



Inhaltsverzeichnis

Dietmar Ratz, Jens Scheffler, Detlef Seese, Jan Wiesenberger

Grundkurs Programmieren in Java

ISBN (Buch): 978-3-446-44073-9

ISBN (E-Book): 978-3-446-44110-1

Weitere Informationen oder Bestellungen unter

<http://www.hanser-fachbuch.de/978-3-446-44073-9>

sowie im Buchhandel.

Inhaltsverzeichnis

Vorwort	17
1 Einleitung	19
1.1 Java – mehr als nur kalter Kaffee?	19
1.2 Java für Anfänger – das Konzept dieses Buches	20
1.3 Zusatzmaterial und Kontakt zu den Autoren	22
1.4 Verwendete Schreibweisen	22
2 Einige Grundbegriffe aus der Welt des Programmierens	23
2.1 Computer, Software, Informatik und das Internet	23
2.2 Was heißt Programmieren?	26
I Einstieg in das Programmieren in Java	29
3 Aller Anfang ist schwer	31
3.1 Mein erstes Programm	31
3.2 Formeln, Ausdrücke und Anweisungen	32
3.3 Zahlenbeispiele	33
3.4 Verwendung von Variablen	34
3.5 „Auf den Schirm!“	34
3.6 Das Programmgerüst	35
3.7 Eingeben, übersetzen und ausführen	37
3.8 Übungsaufgaben	38
4 Grundlagen der Programmierung in Java	39
4.1 Grundelemente eines Java-Programms	39
4.1.1 Kommentare	41
4.1.2 Bezeichner und Namen	43
4.1.3 Literale	44
4.1.4 Reservierte Wörter, Schlüsselwörter	44

4.1.5	Trennzeichen	45
4.1.6	Interpunktionszeichen	46
4.1.7	Operatorsymbole	46
4.1.8	import -Anweisungen	47
4.1.9	Zusammenfassung	48
4.1.10	Übungsaufgaben	48
4.2	Erste Schritte in Java	49
4.2.1	Grundstruktur eines Java-Programms	50
4.2.2	Ausgaben auf der Konsole	51
4.2.3	Eingaben von der Konsole	52
4.2.4	Schöner programmieren in Java	53
4.2.5	Zusammenfassung	54
4.2.6	Übungsaufgaben	54
4.3	Einfache Datentypen	55
4.3.1	Ganzzahlige Datentypen	55
4.3.2	Gleitkommatypen	57
4.3.3	Der Datentyp char für Zeichen	59
4.3.4	Zeichenketten	60
4.3.5	Der Datentyp boolean für Wahrheitswerte	60
4.3.6	Implizite und explizite Typumwandlungen	60
4.3.7	Zusammenfassung	62
4.3.8	Übungsaufgaben	62
4.4	Der Umgang mit einfachen Datentypen	63
4.4.1	Variablen	63
4.4.2	Operatoren und Ausdrücke	67
4.4.2.1	Arithmetische Operatoren	68
4.4.2.2	Bitoperatoren	70
4.4.2.3	Zuweisungsoperator	72
4.4.2.4	Vergleichsoperatoren und logische Operatoren	73
4.4.2.5	Inkrement- und Dekrementoperatoren	75
4.4.2.6	Priorität und Auswertungsreihenfolge der Operatoren	76
4.4.3	Allgemeine Ausdrücke	77
4.4.4	Ein- und Ausgabe	78
4.4.4.1	Statischer Import der IOTools-Methoden	79
4.4.5	Zusammenfassung	81
4.4.6	Übungsaufgaben	81
4.5	Anweisungen und Ablaufsteuerung	84
4.5.1	Anweisungen	85
4.5.2	Blöcke und ihre Struktur	85
4.5.3	Entscheidungsanweisung	86
4.5.3.1	Die if -Anweisung	86

4.5.3.2	Die switch -Anweisung	87
4.5.4	Wiederholungsanweisungen, Schleifen	89
4.5.4.1	Die for -Anweisung	89
4.5.4.2	Vereinfachte for -Schleifen-Notation	90
4.5.4.3	Die while -Anweisung	91
4.5.4.4	Die do -Anweisung	91
4.5.4.5	Endlosschleifen	92
4.5.5	Sprungbefehle und markierte Anweisungen	93
4.5.6	Zusammenfassung	95
4.5.7	Übungsaufgaben	95
5	Referenzdatentypen	105
5.1	Felder	107
5.1.1	Was sind Felder?	109
5.1.2	Deklaration, Erzeugung und Initialisierung von Feldern	110
5.1.3	Felder unbekannter Länge	113
5.1.4	Referenzen	115
5.1.5	Ein besserer Terminkalender	119
5.1.6	Mehrdimensionale Felder	121
5.1.7	Mehrdimensionale Felder unterschiedlicher Länge	124
5.1.8	Vorsicht, Falle: Kopieren von mehrdimensionalen Feldern	126
5.1.9	Vereinfachte for -Schleifen-Notation	127
5.1.10	Zusammenfassung	129
5.1.11	Übungsaufgaben	129
5.2	Klassen	132
5.2.1	Was sind Klassen?	133
5.2.2	Deklaration und Instantiierung von Klassen	134
5.2.3	Komponentenzugriff bei Objekten	135
5.2.4	Ein erstes Adressbuch	136
5.2.5	Klassen als Referenzdatentyp	138
5.2.6	Felder von Klassen	141
5.2.7	Vorsicht, Falle: Kopieren von geschachtelten Referenzdatentypen	144
5.2.8	Auslagern von Klassen	145
5.2.9	Zusammenfassung	147
5.2.10	Übungsaufgaben	147
6	Methoden, Unterprogramme	149
6.1	Methoden	150
6.1.1	Was sind Methoden?	150
6.1.2	Deklaration von Methoden	151
6.1.3	Parameterübergabe und Ergebnisrückgabe	152

6.1.4	Aufruf von Methoden	154
6.1.5	Überladen von Methoden	155
6.1.6	Variable Argument-Anzahl bei Methoden	157
6.1.7	Vorsicht, Falle: Referenzen als Parameter	158
6.1.8	Sichtbarkeit und Verdecken von Variablen	160
6.1.9	Zusammenfassung	162
6.1.10	Übungsaufgaben	162
6.2	Rekursiv definierte Methoden	163
6.2.1	Motivation	163
6.2.2	Gute und schlechte Beispiele für rekursive Methoden	165
6.2.3	Zusammenfassung	168
6.3	Die Methode <code>main</code>	168
6.3.1	Kommandozeilenparameter	169
6.3.2	Anwendung der vereinfachten <code>for</code> -Schleifen-Notation . . .	170
6.3.3	Zusammenfassung	171
6.3.4	Übungsaufgaben	171
6.4	Methoden aus anderen Klassen aufrufen	173
6.4.1	Klassenmethoden	173
6.4.2	Die Methoden der Klasse <code>java.lang.Math</code>	174
6.4.3	Statischer Import	175
6.5	Methoden von Objekten aufrufen	176
6.5.1	Instanzmethoden	176
6.5.2	Die Methoden der Klasse <code>java.lang.String</code>	177
6.6	Übungsaufgaben	180
 II Objektorientiertes Programmieren in Java		185
7	Die objektorientierte Philosophie	187
7.1	Die Welt, in der wir leben	187
7.2	Programmierparadigmen – Objektorientierung im Vergleich	188
7.3	Die vier Grundpfeiler objektorientierter Programmierung	190
7.3.1	Generalisierung	190
7.3.2	Vererbung	192
7.3.3	Kapselung	195
7.3.4	Polymorphismus	196
7.3.5	Weitere wichtige Grundbegriffe	197
7.4	Modellbildung – von der realen Welt in den Computer	198
7.4.1	Grafisches Modellieren mit UML	198
7.4.2	Entwurfsmuster	199
7.5	Zusammenfassung	200
7.6	Übungsaufgaben	201

8	Der grundlegende Umgang mit Klassen	203
8.1	Vom Referenzdatentyp zur Objektorientierung	203
8.2	Instanzmethoden	205
8.2.1	Zugriffsrechte	205
8.2.2	Was sind Instanzmethoden?	206
8.2.3	Instanzmethoden zur Validierung von Eingaben	209
8.2.4	Instanzmethoden als erweiterte Funktionalität	210
8.3	Statische Komponenten einer Klasse	211
8.3.1	Klassenvariablen und -methoden	212
8.3.2	Klassenkonstanten	214
8.4	Instantiierung und Initialisierung	217
8.4.1	Konstruktoren	217
8.4.2	Überladen von Konstruktoren	219
8.4.3	Der statische Initialisierer	221
8.4.4	Der Mechanismus der Objekterzeugung	224
8.5	Zusammenfassung	229
8.6	Übungsaufgaben	229
9	Vererbung und Polymorphismus	249
9.1	Wozu braucht man Vererbung?	249
9.1.1	Aufgabenstellung	249
9.1.2	Analyse des Problems	250
9.1.3	Ein erster Ansatz	250
9.1.4	Eine Klasse für sich	251
9.1.5	Stärken der Vererbung	252
9.1.6	Vererbung verhindern durch final	255
9.1.7	Übungsaufgaben	256
9.2	Die super -Referenz	257
9.3	Überschreiben von Methoden und Variablen	259
9.3.1	Dynamisches Binden	259
9.3.2	Überschreiben von Methoden verhindern durch final	261
9.4	Die Klasse <code>java.lang.Object</code>	262
9.5	Übungsaufgaben	265
9.6	Abstrakte Klassen und Interfaces	266
9.7	Übungsaufgaben	269
9.8	Weiteres zum Thema Objektorientierung	274
9.8.1	Erstellen von Paketen	274
9.8.2	Zugriffsrechte	276
9.8.3	Innere Klassen	277
9.8.4	Anonyme Klassen	282
9.9	Zusammenfassung	284
9.10	Übungsaufgaben	284

