629, 641  
FocusEvent 552  
focusGained 556  
FocusListener 556  
focusLost 556  
Fokus 505  
folder 26  
Font 495  
Fonts 478, 495  
**for** 106  
forEach 693, 707  
formale Argumente 169  
formale Parameter 169  
Format 444  
Format 434  
format 435, 445  
formatierte Ausgabe 434, 436, 444  
Frame 479  
Freeware 768  
FTP 660  
FULL 447  
Function 691  
Funktion 167  
funktionale Interfaces 689  
  
ganze Zahlen 61  
Ganzzahlen, lange 421  
ganzzahliger Anteil 64  
Garbage Collector 769  
gcd 424  
Generalisierung 208  
generate 705, 706  
  
generische Datentypen 367, 450  
generische Klassen 369  
generische Methoden 377  
gepufferte Ströme 633  
get 441, 458, 466, 652, 692  
get-Methoden 226  
getActionCommand 554  
getBackground 489  
getByName 662  
getClass 282  
getClickCount 554  
getComponents 491  
getContentPane 526–528  
getDateInstance 446  
getDateTimeInstance 447  
getFirstIndex 554  
getFont 489  
getForeground 489  
getHeight 489, 575  
getHostAddress 662  
getHostName 662  
getIcon 502  
getInputStream 666  
getInsets 575  
getInstance 440  
getItem 531, 554  
getItemAt 512  
getItemCount 512, 531  
getKeyStroke 532  
getLastIndex 554  
getLineCount 520  
getLineWrap 520  
getMaxSelectionIndex 514  
getMenu 531  
getMenuCount 531  
getMinSelectionIndex 514  
getName 593, 627  
getOutputStream 666  
getPriority 593, 602  
getSelectedIndex 512  
getSelectedIndices 515  
getSelectedItem 512  
getSelectedText 517  
getSelectedValuesList 515  
getSelectionMode 515  
getSource 551  
getStateChange 554  
getText 502, 517

getThreadGroup 593  
getTime 438, 440  
getTimeInstance 447  
getToolTipText 491  
getWidth 489, 575  
getWindow 554  
getWrapStyleWord 520  
getX 554  
getY 554  
Gibibyte 776  
Gigabyte 776  
GirdPane 731  
Gleitkommadatentypen 64  
Gleitkommazahlen 64, 65  
    lange 425  
gradle 759  
Grafikkoordinaten 574  
grafische Darstellung 573  
grafische Oberfläche 475, 477  
Grammatik 769  
Graphical User Interface 477  
Graphics 575  
GRAY 493  
GREEN 493  
GregorianCalendar 440  
Gregorianischer Kalender 769  
GridLayout 500  
Gruppen, Thread- 603  
GUI 477, 769  
Gültigkeitsbereich 95  
GZIPInputStream 651  
GZIPOutputStream 651

Hacker 769  
HALF\_DOWN 429  
HALF\_EVEN 429  
HALF\_UP 429  
Hardware 26  
Hashcode 282  
hashCode 282, 389  
HashSet 453  
Hashtabellen 282  
hasMoreElements 470  
hasMoreTokens 470  
hasNext 452  
Hauptmethode 41, 56, 186  
Hauptprogramm 186  
Hauptroutine 169

HBox 731  
headSet 456  
heavyweight 485  
hexadezimale Zahlen 62, 769  
HIDE\_ON\_CLOSE 526  
Hilfsklassen 407  
höhere Programmiersprache 28  
HORIZONTAL 533  
HORIZONTAL\_SCROLLBAR\_ALWAYS 522  
HORIZONTAL\_SCROLLBAR\_AS\_NEEDED 522  
HORIZONTAL\_SCROLLBAR\_NEVER 522  
Host 770  
Host-Namen 662  
HOUR\_OF\_DAY 441  
HTML 770  
HTTP 660, 770  
Hüll-Klassen 290, 413  
    Autoboxing 418  
    Autounboxing 418  
    Boxing 418  
    Unboxing 418  
Hypertext Transfer Protocol 660