10 Exceptions und Errors	295
10.1 Eine Einführung in Exceptions	296
10.1.1 Was ist eine Exception?	296
10.1.2 Übungsaufgaben	298
10.1.3 Abfangen von Exceptions	298
10.1.4 Ein Anwendungsbeispiel	299
10.1.5 Die <code>RuntimeException</code>	302
10.1.6 Übungsaufgaben	303
10.2 Exceptions für Fortgeschrittene	305
10.2.1 Definieren eigener Exceptions	305
10.2.2 Übungsaufgaben	307
10.2.3 Vererbung und Exceptions	307
10.2.4 Vorsicht, Falle!	311
10.2.5 Der finally -Block	313
10.2.6 Die Klassen <code>Throwable</code> und <code>Error</code>	317
10.2.7 Zusammenfassung	319
10.2.8 Übungsaufgaben	319
10.3 Assertions	320
10.3.1 Zusicherungen im Programmcode	320
10.3.2 Compilieren des Programmcodes	321
10.3.3 Ausführen des Programmcodes	322
10.3.4 Zusammenfassung	322
11 Fortgeschrittene objektorientierte Programmierung	323
11.1 Aufzählungstypen	324
11.1.1 Deklaration eines Aufzählungstyps	324
11.1.2 Instanzmethoden der enum -Objekte	325
11.1.3 Selbstdefinierte Instanzmethoden für enum -Objekte	325
11.1.4 Übungsaufgaben	327
11.2 Generische Datentypen	329
11.2.1 Generizität in alten Java-Versionen	329
11.2.2 Generizität ab Java 5.0	332
11.2.3 Einschränkungen der Typ-Parameter	334
11.2.4 Wildcards	336
11.2.5 Bounded Wildcards	337
11.2.6 Generische Methoden	339
11.2.7 Ausblick	341
11.2.8 Übungsaufgaben	341
11.3 Sortieren von Feldern und das Interface <code>Comparable</code>	346
12 Einige wichtige Hilfsklassen	349
12.1 Die Klasse <code>StringBuffer</code>	349

12.1.1	Arbeiten mit <code>String</code> -Objekten	349
12.1.2	Arbeiten mit <code>StringBuffer</code> -Objekten	352
12.1.3	Übungsaufgaben	354
12.2	Die Wrapper-Klassen (Hüll-Klassen)	355
12.2.1	Arbeiten mit „eingepackten“ Daten	355
12.2.2	Aufbau der Wrapper-Klassen	356
12.2.3	Ein Anwendungsbeispiel	359
12.2.4	Automatische Typwandlung für die Wrapper-Klassen . . .	360
12.2.5	Übungsaufgaben	362
12.3	Die Klassen <code>BigInteger</code> und <code>BigDecimal</code>	363
12.3.1	Arbeiten mit langen Ganzzahlen	363
12.3.2	Aufbau der Klasse <code>BigInteger</code>	365
12.3.3	Übungsaufgaben	367
12.3.4	Arbeiten mit langen Gleitkommazahlen	367
12.3.5	Aufbau der Klasse <code>BigDecimal</code>	370
12.3.6	Viele Stellen von Nullstellen gefällig?	373
12.3.7	Übungsaufgaben	374
12.4	Die Klasse <code>DecimalFormat</code>	375
12.4.1	Standard-Ausgaben in Java	375
12.4.2	Arbeiten mit <code>Format</code> -Objekten	376
12.4.3	Vereinfachte formatierte Ausgabe	378
12.4.4	Übungsaufgaben	379
12.5	Die Klassen <code>Date</code> und <code>Calendar</code>	379
12.5.1	Arbeiten mit „Zeitpunkten“	380
12.5.2	Auf die Plätze, fertig, los!	381
12.5.3	Spezielle <code>Calendar</code> -Klassen	382
12.5.4	Noch einmal: Zeitmessung	384
12.5.5	Übungsaufgaben	386
12.6	Die Klassen <code>SimpleDateFormat</code> und <code>DateFormat</code>	386
12.6.1	Arbeiten mit <code>Format</code> -Objekten für Datum/Zeit-Angaben . .	386
12.6.2	Übungsaufgaben	391
12.7	Die <code>Collection</code> -Klassen	391
12.7.1	„Sammlungen“ von Objekten – der Aufbau des Interface <code>Collection</code>	391
12.7.2	„Sammlungen“ durchgehen – der Aufbau des Interface <code>Iterator</code>	394
12.7.3	Mengen	395
12.7.3.1	Das Interface <code>Set</code>	395
12.7.3.2	Die Klasse <code>HashSet</code>	395
12.7.3.3	Das Interface <code>SortedSet</code>	397
12.7.3.4	Die Klasse <code>TreeSet</code>	398
12.7.4	Listen	399

12.7.4.1	Das Interface List	400
12.7.4.2	Die Klassen ArrayList und LinkedList	400
12.7.4.3	Suchen und Sortieren – die Klassen Collections und Arrays	402
12.7.5	Übungsaufgaben	405
12.8	Die Klasse StringTokenizer	406
12.8.1	Übungsaufgaben	408
III	Grafische Oberflächen in Java	409
13	Aufbau grafischer Oberflächen in Frames – von AWT nach Swing	411
13.1	Grundsätzliches zum Aufbau grafischer Oberflächen	411
13.2	Ein einfaches Beispiel mit dem AWT	413
13.3	Let's swing now!	415
13.4	Etwas „Fill-in“ gefällig?	417
13.5	Die AWT- und Swing-Klassenbibliothek im Überblick	419
13.6	Übungsaufgaben	421
14	Swing-Komponenten	423
14.1	Die abstrakte Klasse Component	423
14.2	Die Klasse Container	424
14.3	Die abstrakte Klasse JComponent	425
14.4	Layout-Manager, Farben und Schriften	426
14.4.1	Die Klasse Color	427
14.4.2	Die Klasse Font	429
14.4.3	Layout-Manager	430
14.4.3.1	Die Klasse FlowLayout	431
14.4.3.2	Die Klasse BorderLayout	433
14.4.3.3	Die Klasse GridLayout	434
14.5	Einige Grundkomponenten	436
14.5.1	Die Klasse JLabel	438
14.5.2	Die abstrakte Klasse AbstractButton	438
14.5.3	Die Klasse JButton	440
14.5.4	Die Klasse JToggleButton	441
14.5.5	Die Klasse JCheckBox	442
14.5.6	Die Klassen JRadioButton und ButtonGroup	443
14.5.7	Die Klasse JComboBox	445
14.5.8	Die Klasse JList	448
14.5.9	Die abstrakte Klasse JTextComponent	451
14.5.10	Die Klassen JTextField und JPasswordField	452
14.5.11	Die Klasse JTextArea	454
14.5.12	Die Klasse JScrollPane	456

14.5.13 Die Klasse <code>JPanel</code>	458
14.6 Spezielle Container, Menüs und Toolbars	460
14.6.1 Die Klasse <code>JFrame</code>	460
14.6.2 Die Klasse <code>JWindow</code>	461
14.6.3 Die Klasse <code>JDialog</code>	461
14.6.4 Die Klasse <code>JMenuBar</code>	465
14.6.5 Die Klasse <code>JToolBar</code>	467
14.7 Übungsaufgaben	470
15 Ereignisverarbeitung	473
15.1 Zwei einfache Beispiele	474
15.1.1 Zufällige Grautöne als Hintergrund	474
15.1.2 Ein interaktiver Bilderrahmen	477
15.2 Programmiervarianten für die Ereignisverarbeitung	481
15.2.1 Innere Klasse als Listener-Klasse	481
15.2.2 Anonyme Klasse als Listener-Klasse	481
15.2.3 Container-Klasse als Listener-Klasse	482
15.2.4 Separate Klasse als Listener-Klasse	483
15.3 Event-Klassen und -Quellen	485
15.4 Listener-Interfaces und Adapter-Klassen	489
15.5 Listener-Registrierung bei den Event-Quellen	494
15.6 Auf die Plätze, fertig, los!	498
15.7 Übungsaufgaben	502
16 Einige Ergänzungen zu Swing-Komponenten	507
16.1 Zeichnen in Swing-Komponenten	507
16.1.1 Grafische Darstellung von Komponenten	507
16.1.2 Das Grafik-Koordinatensystem	508
16.1.3 Die abstrakte Klasse <code>Graphics</code>	509
16.1.4 Ein einfaches Zeichenprogramm	512
16.1.5 Layoutveränderungen und der Einsatz von <code>revalidate</code>	514
16.2 Noch mehr Swing gefällig?	517
16.3 Übungsaufgaben	518
17 Applets	521
17.1 Erstellen und Ausführen von Applets	521
17.1.1 Vom Frame zum Applet am Beispiel	521
17.1.2 Applet in HTML-Datei einbetten	523
17.1.3 Applet über HTML-Datei ausführen	525
17.2 Die Methoden der Klasse <code>JApplet</code>	526
17.3 Zwei Beispiele	528
17.3.1 Auf die Plätze, fertig, los!	529

17.3.2	Punkte verbinden im Applet	532
17.4	Details zur HTML-Einbettung	533
17.4.1	Der Applet-Tag	533
17.4.2	Die Methode <code>showDocument</code>	536
17.5	Sicherheitseinschränkungen bei Applets	538
17.6	Übungsaufgaben	542
IV	Threads, Datenströme und Netzwerk-Anwendungen	545
18	Parallele Programmierung mit Threads	547
18.1	Ein einfaches Beispiel	547
18.2	Threads in Java	549
18.2.1	Die Klasse <code>Thread</code>	550
18.2.2	Das Interface <code>Runnable</code>	554
18.2.3	Threads vorzeitig beenden	556
18.3	Wissenswertes über Threads	558
18.3.1	Lebenszyklus eines Threads	558
18.3.2	Thread-Scheduling	560
18.3.3	Dämon-Threads und Thread-Gruppen	560
18.4	Thread-Synchronisation und -Kommunikation	561
18.4.1	Das Leser/Schreiber-Problem	562
18.4.2	Das Erzeuger/Verbraucher-Problem	566
18.5	Threads in Frames und Applets	573
18.5.1	Auf die Plätze, fertig, los!	573
18.5.2	Spielereien	577
18.5.3	Swing-Komponenten sind nicht Thread-sicher	579
18.6	Übungsaufgaben	580
19	Ein- und Ausgabe über I/O-Streams	583
19.1	Grundsätzliches zu I/O-Streams in Java	584
19.2	Dateien und Verzeichnisse – Die Klasse <code>File</code>	584
19.3	Ein- und Ausgabe über Character-Streams	587
19.3.1	Einfache <code>Reader</code> - und <code>Writer</code> -Klassen	588
19.3.2	Gepufferte <code>Reader</code> - und <code>Writer</code> -Klassen	591
19.3.3	Die Klasse <code>StreamTokenizer</code>	593
19.3.4	Die Klasse <code>PrintWriter</code>	594
19.3.5	Die Klassen <code>IOTools</code> und <code>Scanner</code>	596
19.3.5.1	Was machen eigentlich die <code>IOTools</code> ?	596
19.3.5.2	Konsoleneingabe über ein <code>Scanner</code> -Objekt	597
19.4	Ein- und Ausgabe über Byte-Streams	598
19.4.1	Einige <code>InputStream</code> - und <code>OutputStream</code> -Klassen	599
19.4.2	Die Serialisierung und Deserialisierung von Objekten	601

19.4.3	Die Klasse <code>PrintStream</code>	603
19.5	Einige abschließende Bemerkungen	603
19.6	Übungsaufgaben	604
20	Client/Server-Programmierung in Netzwerken	607
20.1	Wissenswertes über Netzwerk-Kommunikation	608
20.1.1	Protokolle	608
20.1.2	IP-Adressen	610
20.1.3	Ports und Sockets	611
20.2	Client/Server-Programmierung	612
20.2.1	Die Klassen <code>ServerSocket</code> und <code>Socket</code>	613
20.2.2	Ein einfacher Server	615
20.2.3	Ein einfacher Client	618
20.2.4	Ein Server für mehrere Clients	619
20.2.5	Ein Mehrzweck-Client	622
20.3	Wissenswertes über URLs	625
20.3.1	Client/Server-Kommunikation über URLs	625
20.3.2	Netzwerkverbindungen in Applets	626
20.4	Übungsaufgaben	627
V	Aktuelles, Ausblick und Anhang	631
21	Neuerungen in Java 7	633
21.1	Spracherweiterungen	633
21.1.1	Elementare Datentypen und Anweisungen	633
21.1.1.1	Binäre ganzzahlige Literalkonstanten	633
21.1.1.2	Unterstrich als Trennzeichen in Literalkonstanten	634
21.1.1.3	Strings in der switch -Anweisung	635
21.1.2	Verkürzte Notation bei generischen Datentypen	638
21.1.3	Ausnahmebehandlung	642
21.1.3.1	Mehrere Ausnahme-Typen in einem catch -Block	642
21.1.3.2	try -Block mit Ressourcen	645
21.2	Erweiterungen der Klassenbibliothek	648
21.2.1	Dateien und Verzeichnisse	648
21.2.1.1	Das Interface <code>Path</code> und die Klasse <code>Paths</code>	648
21.2.1.2	Die Klasse <code>Files</code>	649
21.2.2	Grafische Oberflächen	652
22	Neuerungen in Java 8	655
22.1	Lambda-Ausdrücke	655
22.1.1	Lambda-Ausdrücke in Aktion – zwei Beispiele	656
22.1.2	Lambda-Ausdrücke im Detail	659

22.1.3	Lambda-Ausdrücke und funktionale Interfaces	661
22.1.4	Vordefinierte funktionale Interfaces und Anwendungen auf Datenstrukturen	663
22.1.5	Methoden-Referenzen als Lambda-Ausdrücke	668
22.1.6	Zugriff auf Variablen aus der Umgebung innerhalb eines Lambda-Ausdrucks	670
22.2	Interfaces mit Default-Methoden und statischen Methoden	672
22.2.1	Deklaration von Default-Methoden	672
22.2.2	Deklaration von statischen Methoden	673
22.2.3	Auflösung von Namensgleichheiten bei Default-Methoden	674
22.2.4	Interfaces und abstrakte Klassen in Java 8	676
22.3	Streams und Pipeline-Operationen	676
22.3.1	Streams in Aktion	677
22.3.2	Streams und Pipelines im Detail	679
22.3.3	Erzeugen von endlichen und unendlichen Streams	680
22.3.4	Die Stream-API	682
23	Blick über den Tellerrand	687
23.1	Der Vorhang fällt	687
23.2	A fool with a tool	688
23.3	Alles umsonst?	689
23.4	Und fachlich?	690
23.5	Zu guter Letzt	692
A	Der Weg zum guten Programmierer...	693
A.1	Die goldenen Regeln der Code-Formatierung	694
A.2	Die goldenen Regeln der Namensgebung	697
A.3	Zusammenfassung	699
B	Die Klasse <code>IOTools</code> – Tastatureingaben in Java	701
B.1	Kurzbeschreibung	701
B.2	Anwendung der <code>IOTools</code> -Methoden	702
C	Der Umgang mit der API-Spezifikation	705
C.1	Der Aufbau der API-Spezifikation	705
C.2	Der praktische Einsatz der API-Spezifikation	706
D	Glossar	711
	Literaturverzeichnis	725
	Stichwortverzeichnis	729

Vorwort

Unsere moderne Welt mit ihren enormen Informations- und Kommunikationsbedürfnissen wäre ohne Computer und mobile Endgeräte wie Smartphones oder GPS-Geräte und deren weltweite Vernetzung undenkbar. Ob wir Einkäufe abwickeln, uns Informationen beschaffen, Fahrpläne abrufen, Reisen buchen, Bankgeschäfte tätigen oder einfach nur Post verschicken – wir benutzen diese Techniken wie selbstverständlich. Immer mehr Internet-Nutzer finden zudem Gefallen daran, Informationen oder gar Dienste, z. B. als Apps oder Web-Services, zur Verfügung zu stellen. Die Programmiersprache Java, die sich in den letzten Jahren zu einer weit verbreiteten Software-Entwicklungsplattform entwickelt hat, ermöglicht es, Software fürs Internet mit wenig Aufwand zu erstellen.

Dienstleistungen, Produkte und die gesamte Arbeitswelt basieren in zunehmendem Maße auf Software. Schul- und vor allem Hochschulabgänger werden mit Sicherheit an ihrem späteren Arbeitsplatz in irgendeiner Weise mit Software oder gar Software-Entwicklung zu tun haben. Eine qualifizierte Programmiergrundausbildung ist somit unerlässlich, um bewusst und aktiv am modernen gesellschaftlichen Leben teilnehmen oder gar an der Gestaltung moderner Informatikanwendungen mitwirken zu können. Leider erscheint vielen das Erlernen einer Programmiersprache zu Beginn einer weitergehenden Informatikausbildung als unüberwindbare Hürde. Mit Java rückte eine Sprache als Ausbildungssprache in den Vordergrund, die sehr mächtig und vielfältig ist, deren Komplexität es Programmier-Anfängern aber nicht unbedingt leichter macht, in die „Geheimnisse“ des Programmierens eingeweiht zu werden.

Angeregt durch unsere Erfahrungen aus vielen Jahren 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. Dabei wollten wir vor allem die Hauptanforderung „Verständlichkeit auch für Programmier-Anfänger“ erfüllen. Schülerinnen und Schülern, Studentinnen und Studenten, aber auch Hausfrauen und Hausmännern sollte mit diesem Buch ein leicht verständlicher Grundkurs „Programmieren in Java“ vermittelt werden. Auf theoretischen Ballast oder ein breites Informatikfundament wollten wir deshalb bewusst verzichten. Wir hofften, unser Konzept auch absolute Neulinge behutsam in die Materie einzuführen, überzeugt unsere Leserinnen und Leser. Diese Hoffnung wurde, wie

wir zahlreichen überaus positiven Leserkomentaren entnehmen konnten, mehr als erfüllt. So liegt nun bereits die siebte, überarbeitete Auflage vor, in der wir viele konstruktive Umgestaltungsvorschläge von Leserinnen und Lesern berücksichtigt und außerdem Neuerungen der Java-Version 8 aufgenommen haben.

Wenn man nach dem erfolgreichsten aller Bücher Ausschau hält, 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 all unserer guten Vorsätze (zumindest zu Beginn Ihrer Lektüre) wie Fach-Chinesisch oder böhmische Dörfer vorkommen könnte. Lassen Sie sich davon aber nicht abschrecken, denn 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: Die ehemaligen Tutoren Thomas Much, Michael Ohr und Oliver Wagner haben viel Schweiß und Mühe in die Erstellung von Teilen eines ersten Vorlesungsskripts gesteckt. Eine wichtige Rolle für die „Reifung“ bis zur vorliegenden Buchfassung spielten unsere „Korrektoren“ und „Testleser“. Hagen Buchwald, Michael Decker, Tobias Dietrich, Rudi Klatte, Niklas Kühl, Roland Küstermann, Joachim Melcher, Cornelia Richter-von Hagen, Sebastian Ratz, Frank Schlottmann, Oliver Schöll und Leonard von Hagen brachten mit großem Engagement wertvolle Kommentare und Verbesserungsvorschläge ein oder unterstützten uns beim Auf- und Ausbau der Buch-Webseite, bei der Überarbeitung von Grafiken oder mit der Erstellung und Bereitstellung von einfach zu handhabenden Entwicklungstools. Schließlich sind da noch mehrere Studierenden-Jahrgänge der Studiengänge Wirtschaftsingenieurwesen, Wirtschaftsmathematik, Technische Volkswirtschaftslehre und Wirtschaftsinformatik, die sich im Rahmen unserer Lehrveranstaltungen „Programmieren I“, „Programmierung kommerzieller Systeme“, „Fortgeschrittene Programmieretechniken“, „Web-Programmierung“ und „Verteilte Systeme“ mit den zugehörigen Webseiten, Foliensätzen und Übungsblättern „herumgeschlagen“ und uns auf Fehler und Unklarheiten hingewiesen haben. Das insgesamt sehr positive Feedback, auch aus anderen Studiengängen, war und ist Ansporn für uns, diesen Grundkurs Programmieren weiterzuentwickeln. Schließlich geht auch ein Dankeschön an die Leserinnen und Leser, die uns per E-Mail Hinweise und Tipps für die inhaltliche Verbesserung von Buch und Webseite zukommen ließen.