I/O-Stream 625  
ICANN 662  
Icon 503  
IDE 31, 752, 770  
if 95  
Ikonisieren 480  
ImageIcon 503  
imperative Programmierung 206, 770  
implements 292  
implizite Typkonvertierung 68, 173, 272  
implizite Typumwandlung 68, 173, 272  
import 53  
Index 125  
indexOf 458  
indizierte Variablen 125  
InetAddress 662  
Infix 75  
InflaterInputStream 651  
Informatik 27  
Information Hiding PrincipleKEY 153  
Initialisierer, statische 239  
Initialisierung 73, 235  
Inkrementoperator 83  
innere Klasse 152, 309, 541, 547

input stream 761  
InputStream 626, 640  
InputStreamReader 630  
insert 411  
Insets 575  
**instanceof** 281, 286  
Instant 735  
Instanz 154  
Instanziierung 154, 235  
Instanzmethoden 195, 213, 223, 224  
Instanzvariablen 152  
**int** 61  
IntBuffer 651  
Integer 414  
Integrated Development Environment 31  
Integrierte Entwicklungsumgebung 31, 752  
Interfaces 291, 292  
  funktionale 689  
  versiegelte 390  
Internet 27  
Internetprotokoll 660  
Interpreter 28, 30, 770  
Interpunktionszeichen 51, 52  
interrupt 593, 598  
interrupted 594  
Intranet 27  
IntStream 705  
intValue 416  
invalidate 581  
Invarianz 374  
invokeAndWait 621  
invokeLater 621  
IOTools 639  
IP 660, 661, 771  
IP-Adresse 662, 771  
isAlive 593, 601  
isDaemon 593, 603  
isDirectory 627  
isEditable 512, 517  
isEmpty 451, 466  
isEnabled 489  
isFile 627  
isFocusPainted 505  
isInterrupted 593, 598  
isModal 528  
isOpaque 491  
isSelected 505  
isSelectedIndex 515  
isSelectionEmpty 515  
isVisible 490  
**IT** 771  
ITALIC 495  
ItemEvent 552, 553  
ItemListener 555  
itemStateChanged 555  
Iterable 451  
iterate 705, 706  
Iterator 452  
iterator 451, 452  
Jakarta EE 714, 738  
Jakarta Enterprise Edition 714  
JAR-Datei 771  
Java  
  Bytecode 28, 30  
  Compiler 30  
  Development Kit 30  
  Interpreter 28, 30  
Java EE 714, 738  
Java Enterprise Edition 714, 738  
Java Foundation Classes 478  
  java.awt 478, 492  
  java.awt.event 532, 533, 551  
  java.io 336, 626, 635, 650  
  java.lang 192, 305, 407, 410  
  java.math 421  
  java.net 659, 662, 665  
  java.nio 651  
  java.nio.file 651  
  java.text 434, 444  
  java.util 437, 447, 449, 460, 462, 469, 551  
  java.util.functions 691  
  java.util.stream 700  
  javadoc 48, 751  
JavaFX 475, 714, 725  
  Application-Thread 727  
  Container 731  
  Data Binding 734  
  Scene Builder 734  
JavaScript 715  
  javax.swing 478, 491, 532  
  javax.swing.event 551  
  javax.swing.text 517  
  JButton 505  
  JCheckBox 508  
  JComboBox 511

- JComponent 491  
JDialog 527  
JDK 30, **771**  
JEE 714  
JFC **478**  
JFrame 481, 526  
Jigsaw 714  
JLabel 483, 503  
JList 514  
JMenuBar 530  
join 610  
 JPanel 524  
JPasswordField 517  
JRadioButton 509  
JScrollPane 522  
JShell 714, **715**  
JTextArea 520  
JTextComponent 517  
JTextField 517  
JToggleButton 507  
JToolBar 533  
JWindow 527
- Kapselung** 153, **213**  
KeyEvent 552  
KeyListener 556  
keyPressed 556  
keyReleased 556  
keySet 466  
keyTyped 556  
Kibibyte 78, **776**  
Kilobyte 78, **776**  
Kindklassen 269  
Klapptafeln 511  
Klasse 40  
**Klassen** 150, **151**, 205  
  abstrakte 268, **291**, 292, 294  
  Adapter- 558  
  anonyme 315, 547  
  Attribute 152  
  Diagramm 153, 216  
  Eltern- 269  
  Exception- 332  
  generische 369  
  Hüll- 290, 413  
  innere 152, 309, 547  
  Instanz 154  
  Instanziierung 154  
  Instanzvariablen 152  
  Kapselung 153  
  Kind- 269  
  Komponentenvariablen 152  
  Methoden 152, 191, 230  
  Namen 40  
  Referenz 157  
  Sub- 209, 267, 269  
  Super- 209, 269  
  Variablen 152, 230  
  versiegelte 390  
  Wrapper- 290, 413  
Klassendiagramm 153, **216**  
Klassenkonstanten 232  
Klassenmethoden 191, 230  
Klassenvariablen 152, 230  
Klicken 771  
Knöpfe 505, 507–509  
Kommandozeile 30  
Kommandozeilenparameter 187  
Kommentare 47, 51  
  mit javadoc 48  
Kommunikation, Thread- 603  
Kompatibilität 771  
Komponenten 477, 478  
  grafische Darstellung 573  
  statische 230  
Komponentenvariablen 152  
Komposition 215  
Konkatenation, String- 408  
Konsole 57  
Konsolenfenster 30, **772**  
Konstanten  
  Klassen- 232  
  Literal- 50, 62, 63  
  symbolische 73, 232  
  Zeichenketten- 407  
konstanter Ausdruck 78  
Konstruktoren 235  
  Default- 236  
  Standard- 236  
  Überladen 237, 238  
Koordinatensystem 574  
Kopie  
  Feld- 177  
  flache 136, 144, 163  
  mit clone 136  
  Referenz- 176

Tiefen- 136, 144, 163, 164  
Kovarianz 374

Labels 483, 503  
Lambda-Ausdrücke 683, 687  
Länge eines Feldes 129  
Langzahlen 421, 425  
last 456  
lastIndexOf 458  
Laufzeitfehler 86  
launch 727  
Layout 485  
Layout-Manager 478, 496  
LayoutManager 496  
Lebenszyklus, Thread- 600  
Leere Anweisung 94  
Leerzeichen 51  
LEFT 497, 502, 518  
leichtgewichtige Prozesse 591  
length 412, 627  
Leser/Schreiber-Problem 603  
LIGHT\_GRAY 493  
lightweight 485  
limit 707  
lines 705  
LinkedList 458  
List 449, 458  
list 627, 705  
Liste 457  
Listener 478, 554  
    Registrierung 560  
ListSelectionEvent 553, 555  
ListSelectionListener 555  
ListSelectionModel 515  
Literale 50  
Literalkonstanten 50, 62, 63, 65–67  
    null 50  
LocalDate 735  
LocalDateTime 736  
Locale 447  
LocalTime 735  
logische Aussagen 67  
logische Operatoren 81  
    Oder 79, 81, 359  
    Und 79, 81  
LONG 447  
Long 414  
long 61

LongStream 706  
longValue 416  
Look and Feel 562, 772

MAGENTA 493  
main 41, 56, 169, 186  
make 759  
Map 465  
map 707  
Marke 110  
markierte Anweisungen 110  
Maschinensprache 27, 772  
Math 192  
maven 759  
max 424, 430  
max 192  
MAX\_PRIORITY 602  
MAX\_VALUE 416  
Maximieren 480  
Mebibyte 776  
MEDIUM 447  
Megabyte 776  
Mehrdimensionale Felder 138, 142  
Mehrfachvererbung 293  
mehrzeiliger Textblock 67  
Menü 531  
Menüleisten 530  
Menge 453  
menuCanceled 557  
menuDeselected 557  
MenuEvent 552, 553  
MenuListener 557  
menuSelected 557  
Message 332  
META\_MASK 533  
Methoden 41, 43, 152, 167, 168  
    Argument 169  
    Aufruf 43, 172  
    Call by value 172  
    Default- 301  
    Deklaration 169  
    dynamisches Binden 278  
    Ergebnisrückgabe 170  
    generische 377  
    get- 226  
    Instanz- 195, 213, 223, 224  
    Klassen- 191, 230  
    Kopf 169