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

Kapitel 1

Einleitung

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

Wie kann Ihnen dieses Buch helfen, so etwas zu vermeiden? Die traurige Antwort lautet: Gar nicht! Sie können mit diesem Buch weder Frauen beeindrucken noch hochgewachsene Kerle niederschlagen (denn dafür ist es einfach zu leicht). Wenn Sie also einen schnellen Weg zum sicheren Erfolg suchen, sind Sie wohl mit anderen Werken besser beraten. Wozu ist das Buch also zu gebrauchen? Die folgenden Seiten verraten es Ihnen.

1.1 Java – mehr als nur kalter Kaffee?

Seit dem Einzug von Internet und World Wide Web (WWW) ins öffentliche Leben surfen, mailen und chatten Millionen von Menschen täglich in der virtuellen Welt. Es gehört beinahe schon zum guten Ton, im Netz der Netze vertreten zu sein. Ob Großkonzern oder privater Kegelclub – jeder will seine eigene Homepage.

Dieser Entwicklung hat es die Firma Sun, die im Januar 2010 von Oracle übernommen wurde, zu verdanken, dass ihre Programmiersprache Java einschlug wie eine Bombe. Am eigentlichen Sprachkonzept war nur wenig Neues, denn die geistigen Väter hatten sich stark an der Sprache C++ orientiert. Im Gegensatz zu C++ konnten mit Java jedoch Programme erstellt werden, die sich direkt in Webseiten einbinden und ausführen lassen. Java war somit die erste Sprache für das WWW.

Natürlich ist für Java die Entwicklung nicht stehen geblieben. Die einstige „Netzsprache“ hat sich in ihrer Version 8 (siehe z. B. [33] und [27]), mit der wir in diesem Buch arbeiten, zu einer vollwertigen Konkurrenz zu den anderen gängigen Konzepten gemausert.¹ Datenbank- oder Netzwerkzugriffe, anspruchsvolle Grafikanwendungen, Spieleprogrammierung – alles ist möglich. Gerade in dem heute so aktuellen Bereich „Verteilte Anwendungsentwicklung“ bietet Java ein breites Spektrum an Möglichkeiten. Mit wenigen Programmzeilen gelingt es, Anwendungen zu schreiben, die das Internet bzw. das World Wide Web (WWW) nutzen oder sogar über das Netz übertragen und in gängigen Web-Browsern gestartet werden können. Grundlage dafür bildet die umfangreiche Java-Klassenbibliothek, die Sammlung einer Vielzahl vorgefertigter Klassen und Interfaces, die einem das Programmiererleben wesentlich vereinfachen. Nicht minder interessante Teile dieser Klassenbibliothek statten Java-Programme mit enormen, weitgehend plattformunabhängigen grafischen Fähigkeiten aus. So können auch Programme mit grafischen Oberflächen portabel bleiben. Dies erklärt sicherlich auch das große Interesse, das der Sprache Java in den letzten Jahren entgegengebracht wurde. Bedenkt man die Anzahl von Buchveröffentlichungen, Zeitschriftenbeiträgen, Webseiten, Newsgroups, Foren und Blogs zum Thema, so wird der erfolgreiche Weg, den die Sprache Java hinter sich hat, offensichtlich. Auch im kommerziellen Bereich ist Java nicht mehr wegzudenken, denn die Produktpalette der meisten großen Softwarehäuser weist mittlerweile eine Java-Schiene auf. Und wer heute auch nur mit einem Handy telefoniert, kommt häufig 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.²

1.2 Java für Anfänger – das Konzept dieses Buches

Da sich Java aus dem etablierten C++ entwickelt hat, gehen viele Buchautoren davon aus, dass derjenige, der Java lernen will, bereits C++ kennt. Das macht erfahrenen Programmierern die Umstellung leicht, stellt Anfänger jedoch vor unüberwindbare Hürden. Manche Autoren versuchen, Eigenschaften der Sprache Java durch Analogien zu C++ oder zu anderen Programmiersprachen zu erklären, und setzen entsprechende Kenntnisse voraus, die den Einstieg in Java problemlos möglich machen. Wie sollen jedoch Anfänger, die über diese Erfahrung noch nicht verfügen, ein solches Buch verstehen? Erst C++ lernen und dann Java?

Die Antwort auf diese Frage ist ein entschiedenes Nein, denn Sie lernen ja auch nicht Latein, um Französisch zu sprechen. Tatsächlich erkennen heutzuta-

¹ Die Version 5 brachte gegenüber der Vorgängerversion 1.4 einige sehr interessante Erweiterungen, Erleichterungen und Verbesserungen. Kleinere Spracherweiterungen brachte Version 7 im Jahr 2011, und im Frühjahr 2014 ist die Version 8 mit umfangreichen Neuerungen erschienen.

² Als potenzieller Berufseinsteiger oder -umsteiger wissen Sie vielleicht ein Lied davon zu singen, wenn Sie sich Stellenanzeigen im Bereich Software-Entwicklung ansehen – Java scheint allgegenwärtig zu sein.

ge immer mehr Autoren und Verlage, dass die Einsteiger-Literatur sträflich vernachlässigt wurde. Es ist daher zu hoffen, dass die Zahl guter und verständlicher Programmierkurse für Neulinge weiter zunimmt. *Einen dieser Kurse halten Sie gerade in den Händen.*

Wie schreibt man nun ein Buch für den absoluten Neueinsteiger, wenn man selbst seit vielen Jahren programmiert? Vor diesem Problem standen die Autoren. Es sollte den Leserinnen und Lesern die Konzepte von Java korrekt vermitteln, ohne sie zu überfordern.

Maßstab für die Qualität dieses Buches war deshalb die Anforderung, dass es sich optimal als Begleitmaterial für einführende und weiterführende Vorlesungen in Bachelor-Studiengängen einsetzen ließ, wie zum Beispiel die Veranstaltungen „Programmieren I – Java“ und „Programmierung kommerzieller Systeme – Anwendungen in Netzen mit Java“ des Instituts für Angewandte Informatik und Formale Beschreibungsverfahren (Institut AIFB), die jedes Winter- bzw. Sommersemester am Karlsruher Institut für Technologie (KIT – Universität des Landes Baden-Württemberg und nationales Großforschungszentrum in der Helmholtz-Gemeinschaft) für rund 600 bzw. 400 Studierende abgehalten wird.

Weil die Autoren auf mehrere Jahre studentische Programmierausbildung (in oben genannten Veranstaltungen, in Kursen an der Dualen Hochschule Baden-Württemberg (DHBW) Karlsruhe und in weiterführenden Veranstaltungen im Bereich Programmieren) zurückblicken können, gab und gibt es natürlich gewisse Erfahrungswerte darüber, welche Themen gerade den Neulingen besondere Probleme bereiteten. Daher auch der Entschluss, das Thema „Objektorientierung“ zunächst in den Hintergrund zu stellen. Fast jedes Java-Buch beginnt mit diesem Thema und vergisst, dass man zuerst programmieren und „algorithmisch denken“ können muss, bevor man die Vorteile der objektorientierten Programmierung erkennen und nutzen kann. Seien Sie deshalb nicht verwirrt, wenn Sie dieses sonst so beliebte Schlagwort vor Seite 185 wenig zu Gesicht bekommen.

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

In den Teilen III und IV führen wir Sie auch in die Programmierung fortgeschrittener Anwendungen auf Basis der umfangreichen Java-Klassenbibliothek ein. Wir können und wollen dabei aber nicht auf jedes Detail eingehen, sodass wir alle Leserinnen und Leser bereits an dieser Stelle dazu animieren möchten, regelmäßig einen Blick in die so genannte API-Spezifikation³ der Klassenbibliothek [33] zu werfen – nicht zuletzt, weil wir im „Programmier-Alltag“ von einem routinier-ten Umgang mit API-Spezifikationen nur profitieren können. Sollten Sie Schwierigkeiten haben, sich mit dieser von Sun bzw. Oracle zur Verfügung gestellten

³ API steht für Application Programming Interface, die Programmierschnittstelle für eine Klasse, ein Paket oder eine ganze Klassenbibliothek.

Dokumentation der Klassenbibliothek zurechtzufinden, hilft Ihnen vielleicht unser kleines Kapitel im Anhang C.

1.3 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 Webseite

<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, interessante Links, eine Liste eventueller Fehler im Buch und deren Korrekturen. Dozenten, die das Material dieses Buchs oder Teile der 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.

1.4 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.

Kapitel 8

Der grundlegende Umgang mit Klassen

Im letzten Kapitel haben wir erfahren, dass sich die objektorientierte Philosophie aus den vier Konzepten Generalisierung, Vererbung, Kapselung und Polymorphismus zusammensetzt. Wir haben jeden dieser Begriffe – in der Theorie – erklärt und uns die Idee klar zu machen 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.

8.1 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  }
```

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

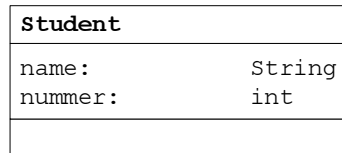


Abbildung 8.1: Die Klasse `Student`, erste Version

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

Abbildung 8.1 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 instantiiieren (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 5 schon zu 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 Polymorphismus. 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 8.2 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 so genannter **Zugriffsmethoden** eine bessere Form der Datenkapselung erreichen.

8.2 Instanzmethoden

8.2.1 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 so genannten **Zugriffsrechte** für die einzelnen Variablen. Momentan haben unsere Variablen die Zugriffsrechte **public**, das heißt, sie sind *öffentlich zugänglich*. Konkret bedeutet es, dass jede andere Klasse auf die Variablen lesenden und schreibenden Zugriff hat. Genau das wollen wir jedoch verhindern!

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

Abbildung 8.2 zeigt diese Modifikation im UML-Diagramm. Wir sehen, dass private Variablen durch ein Minuszeichen vor dem Variablennamen markiert werden. Fehlt dieses Symbol oder ist es durch ein Pluszeichen ersetzt, geht man von öffentlichen Zugangsrechten aus.³

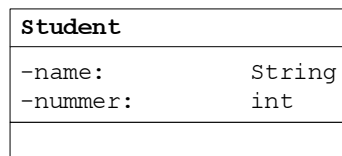


Abbildung 8.2: Die Klasse Student, zweite Version

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

```

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

```

³ Neben öffentlichem und privatem Zugriff gibt es zwei weitere Formen des Zugriffs (siehe Abschnitt 9.8.2).