- Name 169  
Parameter 43, 169  
Referenzen 695  
rekursive 181  
Rückgabetyp 170  
Rumpf 169  
set- 226  
statische 230, 301  
synchronisierte 606  
terminieren 183  
Überladen 174  
Überschreiben 214, 276  
variable Argumente 175, 463  
Wertaufruf 172  
MILLISECOND 441  
min 424, 430  
min 192  
MIN\_PRIORITY 602  
MIN\_VALUE 416  
Minimieren 480  
MINUTE 441  
mkdir 627  
modal 527  
Modell 205  
Modellierung 29, 205  
Modellierungsphase 216  
Modifikatoren 308  
Modul 719  
Modulsystem 714  
monadische Operatoren 53, 75  
Monitor 607  
MONTH 441  
mouseClicked 555  
mouseDragged 556  
mouseEntered 555  
MouseEvent 552  
mouseExited 555  
MouseListener 555  
MouseMotionListener 556  
mouseMoved 556  
mousePressed 555  
mouseReleased 555  
move 653  
multiply 423, 428
- Namen 49  
Datei- 37  
für Threads 600
- Klassen- 40  
Methoden- 169  
NaN 416  
negate 423, 428  
Negation 78  
NEGATIVE\_INFINITY 416  
net 27  
Netz 27  
Netzwerk 659  
Netzwerkprogrammierung 659  
Netzwerkschicht 660  
new 127, 154  
newInputStream 654  
newLine 634  
newOutputStream 654  
newPriority 593  
next 452  
nextElement 470  
nextToken 470, 635  
nextTokens 470  
NOFOLLOW\_LINKS 653  
NORM\_PRIORITY 602  
NORTH 499  
Notation 75  
    Infix 75  
    Postfix 75  
    Präfix 75  
notify 601, 610  
notifyAll 601, 610  
Null-Literal 50  
Null-Referenz 161  
Nullstellen 431  
NumberFormat 434
- Oberfläche, grafische 475, 477  
Object 610  
Object 279  
ObjectInputStream 643  
ObjectOutputStream 643  
Objekt 154  
    erzeugen 154  
Objekte 205, 772  
objektorientierte Programmierung 208, 773  
Oder  
    bedingtes logisches 81  
    exklusives 79  
    logisches 79, 81, 359  
of 705

- oktale Zahlen 62, 773  
OOP 773  
Open Source 773  
Operand 75  
Operationen  
    Abschluss- 704, 707, 708  
    Pipeline- 700, 703  
    Stream- 703  
    Zwischen- 703, 706  
Operator  
    . 155  
    Zugriffs- 155  
Operatoren 51, 53, 74  
    abkürzende Schreibweise 81  
    arithmetische 76  
    Auswertungsreihenfolge 84  
    binäre 53, 75  
    Bit- 78  
    Dekrement- 83  
    Diamond 380, 463  
    dreistellige 53, 75  
    dyadische 53, 75  
    einstellige 53, 75  
    Inkrement- 83  
    logische 81  
    monadische 53, 75  
    Notation 75  
    Prioritäten 84  
    Reihenfolge 75  
    Schiebe- 79  
    ternäre 53, 75  
    triadische 53, 75  
    unäre 53, 75  
    Vergleichs- 81  
    Zuweisungs- 72, 80  
    zweistellige 53, 75  
Operatorsymbole 53  
ORANGE 493  
Ordner 26  
OutOfMemoryError 354  
OutputStream 626, 641  
OutputStreamWriter 630  
Override-Annotation 283
- pack 526, 527, 529  
paint 574  
paintBorder 574  
paintChildren 574
- paintComponent 574  
Paket  
    anonymes 307  
    default 307  
    unbenanntes 307  
Pakete 305  
    java.awt 478, 492  
    java.awt.event 532, 533, 551  
    java.io 336, 626, 635, 650  
    java.lang 192, 305, 407, 410  
    java.math 421  
    java.net 659, 662, 665  
    java.nio 651  
    java.nio.file 651  
    java.text 434, 444  
    java.util 437, 447, 449, 460, 462, 469,  
        551  
    java.util.functions 691  
    java.util.stream 700  
    javax.swing 478, 491, 532  
    javax.swing.event 551  
    javax.swing.text 517  
    Prog1Tools 305, 762  
Paradigmen 206  
parallele Programmierung 589  
parallelStream 705  
Parameter  
    aktueller 172  
    formaler 169  
    Kommandozeilen- 187  
Parameterliste 169, 170  
parse 447  
parseByte 416  
parseDouble 416  
parseFloat 416  
parseInteger 416  
parseLong 416  
parseShort 416  
paste 517  
Path 652  
Paths 652  
Pattern-Matching 286–288  
Peer 485  
Period 736  
Peripheriegeräte 26  
Philosophenproblem 613  
physikalische Schicht 660  
PI 233

- PINK 493  
Pipeline-Operationen 700, 703  
PLAIN 495  
Polymorphie 214, 267, 272  
Popup-Menü 533  
Port 663  
Portabilität 774  
POSITIVE\_INFINITY 416  
Postfix 75  
pow 192, 423  
präemptives Scheduling 602  
Präfix 75  
Predicate 691  
print 631, 637  
printf 436  
println 631, 637  
PrintStream 647  
PrintWriter 637  
Prioritäten 53  
    der Operatoren 84  
    von Threads 602  
Problemanalyse 29  
problemorientierte Programmiersprache 28  
Prog1Tools 305, 762  
Programm 26  
Programmieren 29  
Programmiersprache 27  
Programmierung  
    Client/Server- 659  
    imperative 206  
    Netzwerk- 659  
    objektorientierte 208  
    parallele 589  
Prompt 88, 763  
**protected** 308  
Protokoll 660, 774  
Prozesse, leichtgewichtige 591  
Prozessor 26, 774  
**public** 153  
Pulldown-Menü 531  
Punktoperator 155  
put 466  
putAll 466  
Python 715
- Quellcode 28, 774  
Quellprogramm 28
- Quelltext 28, 774  
Quicksort 183
- Rahmen 479, 481  
RAM 26, 774  
RandomAccessFile 650  
range 705, 706  
rangeClosed 706  
read 336, 629, 630, 640, 641  
readBoolean 642  
readByte 642  
readChar 642, 762  
readDouble 642, 762  
Reader 626, 629  
readFloat 642  
readInt 642, 762  
readInteger 762  
readLine 633  
readLong 642  
readObject 643  
readShort 642  
Record-Patterns 406  
RED 493  
reduce 707  
Referenz 122, 134, 157  
    Null- 161  
Referenzdatentypen 121, 133, 154, 176  
Referenzen  
    Methoden- 695  
Referenzkopie 135, 144, 163, 176  
Regel, Syntax- 45, 46  
Registrierung eines Listeners 560  
Rekursion 181  
    Nachteile 183  
    Vorteile 182  
rekursive Methoden 181  
remainder 423  
remove 451, 452, 458, 466, 491, 510, 531  
removeAll 451, 531  
removeAllItems 512  
removeItem 512  
removeItemAt 512  
renameTo 627  
repaint 573  
Repaint-Manager 573  
replace 412  
REPLACE\_EXISTING 653, 654  
replaceAll 693