```

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

```

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

```

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

```

setzen wollen, erhalten wir beim Übersetzen eine Fehlermeldung der Form

```

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

```

Das heißt: Die Zugriffe wurden verweigert.

8.2.2 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 so genannte **Instanzmethoden** hinzu. Diese Methoden werden ähnlich wie in Kapitel 6 definiert:

```

----- Syntaxregel -----
public <<RUECKGABETYP>> <<METHODENNAME>> ( <<PARAMETERLISTE>> )
{
    // hier den auszufuehrenden Code einfuegen
}

```

Wenn Sie dies mit der Syntaxregelbox auf Seite 151 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 8.3 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 8.3: 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 5.2.3 bekannten Form

<i>Syntaxregel</i>
«OBJEKTNAME».«VARIABLENNAME»

erreichen, indem wir für den Platzhalter «OBJEKTNAME» schlicht und ergreifend `this` einsetzen. Abbildung 8.4 verdeutlicht nochmals die Bedeutung der `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;
}
```

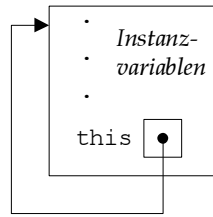



Abbildung 8.4: Die `this`-Referenz

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 zurueck */
public int getNummer() {
    return nummer;
}
```

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

■ Zuletzt formulieren wir eine Methode

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

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

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

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.

```
        nummer = n;  
    }
```

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 6) 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.

8.2.3 Instanzmethoden zur Validierung von Eingaben

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

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

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

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

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

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

```

/** Pruefe die Matrikelnummer des Studenten
    auf ihre Gueltigkeit */
public boolean validateNumber() {
    return
        (nummer >= 10000 && nummer <= 99999 && nummer % 2 != 0);
}

```

Wir können nun also unserem Studenten 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 setNumber(int n) {
    int alteNummer = nummer;
    nummer = n;
    if (!validateNumber()) { // neue Nummer ist nicht gueltig
        nummer = alteNummer;
    }
}

```

Unsere angepasste Methode durchläuft die Prüfung in mehreren Schritten. Zuerst setzt sie die Matrikelnummer des Studenten auf den neuen Wert, speichert aber den alten Wert in der Variable `alteNummer` ab. Anschließend ruft sie die `validateNumber`-Methode 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!

8.2.4 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`. Instantiiert man 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 9.4 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 des Polymorphismus eine maßgeschneiderte Ausgabe für unsere Klasse modelliert.

8.3 Statische Komponenten einer Klasse

Wir haben im letzten Abschnitt mit den Instanzvariablen und -methoden ein wichtiges Gebiet des objektorientierten Programmierens kennen gelernt. Die

⁶ Auch die Methode `println` ist Instanzmethode eines Objektes, des so genannten Ausgabestroms. Das Objekt `System.out` ist ein solcher Strom.

Möglichkeit, Variablen oder sogar ganze Methoden einem bestimmten Objekt zuzuordnen zu können, hat uns Perspektiven erschlossen, die wir mit unseren bisherigen Programmiererfahrungen nicht sahen.

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 6, 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 so genannten **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.

8.3.1 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 instantiierten 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

Syntaxregel

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

Wir stellen fest, dass sich die Definition von Klassenvariablen nicht sehr von dem unterscheidet, was wir in Abschnitt 5.2 über Instanzvariablen gelernt haben. Mit Hilfe des Wortes **private** schützen wir unsere Variable vor Zugriffen von außerhalb. Typ, Variablenname und Initialwert sind uns ebenfalls bekannt und würden im Fall unseres Zählers zu folgender Definition führen:

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

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

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

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

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

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

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

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

Unsere Methode zählt bei Aufruf zuerst die Variable `zaehler` hoch und aktualisiert somit deren Stand. Im zweiten Schritt wird mit Hilfe des **new**-Operators ein

⁷ Der initiale Wert könnte an dieser Stelle auch wegfallen.

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

Abbildung 8.5: Die Klasse `Student`, mit Objektzähler

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 8.4.1 jedoch eine Methode kennen lernen, diese Probleme auf elegante Art und Weise zu lösen.

Jetzt werfen wir noch einen Blick auf unsere gewachsene Klasse `Student` im UML-Klassendiagramm (Abbildung 8.5). 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 einzuzeichnen, 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.

8.3.2 Klassenkonstanten

Wie wir aus Abschnitt 4.4.1 wissen, ist es möglich, mit Hilfe des Schlüsselwortes **final** aus „normalen“ Variablen **final**-Variablen zu machen, sie also zu sym-

bolischen 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 *Studienfach* an einer Hochschule einschreibt, wird dieses Fach in den Systemen vieler Hochschulverwaltungen mit einer bestimmten Nummer identifiziert.

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

Tabelle 8.1: Zuordnung Studienfach – Verwaltungsnummer

Tabelle 8.1 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 8.1 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.";
}
}

```

8.4 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 214 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.

8.4.1 Konstruktoren

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

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

instanziiert, wobei der **new**-Operator angewendet wurde, entsprechend der bereits auf Seite 135 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 8.4.4 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 so genannte **Konstruktor**.

Konstrukturen sind keine Methoden im eigentlichen Sinn, da sie nicht – wie etwa Klassen- oder Instanzmethoden – explizit aufgerufen werden. Sie haben auch keinen Rückgabetyt (nicht einmal **void**). Die Definition des Konstruktors erfolgt nach dem Schema: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 vereinbaren können (was wir im nächsten Abschnitt auch tun werden).

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

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

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

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

ein Objekt instantiiieren. 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:

⁸ Hierbei kann man statt **public** natürlich auch andere Zugriffsrechte vergeben.

```

/** 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();
}

```

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 214 angemahnte Abwärtskompatibilität⁹ wiederhergestellt.

8.4.2 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:

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

```

/** Argumentloser Konstruktor */
public Student() {
    zaehler++;
    geburtsjahr = 1970;
}

```

Wir setzen also den Inhalt unserer Variablen auf einen Standardwert, das Jahr 1970, 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++;
    this.geburtsjahr = geburtsjahr;
}

```

Wir haben unseren Konstruktor also **überladen**, wie wir es schon in Abschnitt 6.1.5 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 1982 geborenen Studenten:

```
Student studi = new Student(1982);
```

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 = 1970;
}

/** 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(1970);
}
```

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.

8.4.3 Der statische Initialisierer

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

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

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

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

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

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

nur lesenden Zugriff auf den Zähler haben, können wir diesen Umstand nicht rückgängig machen.

Wie wir sehen, kommen wir an dieser Stelle mit einer Initialisierung „von außen“ nicht weiter. Wir benötigen eine Möglichkeit, statische Komponenten einer Klasse beim 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:

<i>Syntaxregel</i>
<pre>static { // hier den auszufuehrenden Code einfuegen }</pre>

In einer Klasse können beliebig viele static-Blöcke auftreten. Sobald die Klasse dem Java-System bekannt gemacht wird (das so genannte 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
   =====
*/
```

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**.

```

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 Konstanten `PHANTOM` zugewiesen. Unsere Konstante ist somit belegt und kann nicht mehr verändert werden.

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

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

Haben wir somit gegen das Gesetz, finale Variablen nicht mehr verändern zu können, verstoßen? Die Antwort lautet *nein*, und ihre Begründung liegt wieder einmal in dem Umstand, dass es sich bei Klassen um Referenzdatentypen handelt. In unserer finalen Variablen `PHANTOM` steht nämlich nicht das Objekt selbst, sondern eine *Referenz*, also ein Verweis auf das tatsächliche Objekt. Diese Referenz ist konstant, das heißt, unsere Variable wird immer auf ein und dasselbe Studentenobjekt verweisen. Das Objekt selbst ist jedoch ein ganz „normaler“ Student und kann als solcher von uns auch 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

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

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.

8.4.4 Der Mechanismus der Objekterzeugung

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

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

einen Initialisierer hinzufügen und ferner im Konstruktor die Zeile

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

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

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 8.6 skizziert sind.

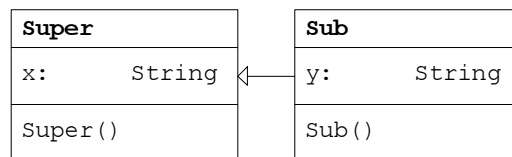


Abbildung 8.6: Beispielklassen für Abschnitt 8.4.4

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

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

```
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 so genannter `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 8.2) belegt.

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

Tabelle 8.2: Default-Werte von Instanzvariablen

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

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

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

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

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

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

Unsere Klasse Sub leitet sich hierbei von der Klasse Super ab, was wir in Java durch das Schlüsselwort **extends** zum Ausdruck bringen. Der restliche Aufbau der Klasse ergibt sich auch aus dem dazugehörigen UML-Diagramm 8.6:

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

Wenn wir nach dem allgemeinen Muster vorgehen, unterteilt sich der Instanzierungsprozess in verschiedene Schritte. Wir haben den Ablauf in neun Einzelschritten zerlegt, die in Abbildung 8.7 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.