Reservierte Wörter 50  
Rest 64  
Resultatstyp 169  
retainAll 451  
**return** 112, 170  
revalidate 581  
RGB-Farbmodell 493, 774  
RIGHT 497, 502, 518  
ROM 26  
round 192  
Routinen 168  
Routing 661  
RTFM 775  
Rückgabetyp 169, 170  
Rumpf  
    einer Methode 169  
    einer Schleife 106  
run 591–593  
Rundungsfehler 65  
Runnable 592, 596  
RuntimeException 338

Scanner 639  
Schaltflächen 505, 507–509  
Scheduler 601, 602, 775  
Scheduling bei Threads 602  
Schiebeoperatoren 79  
Schleifen 105, 107, 145  
    abweisende 107  
    **do** 108  
    Endlos- 109  
    **for** 106  
    nicht-abweisende 108  
    Rumpf 106  
    unendliche 109  
    vereinfachte Notation 188, 451, 453  
    **while** 107  
Schließen 480  
Schlüsselwörter 50  
    **abstract** 268  
    **assert** 356  
    **catch** 335  
    **class** 40  
    **default** 98, 289, 301  
    **enum** 362  
    **extends** 212, 269  
    **final** 73, 232, 273, 279  
    **finally** 351  
    **implements** 292  
    **protected** 308  
    **public** 153  
    **static** 231  
    **super** 276  
    **synchronized** 606  
    **this** 225  
    **catch** 334  
    **throw** 334, 342  
    **throws** 337  
    **transient** 644  
    **try** 334  
    **var** 74  
Schnittstellen 213, 291  
SECOND 441  
Seiteneffekt 177  
Semantik 775  
semantische Ereignisse 552  
Semikolon 52  
Sequenzdiagramm 217  
Serialisierung 643  
Serializable 643  
Server 664, 775  
Server-Host 664  
Server-Rechner 664  
ServerSocket 665  
Set 449, 453  
set 440, 441, 458  
set-Methoden 226  
setAccelerator 532  
setActionCommand 546  
setBackground 490  
setCharAt 412  
setDaemon 593  
setDefaultCloseOperation 481, 526, 528  
setEditable 512, 517  
setEnabled 490  
setFocusPainted 505  
setFont 490  
setForeground 490  
setHorizontalAlignment 502, 518  
setHorizontalScrollBarPolicy 522  
setHorizontalTextPosition 502  
setIcon 502  
setJMenuBar 526, 529  
setLayout 485, 491  
setLineWrap 520

setLocation 490  
setMnemonic 532  
setModal 528  
setName 593  
setOpaque 491  
setPriority 602  
setSelected 505  
setSelectedIndex 512, 515  
setSelectedIndices 515  
setSelectedItem 512  
setSelectionMode 515  
setSize 479, 481, 490  
setSoTimeout 676  
setText 502, 517  
setTime 438, 440  
setTimeInMillis 442  
setTitle 479, 481, 526, 528  
setToolTipText 491  
setVerticalAlignment 502  
setVerticalScrollBarPolicy 522  
setVerticalTextPosition 502  
setVisible 479, 481, 490  
setWrapStyleWord 520  
Shell 30  
SHIFT\_MASK 533  
SHORT 447  
Short 414  
**short** 61  
shortValue 416  
showConfirmDialog 530  
showInputDialog 530  
showMessageDialog 530  
Sichtbarkeit 178  
Simple Mail Transfer Protocol 660  
SimpleDateFormat 444  
SINGLE\_SELECTION 515  
size 451, 466  
Skript 715  
Skriptsprache 715  
sleep 594, 601  
SMTP 660  
Socket 664  
Socket 665, 676  
SocketChannel 651  
Software 26  
sort 461, 463, 693  
sorted 707  
SortedMap 468  
SortedSet 455  
Sortieren 460  
Sourcecode 775  
SOUTH 499  
Speicher, externer 26  
Speicherkapazität 775  
Speicherzelle 26  
Sperre 607  
Spezialisierung 209  
split 472  
**Sprungbefehle** 110  
sqrt 192  
Stage 727  
Standard-Konstruktor 236  
Standardausgabe 433  
Standardwerte 242  
start 590–592, 600  
Starvation 602  
stateChanged 555  
**static** 231  
**static-Block** 239  
statische Importe 89, 194  
statische Initialisierer 239  
statische Komponenten 230  
statische Methoden 230, 301  
Ströme  
    gepufferte 633  
    Stream 700, 705–707  
    stream 705  
    Stream-Operationen 703  
    Streams 700, 703  
    StreamTokenizer 635  
String 186, 196, 407  
    Addition 76, 77  
    Konkatenation 76, 77  
StringBuffer 410  
StringTokenizer 469  
Stylesheet 731  
Subklassen 209, 267, 269  
subSet 456  
subtract 423, 428  
Suchen 460  
sum 708  
**super** 276  
Superklassen 209, 269  
Supplier 692  
Swing 475, 481, 485, 582  
SwingUtilities 563, 621