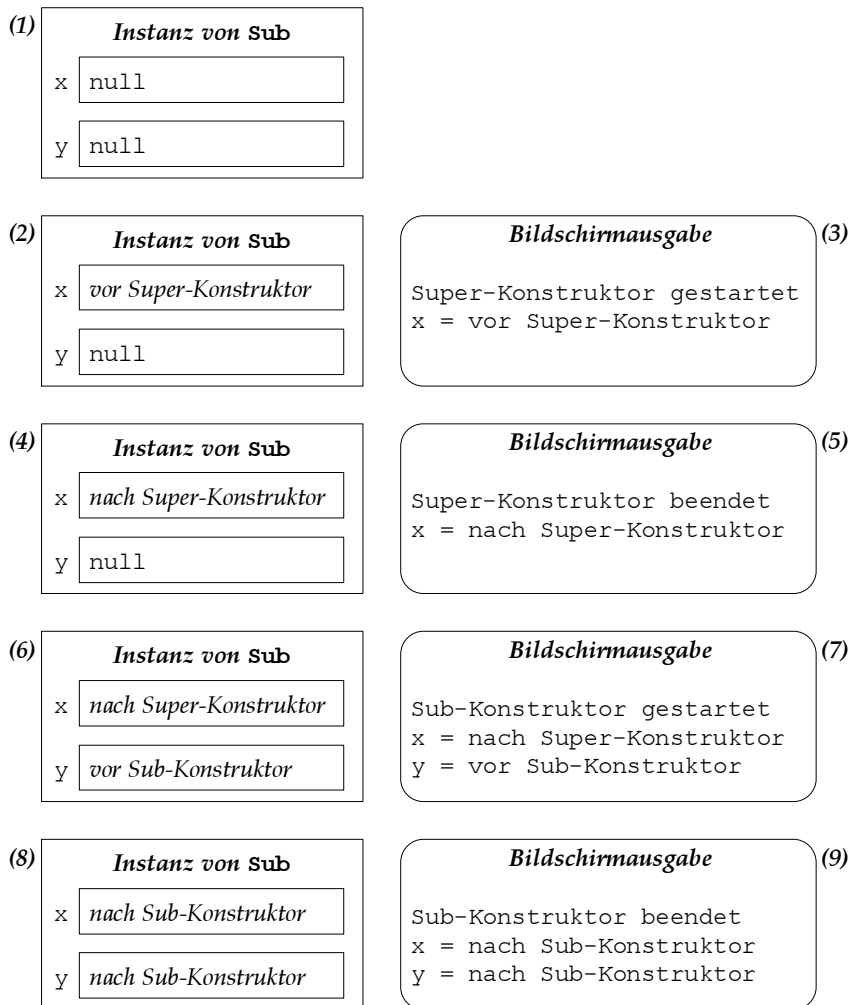


Abbildung 8.7: Instanziierungsprozess von Sub- und Superklasse

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 Super-Klasse 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.
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 Instantiierungsprozesses 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 Instantiierungsprozess 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.

8.5 Zusammenfassung

Wir haben anhand eines einfachen Anwendungsfalles – der Klasse `Student` – die grundlegenden Mechanismen kennen gelernt, um in Java mit Klassen umzugehen. Wir haben Instanzvariablen und Instanzmethoden kennen gelernt – 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** kennen gelernt, 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.

8.6 Übungsaufgaben

Aufgabe 8.1

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 8.2

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 8.3

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 des Polymorphismus wird die neue Methode das Original in allen Karlsruher Studentenobjekten ersetzen. Da die `set`-Methode jedoch die Validierung verwendet, haben wir die Wertzuweisung automatisch dem neuen System angepasst.

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

Aufgabe 8.4

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

- a) Die Klasse ... ist Superklasse der Klasse ...
- b) Die Klasse ... erbt von der Klasse ... die Variable(n) ...
- c) In den drei Klassen gibt es die Instanzvariable(n) ...
- d) In den drei Klassen gibt es die Klassenvariable(n) ...
- e) Auf die Variable(n) ... der Klasse `Klang` kann in der Klasse `Krach` und in der Klasse `Musik` zugegriffen werden.
- f) Auf die Variable(n) ... der Klasse `Krach` hat keine andere Klasse Zugriff.
- g) Die Variable(n) ... hat/haben in allen Instanzen der Klasse `Krach` den gleichen Wert.
- h) Der Konstruktor der Klasse `Klang` wird in den Zeilen ... aufgerufen.
 - i) Die Methode `mehrPower` der Klasse `Klang` wird in den Zeilen ... bis ... überschrieben und in den Zeilen ... bis ... überladen.
 - j) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in Zeile ... und in Zeile ... aufgerufen.
 - k) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in Zeile ... aufgerufen.
 - l) Die Methode `mehrPower`, die in den Zeilen ... bis ... definiert ist, wird in ... aufgerufen.
- m) Die Methode `toString`, die in den Zeilen 7 bis 9 definiert ist, wird in ... aufgerufen.
- n) Die Methoden ... sind Instanzmethoden.

Auf die nachfolgenden Klassen sollen sich Ihre Antworten beziehen:

```
1 public class Klang {
2     public int baesse, hoehen;
3     public Klang(int b, int h) {
4         baesse = b;
5         hoehen = h;
6     }
7     public String toString () {
8         return "B:" + baesse + " H:" + hoehen;
9     }
}
```



```
10     public void mehrPower (int b) {
11         baesse += b;
12     }
13 }
14 public class Krach extends Klang {
15     private int rauschen, lautstaerke;
16     public static int grundRauschen = 4;
17     public Krach (int l, int b, int h) {
18         super(b,h);
19         lautstaerke = 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         r.mehrPower(3);
39         r.mehrPower(2,2);
40     }
41 }
```

Aufgabe 8.5

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 8.6

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 8.7

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

```
1  public class Fehler1 {
2
3      /** Private Instanzvariable */
4      private String name;
5
6      /** Konstruktor */
7      public Fehler1(String name) {
8          name = name;
9      }
10
11     /** String-Ausgabe */
12     public String toString() {
13         return "Name = " + name;
14     }
15
16     /** Hauptprogramm */
17     public static void main(String[] args) {
18         System.out.println(new Fehler1("Testname"));
19     }
20
21 }
```

```
1  public class Fehler2 {
2
3      /** Private Instanzvariable */
4      private String name;
5
6      /** Konstruktor */
7      public Fehler2(String name) {
8          this.name = name;
9      }
10 }
```

```
11  /** String-Ausgabe */
12  public String toString() {
13      return "Name = " + name;
14  }
15
16  /** Hauptprogramm */
17  public static void main(String[] args) {
18      System.out.println(new Fehler2("Testname"));
19  }
20
21  }
```



```
1  public class Fehler3 {
2
3      /** Private Instanzvariable */
4      private String name;
5
6      /** Konstruktor */
7      public Fehler3(String nom) {
8          name = nom;
9      }
10
11     /** String-Ausgabe */
12     public String toString() {
13         return "Name = " + name;
14     }
15
16     /** Hauptprogramm */
17     public static void main(String[] args) {
18         System.out.println(new Fehler2("Testname"));
19     }
20
21 }
```



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

```
1 public class Fehler5 {
2
3     /** Private Instanzvariable */
4     private String name;
5
6     /** Konstruktor */
7     public 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
23 }
```

Aufgabe 8.8

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 }
```

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

```
TennisSpieler maier;
maier = new TennisSpieler();
```

Warum benötigt man die zweite Anweisung überhaupt?

- c) Erläutern Sie die Bedeutung der `this`-Referenz grafisch und anhand der Methode `altersDifferenz`.
d) Wie erfolgt der Zugriff auf die Daten (Variablen) und Methoden der Klasse?
e) 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?

- f) Erläutern Sie den Unterschied zwischen Instanzvariablen und Klassenvariablen.
g) Erweitern Sie die Klasse `TennisSpieler` um eine Instanzvariable namens `verfolger`, die eine Referenz auf einen weiteren Tennisspieler (den unmittelbaren Verfolger in der Weltrangliste) darstellt, und um eine Instanzvariable `startNummer`, die es ermöglicht, allen Tennisspielern (z. B. bei der Erzeugung eines neuen Objektes für eine Teilnehmerliste eines Turniers) eine (eindeutige) ganzzahlige Nummer zuzuordnen.
h) Erweitern Sie die Klasse `TennisSpieler` um eine Klassenvariable namens `folgeNummer`, die die jeweils nächste zu vergebende Startnummer enthält.
i) Modifizieren Sie den Konstruktor der Klasse `TennisSpieler` so, dass er jeweils eine entsprechende Startnummer vergibt und die Klassenvariable `folgeNummer` jeweils erhöht. Geben Sie auch eine Überladung dieses Konstruktors an, die es ermöglicht, bei der Objekterzeugung auch noch den Verfolger in der Weltrangliste anzugeben.
j) Wie verändert sich der Wert der Variablen `startNummer` und `folgeNummer` in den Objekten `maier`, `schmid` und `berger` mit den nachfolgenden Anweisungen?

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

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

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

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

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

ausgeben?

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

Aufgabe 8.9

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 8.10

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 8.8).

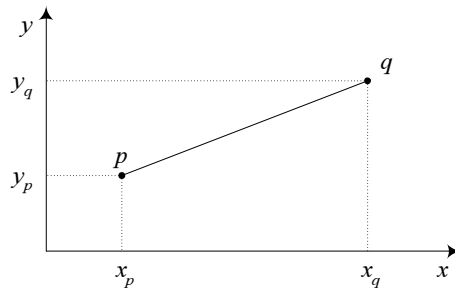


Abbildung 8.8: Definition einer Strecke

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 statten Sie die Klasse `Punkt` mit Konstruktoren und Instanzmethoden aus. Schreiben Sie
 - einen Konstruktor mit zwei **double**-Parametern (die x - und y -Koordinaten des Punktes),
 - eine Methode `getX()`, die die x -Koordinate des Objekts zurückliefert,
 - eine Methode `getY()`, die die y -Koordinate des Objekts zurückliefert,
 - eine **void**-Methode `read()`, die die x - und y -Koordinaten des Objekts einliest, und
 - eine `String`-Methode `toString()`, die die `String`-Darstellung des Objekts in der Form `(xStr, yStr)` zurückliefert, wobei `xStr` und `yStr` die `String`-Darstellungen der Werte von `x` und `y` sind.
- b) Implementieren Sie die Klasse `Strecke` mit zwei privaten Instanzvariablen `p` und `q` vom Typ `Punkt`, die die beiden Randpunkte einer Strecke repräsentieren, und statten 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 8.11

Gegeben sei die folgende Klasse:

```
1 public class AchJa {
2
3     public int x;
4     static int ach;
5
6     int ja (int i, int j) {
7         int y;
8         if ((i <= 0) || (j <= 0) || (i % j == 0) || (j % i == 0)) {
9             System.out.print(i+j);
10            return i + j;
11        }
12        else {
13            x = ja(i-2,j);
14            System.out.print(" + ");
15            y = ja(i,j-2);
16            return x + y;
17        }
18    }
19
20    public static void main (String[] args) {
21        int n = 5, k = 2;
22        AchJa so = new AchJa();
23        System.out.print("ja(" + n + ", " + k + ") = ");
24        ach = so.ja(n,k);
25        System.out.println(" = " + ach);
26    }
27 }
```

- 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 8.12

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 8.13

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 8.14

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

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

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

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

```

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

```

Aufgabe 8.15

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 8.9 beschrieben aus.

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

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

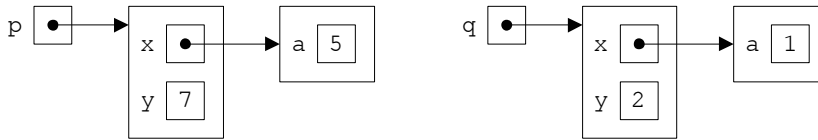


Abbildung 8.9: Ausgangsbild Aufgabe 8.15

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 8.16

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 der 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 Füllstands-Angabe 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 Glasboden-Fläche, U den Glasboden-Umfang und s den Füllstand eines Trinkglases, so sollen die Innenfläche I und die Füllmenge M dieses Trinkglases durch

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

berechnet werden.

```

public class TrinkGlas {
    .
    .
    .
    .
    .
    public TrinkGlas (double fuellStand, Glasboden boden) {
        .
        .
        .
    }
    public void verkleinern (double x) {
        .
        .
        .
    }
}

```


Stichwortverzeichnis

| 71, 74, 644
|| 74
|= 73
* 68
*/ 41, 42
*= 73
+ 46
++ 46
+= 73
- 46
-- 46
-= 73
-> 659
/ 68
/* 41
/** 42
// 41
/= 73
:: 668
< 73
<< 71
<= 73
<> 639
== 46, 73
> 73
>= 73
>> 71
? : 47, 74
% 68
%= 73
& 71, 74
&= 73
&& 74
^ 71
~ 71

Ablaufsteuerung 84
abs 366, 372
Abschlussoperationen 680, 683, 684
abstract 250
Abstract Window Toolkit 409, 419
AbstractButton 438
abstrakte Klassen 251, 266, 266
Absturz 711
accept 613, 664
ActionEvent 486, 487
ActionListener 475, 489
actionPerformed 475, 489
Adapter-Klassen 492
add 366, 371, 383, 392, 400, 419, 424, 425, 444, 465
addActionListener 476
addAll 392
addItem 446
addSeparator 465
Adresse 64, 105
after 380, 382
Aggregation 198
aktueller Parameter 154
algorithmische Beschreibung 28
Algorithmus 24, 711
Alias-Namen 610
allMatch 683
ALT_MASK 467
ancestorAdded 491
AncestorEvent 486
AncestorListener 491
ancestorMoved 491
ancestorRemoved 491
anonyme Klasse 282, 481
Anteil, ganzzahliger 57

- Anweisungen **32, 84**
 - Ausdrucks- 85
 - break** 93
 - case** 87
 - continue** 94
 - default** 87
 - do** 91
 - Entscheidungs- 86
 - for** 89
 - if** 86
 - import** 47
 - leere 85
 - markierte 93
 - package** 275
 - return** 95, 152
 - switch** 87, 636
 - while** 91
 - Wiederholungs- 89
- Anwendungsfälle 198
- Anwendungsschicht **608**
- Anwendungssoftware **24**
- anyMatch 683
- API **705, 706, 711**
- API-Spezifikation 175
- append 353
- APPEND 650
- Applet **49, 409, 521, 527, 711**
 - signiertes **539**
- Applet-Tag 533
- AppletContext 536
- Applikation **49, 409, 712**
- apply 663, 664
- Arbeitsspeicher **24, 105, 712**
- Archiv 534
- args 151, 168
- Argument, formales **151**
- Argumentliste **151**
- ArithmeticException 296
- Arithmetische Operatoren 68
- arraycopy 119, 159
- ArrayList 400
- Arrays 404
- asList 405
- Assemblersprache **26**
- assert** 320
- AssertionError 320
- Assertions **320**
- ATOMIC_MOVE 650
- Attribute **524**
- Aufruf von Methoden 154
- Aufzählungstypen **324**
- Ausdrücke **32, 67, 77**
 - konstante 70
 - Lambda- **655, 659**
- Ausdrucksanweisung 85
- Ausgabe **34, 51, 375, 376, 583, 587, 598, 603**
 - formatierte **378**
- Ausgabeeinweisung 34
- Ausgabegeräte **24**
- Ausnahme 296, 642
- Aussage, logische 60
- Auswahllisten 445, 448
- Auswertungsreihenfolge 76
- Autoboxing **361**
- AutoCloseable 647
- automatische Typkonvertierung 60, 155, 254
- automatische Typumwandlung 60, 155, 254
- Autounboxing **361**
- average 684
- AWT **409, 419**
- AWTEvent 486
- Basis-Container 413, 415
- Baukastenprinzip 411
- Beautifier **87**
- bedingtes logisches Oder 74
- bedingtes logisches Und 74
- Beenden **414**
- before 380, 382
- Behälter 411
- Benutzungsschnittstelle, grafische **411**
- Berechnungen 67
- Betriebssystem **712**
- Bezeichner 43
- BiConsumer 664
- BiFunction 663
- BigDecimal 370
- BigInteger 365
- Bildlaufleisten 456
- Bildschirmausgabe 34
- binäre Operatoren 67
- binäre Zahlen **634, 712**
- Binärfolge 55

- BinaryOperator 664
- binarySearch 403, 404
- Binden
 - dynamisches 260
- BiPredicate 664
- Bit 24, 70
- Bitoperatoren 70
- BLACK 427
- Block 35, 46, 51, 85
 - Anfang 51
 - Ende 51
 - static-** 222
 - Struktur 85
- BLUE 427
- BOLD 429
- boolean** 60
- Boolean 357
- booleanValue 358
- BorderLayout 433
- BOTTOM 436
- Bounded Wildcards 337
- Boxing 360
- break** 88, 93
- Browser 409, 713
- Buchstaben 43
- Buffer 604
- BufferedInputStream 600
- BufferedOutputStream 600
- BufferedReader 591, 681
- BufferedWriter 591
- Bug 713
- Bugfix 713
- ButtonGroup 443
- byte** 56
- Byte 24, 70
- Byte 357
- Byte-Streams 584
- Bytecode 27, 37
- Byteströme 584
- byteValue 358

- Calendar 382
- Call by reference 158
- Call by value 154
- canRead 585
- canWrite 585
- CaretEvent 487
- CaretListener 489

- caretUpdate 489
- case** 87
- Cast 60
- catch** 298, 299
 - mehrfaches 644
- CENTER 431, 433, 436, 452
- ChangeEvent 487
- ChangeListener 489
- Channel 604
- char** 59
- Character 357
- Character-Streams 584
- charValue 358
- CheckedInputStream 604
- CheckedOutputStream 604
- clear 392
- clearSelection 448
- Client 612, 713
- Client-Host 612
- Client-Rechner 612
- Client/Server-Programmierung 607
- clone 119
- close 587, 599, 614, 648
- Code Formatter 87
- Codierung 28
- Collection 391, 392, 680
- Collections 402
- Color 427
- Comparable 347, 397
- Comparator 666
- compare 666
- compareTo 357, 366, 372, 380, 397
- Compiler 27, 37, 713
- Component 423
 - componentAdded 491
 - ComponentEvent 486
 - componentHidden 491
 - ComponentListener 490
 - componentMoved 490
 - componentRemoved 491
 - componentResized 490
 - componentShown 491
- Computer 23, 713
- Computersystem 24
- Consumer 663
- Container 411, 412, 458, 460, 461
- Container 418, 424
- ContainerEvent 486

- ContainerListener 491
- contains 393
- containsAll 393
- Content-Pane 418
- continue** 94
- copy 451, 452, 650
- COPY_ATTRIBUTES 650
- Corba 692
- count 683
- countTokens 406
- CREATE 650
- createNewFile 585
- CTRL_MASK 467
- currentThread 551
- cut 451, 452
- CVS 689
- CYAN 427

- Dämon-Threads 560
- DARK_GRAY 427
- data hiding 195, 205
- DataInputStream 599, 600
- DataOutputStream 599, 600
- Date 380
- DateFormat 386
- Datei 24, 584, 713
 - Namen 36
 - Namen-Erweiterung 25, 36
- Datenbank 691, 713
- Datenkapselung 195
- Datentypen 34, 55, 714
 - einfache 55
 - ganzzahlige 55
 - generische 329, 638
 - Gleitkomma- 57
 - Referenz- 105
- Datumsangaben 379, 382, 386
- DAY_OF_MONTH 383
- DAY_OF_YEAR 383
- Deadlock 560, 714
- DecimalFormat 376
- default** 87, 672
- Default-Konstruktor 218
- Default-Methoden 672
- Default-Werte 225
- DeflaterOutputStream 604
- Deklaration 65
 - von Methoden 151
 - von Variablen 34
- Dekrementoperator 75
- Delegation Event Model 474
- delete 354, 585
- DELETE_ON_CLOSE 650
- deleteCharAt 354
- deleteIfExists 650
- Deserialisierung 601
- destroy 527
- Diamond-Operator 639
- Dienst 612
- directory 25
- dispose 460-462
- DISPOSE_ON_CLOSE 461
- distinct 683
- divide 366, 371
- Division 57
- DNS 610, 714
- do** 91
- DO_NOTHING_ON_CLOSE 461
- Domain Name Service 610
- Domain-Namen 610
- Doppelklicken 714
- double** 58
- Double 357
- DoubleBuffer 604
- doubleValue 358
- Download 714
- drawArc 510
- drawLine 509
- drawOval 509
- drawPolygon 509
- drawPolyline 509
- drawRect 509
- drawString 510
- dreistellige Operatoren 67
- dyadische Operatoren 46, 67
- dynamisches Binden 260

- E/A-Ströme 583
- EAST 433
- Editor 28, 714
- effektiv final 671
- einfache Datentypen 55
- Eingabe 583, 587, 598, 603
- Eingabegeräte 24
- Eingabestrom 701
- einstellige Operatoren 67

- EJB **691**
- Elementklasse **146**
- else** **86**
- Elternklasse **251**
- Entscheidungsanweisung **86**
- Entwicklungsumgebung **689**
- Entwurfsmuster **199**
- equals **263, 348, 357, 366, 372, 380, 397, 636**
- equals-Vertrag **263**
- erben **192**
- Ereignis **412, 473**
- Ereignisempfänger **474**
- Ereignisquellen **474**
- Ereignisverarbeitung **473, 481**
- Ergebnisrückgabe **152**
- Ergebnistyp **68, 151**
- Error **317**
- Error message **296**
- erweitern **193**
- Erweiterung **191**
 - Dateinamen- **25, 36**
- Erzeuger/Verbraucher-Problem **562**
- Escape-Sequenzen **59**
- Ethernet **608**
- Event **473**
- Event-Dispatching-Thread **579**
- EventListener **489**
- EventObject **486**
- Exception **642**