**switch** 98, 102  
Switch-Ausdrücke 104  
symbolische Konstanten 73, 232  
Synchronisation, Thread- 603  
Synchronisieren 776  
synchronisierte Methoden 606  
**synchronized** 606  
Syntax 45, 46, 776  
    Regel 45, 46  
    Variable 46  
System 631  
Systemsoftware 26  
Szenengraph 730

Tabulatorzeichen 51  
**tailSet** 456  
Targets 596  
Tastatureingaben 761  
Tastaturfokus 505  
Tastaturkommandos 480  
TCP 660, 661, 776  
Telnet-Programm 669  
Terabyte 776  
Terminal 30  
terminieren 183  
ternäre Operatoren 53, 75  
**test** 691  
Textblock 67  
Texteditor 29, 30  
Textkomponenten 517, 520  
**this** 225  
Thread 592, 674  
Threads 589, 776  
    Dämon- 602  
    Deadlock 602  
    Frames 615  
    Gruppen 603  
    Kommunikation 603  
    Lebenszyklus 600  
    Namen 600  
    Scheduling 602  
    Sicherheit 621  
    Starvation 602  
    Swing 615  
    Synchronisation 603  
    vorzeitig beenden 598  
**throw** 334, 342  
Throwable 354

**throws** 337  
Tibibyte 776  
Tiefenkopie 136, 144, 163, 164  
Timeline 734  
**toArray** 451  
Token 469  
Toolbars 533  
Tooltip 492, 777  
TOP 502  
Top-Level-Container 479, 481  
**toString** 229, 280, 411, 424, 430, 434, 438  
**transient** 644  
Transmission Control Protocol 660  
Transportschicht 660  
**TreeSet** 456  
Trennzeichen 51  
triadische Operatoren 53, 75  
**true** 67  
**TRUNCATE\_EXISTING** 654  
**try** 334  
    mit Ressourcen 360, 648  
**TT\_EOF** 635  
**TT\_EOL** 635  
**TT\_NUMBER** 635  
**TT\_WORD** 635  
Typ 76  
    Daten- 41  
    Ergebnis- 76  
    Rückgabe- 169, 170  
Typ-Parameter 367, 371, 380  
Typecast 68  
Typkonvertierung 68  
    automatische 68, 173, 272  
    explizite 69  
    implizite 68, 173, 272  
Typsicherheit bei Collections 450  
Typumwandlung 68  
    automatische 68, 173, 272  
    implizite 68, 173, 272  
Typ-Variablen 369  
Typwandlung bei Wrapper-Klassen 418

Überladen 174  
    von Konstruktoren 237, 238  
    von Methoden 174  
Überschreiben von Methoden 214, 276  
Übersetzer 28, 777  
UDP 660, 661, 777

- UIManager 563  
Umgebungsvariable 777  
UML 216, 777  
Umlaute 49  
unäre Operatoren 53, 75  
UnaryOperator 692  
unbenannte Variablen 360  
Unboxing 418  
Und  
    bedingtes logisches 81  
    logisches 79, 81  
unendliche Streams 705, 706  
Unicode 66, 778  
    Schreibweise 66  
Unified Modeling Language 216, 778  
Unit-Test 757  
UNNECESSARY 429  
Unterprogramm 167  
Unterstrich 49, 63  
UP 429  
Update 778  
URL 778  
use cases 216  
Use-Case-Diagramm 217  
User Datagram Protocol 660
- validate 581  
values 466  
**var** 74  
variable Methodenargumente 463  
Variable, Syntax- 46  
Variablen 41, 70  
    Deklaration 41, 72  
    Gültigkeitsbereich 95  
    indizierte 125  
    Initialisierung 73  
    Instanz- 152  
    Klassen- 152, 230  
    lokale 169  
    Name 72  
    unbenannte 360  
Variablen Deklaration 41  
VBox 729, 731  
Verdecken 178  
vereinfachte Eingabe 639  
Vererbung 210, 267, 271  
Vergleichsoperatoren 81  
Verkettung, String- 408
- vernetzt 27  
versiegelte Interfaces 390  
versiegelte Klassen 390  
Versionsverwaltung 754  
Verteilungsdiagramm 217  
VERTICAL 533  
VERTICAL\_SCROLLBAR\_ALWAYS 522  
VERTICAL\_SCROLLBAR\_AS\_NEEDED 522  
VERTICAL\_SCROLLBAR\_NEVER 522  
Verzeichnis 26, 626  
VirtualMachineError 354  
virtuelle Maschine 29  
**void** 169, 170, 186  
Vorzeichen 61
- Wahrheitswert 67  
wait 601, 610  
walk 705  
web 27  
Webcontainer 738  
Werkzeugleisten 533  
Wertaufruf 172  
Wertebereich 60  
WEST 499  
**while** 107  
WHITE 493  
Wiederholungsanweisungen 105  
Wildcards 373  
    Bounded 375  
windowActivated 557  
WindowAdapter 559  
windowClosed 557  
windowClosing 557  
windowDeactivated 557  
windowDeiconified 557  
WindowEvent 553  
WindowFocusListener 557  
windowGainedFocus 557  
windowIconified 557  
WindowListener 557  
windowLostFocus 557  
windowOpened 557  
windowStateChanged 558  
WindowStateListener 557  
Workaround 778  
Wortschatz 778  
Wortsymbole 50  
Wrapper-Klassen 290, 413

Autoboxing 418  
Autounboxing 418  
Boxing 418  
Typwandlung 418  
Unboxing 418  
WRITE 654  
write 629, 630, 641  
writeBoolean 642  
writeByte 642  
writeChar 642  
writeDouble 642  
writeFloat 642  
writeInt 642  
writeLong 642  
writeObject 643  
Writer 626, 629  
writeShort 642  
  
XML 778  
  
YEAR 441  
YELLOW 493  
**yield** 593, 601  
  
Zahlen  
  binäre 62  
  ganze 61  
  Gleitkomma- 65  
  hexadezimale 62  
  negative 61  
  oktale 62  
  Zeichen 66  
  Zeichenkettenliterale 407  
  Zeichenströme 626  
  Zeichnen 573  
  Zeile 139  
  Zeilenendezeichen 51  
  Zeitangaben 437, 440, 444  
  Zeitpunkte 438  
  Zeitscheibenverfahren 602  
  Zentraleinheit 26  
  Zielprogramm 28  
  Ziffern 49  
  ZipInputStream 651  
  ZipOutputStream 651  
  ZonedDateTime 736  
  Zugriffsmethoden 223  
  Zugriffsrechte 153, 223, 308  
  Zusicherungen 356  
  Zuweisung 42, 72  
  zuweisungskompatibel 171  
  Zuweisungsoperator 72, 80  
  Zweierkomplement 61  
  zweistellige Operatoren 53, 75  
  Zwischenoperationen 703, 706