- Exception **296, 302**
- exists **585**
- exit **561**
- EXIT_ON_CLOSE **416, 461**
- exklusives Oder **71**
- explizite Typkonvertierung **61**
- Exponentenschreibweise **33, 58**
- extends** **194, 251**
- externer Speicher **24, 714**

- false** **60**
- Farben **412, 427**
- FDDI **608**
- Fehler **28**
- Fehlermeldung **46, 56, 296**
- Felder **107, 109**
 - flache Kopie **119, 127, 145**
 - Index **109**

- Initialisierer **112**
- Komponenten **109**
- Kopie **158**
- Länge **113**
- mehrdimensionale **121, 124**
- Referenzkopie **118, 126, 144**
- Tiefenkopie **119, 127, 144, 145**
- von Feldern **122**
- Zeile **121**

- Feldinitialisierer **112**
- Feldkomponenten **109**
- Feldlänge **113**
- Fenster **413, 415**
- Fiber Distributed Data Interface **608**
- file **24**
- File **584**
- File Transfer Protocol **608**
- FileChannel **604**
- FileInputStream **599**
- FileOutputStream **599**
- FileReader **300, 588**
- Files **649, 681**
- FileWriter **588**
- fillArc **510**
- fillOval **510**
- fillPolygon **510**
- fillRect **510**
- filter **683**
- final** **66, 214, 255, 261**
- final, effektiv **671**
- finally** **316**
- find **681**
- first **397**
- flache Kopie **119, 127, 145**
- float** **58**
- Float **357**
- FloatBuffer **604**
- Floating point numbers **33**
- floatValue **358**
- FlowLayout **431**
- flush **588, 599**
- FocusEvent **486**
- focusGained **490**
- FocusListener **490**
- focusLost **490**
- Fokus **439**
- Font **429**
- Fonts **412, 429**

- for** 89
- forEach 666, 683
- formale Argumente 151
- formale Parameter 151
- format 376, 387
- Format 386
- Format 376
- formatierte Ausgabe 376, 378, 386
- Formeln 32
- Frame 413
- Freeware 87, 715
- FTP 608
- FULL 389
- Function 663
- Funktion 149
- funktionale Interfaces 661

- ganze Zahlen 33
- Ganzzahlen, lange 363
- ganzzahliger Anteil 57
- Garbage-Collector 715
- gcd 366
- Generalisierung 190
- generate 681, 682
- generische Datentypen 329, 392, 638
- generische Klassen 332
- generische Methoden 339
- gepufferte Ströme 591
- get 383, 400, 648, 664
- get-Methoden 208
- getActionCommand 488
- getAppletContext 536
- getBackground 423
- getByName 610
- getClickCount 488
- getCodeBase 540, 626
- getComponents 425
- getContentPane 460–462
- getDateInstance 389
- getDateTimeInstance 389
- getDocumentBase 540, 626
- getFirstIndex 488
- getFont 423
- getForeground 423
- getHeight 423, 509
- getHost 627
- getHostAddress 610
- getHostName 610
- getIcon 436
- getInputStream 614
- getInsets 509
- getInstance 382
- getItem 465, 488
- getItemAt 446
- getItemCount 446, 466
- getKeyStroke 467
- getLastIndex 488
- getLineCount 454
- getLineWrap 454
- getMaxSelectionIndex 448
- getMenu 465
- getMenuCount 465
- getMinSelectionIndex 448
- getName 551, 585
- getOutputStream 614
- getParameter 535
- getPriority 551, 560
- getSelectedIndex 446
- getSelectedIndices 449
- getSelectedItem 446
- getSelectedText 451
- getSelectedValues 449
- getSelectionMode 449
- getSource 486
- getStateChange 488
- getText 436, 451
- getThreadGroup 551
- getTime 380, 382
- getTimeInstance 389
- getToolTipText 425
- getWidth 423, 509
- getWindow 488
- getWrapStyleWord 454
- getX 488
- getY 488
- Gibibyte 24
- Gigabyte 24
- Gleitkommadatentypen 57
- Gleitkommazahlen 33, 57, 58
 - lange 367
- Grafik-Koordinaten 508
- grafische Darstellung 507
- grafische Oberfläche 409, 411
- Grammatik 715
- Graphical User Interface 411
- Graphics 509

- GRAY 427
- GREEN 427
- GregorianCalendar 382
- Gregorianischer Kalender 715
- GridLayout 434
- Gruppen, Thread- 561
- GUI 411, 715
- Gültigkeitsbereich 85
- GZIPInputStream 604
- GZIPOutputStream 604

- Hacker 716
- Hardware 24
- hashCode 264, 348
- HashCode 264
- HashSet 395
- Hashtabellen 264
- hasMoreElements 406
- hasMoreTokens 406
- hasNext 394
- Hauptmethode 36, 50, 168
- Hauptprogramm 168
- Hauptroutine 151
- headSet 398
- heavyweight 419
- hexadezimale Zahlen 57, 634, 716
- HIDE_ON_CLOSE 461
- Hilfsklassen 349
- höhere Programmiersprache 26
- HORIZONTAL 467
- HORIZONTAL_SCROLLBAR_ALWAYS 456
- HORIZONTAL_SCROLLBAR_AS_NEEDED 456
- HORIZONTAL_SCROLLBAR_NEVER 456
- Host 716
- Host-Namen 610
- HOURL_OF_DAY 383
- HTML 521, 523, 533, 716
- HTTP 608, 716
- Hüll-Klassen 265, 355
 - Autoboxing 361
 - Autounboxing 361
 - Boxing 360
 - Unboxing 360
- Hyper Text Markup Language 521
- Hypertext Transfer Protocol 608

- I/O-Stream 583
- ICANN 610

- Icon 437
- IDE 689
- if 86
- Ikonisieren 414
- ImageIcon 437, 542
- imperative Programmierung 188, 717
- implements 267
- implizite Typkonvertierung 60, 155, 254
- implizite Typumwandlung 60, 155, 254
- import 47
- Index 109
- indexOf 400
- indizierte Variablen 109
- InetAddress 610
- Infix 67
- InflaterInputStream 604
- Informatik 26
- init 527
- Initialisierer, statische 222
- Initialisierung 65, 217
- Inkrementoperator 75
- innere Klasse 133, 277, 475, 481
- input stream 701
- InputStream 584, 598
- InputStreamReader 588
- insert 353
- Insets 509
- instanceof 263
- Instanziierung 135, 217
- Instanz 133
- Instanzmethoden 176, 195, 205, 206
- Instanzvariablen 133
- int 56
- IntBuffer 604
- integer 33
- Integer 357
- Interfaces 266, 267
 - funktionale 661
- Internet 25
- Internetprotokoll 608
- Interpreter 27, 37, 717
- Interpunktionszeichen 45, 46
- interrupt 551, 556
- interrupted 552
- Intranet 25
- IntStream 681
- intValue 358
- invalidate 515

- invokeAndWait 580
- invokeLater 580
- IOTools 596
- IP **608, 609, 717**
- IP-Adresse **610, 717**
- isAlive 551, 559
- isDaemon 551, 561
- isDirectory 585
- isEditable 446, 451
- isEmpty 393
- isEnabled 423
- isFile 585
- isFocusPainted 439
- isInterrupted 551, 556
- isModal 463
- isOpaque 425
- isSelected 439
- isSelectedIndex 449
- isSelectionEmpty 449
- isVisible 424
- IT **717**
- ITALIC 429
- ItemEvent 487
- ItemListener 489
- itemStateChanged 489
- Iterable 393, 672
- iterate 681, 682
- iterator 393, 394
- Iterator 394
- Iterator 278

- JApplet 523, 526
- Jar
 - Archive 534
- JAR-Datei **717**
- Java
 - Bytecode 27, 37
 - Compiler 37
 - Development Kit 37
 - Interpreter 27, 37
 - Version 7 **633**
 - Version 8 **655**
- Java community process **323**
- Java Community Process 633
- Java Foundation Classes **409, 412**
- Java specification request **323**
- Java Specification Request 633
- Java-Plug-in 525

- java.awt 412, 426
- java.awt.event 466, 467, 486
- java.io 300, 584, 593, 603
- java.lang 174, 274, 349, 352
- java.math 363
- java.net 537, 607, 610, 613, 626
- java.nio 604, 648
- java.nio.file 648
- java.text 376, 386
- java.util 278, 379, 389, 391, 402, 404, 406, 486
- java.util.functions 663
- java.util.stream 676
- javadoc 42
- javax.swing 412, 425, 467
- javax.swing.event 486
- javax.swing.text 451
- JButton 440
- JCheckBox 442
- JComboBox 445
- JComponent 425
- JCP **323, 633**
- JDBC **691**
- JDialog 461
- JDK 37
- JDO **691**
- JEE **691**
- JFC **409, 412**
- JFrame 415, 460
- JLabel 417, 438
- JList 448
- JMenuBar 465
- JNI **691**
- join 568
- JPA **691**
- JPanel 458
- JPasswordField 452
- JRadioButton 443
- JScrollPane 456
- JSDK 37
- JSR **323, 633**
- JTextArea 454
- JTextComponent 451
- JTextField 452
- JToggleButton 441
- JToolBar 467
- JWindow 461

- Kapselung **195**
- KeyEvent **486**
- KeyListener **490**
- keyPressed **490**
- keyReleased **490**
- keyTyped **490**
- Kibibyte **24, 70**
- Kilobyte **24, 70**
- Kindklassen **251**
- Klapptafeln **445**
- Klassen **35, 36, 50, 132, 133, 187**
 - abstrakte **251, 266**
 - Adapter- **492**
 - anonyme **282, 481**
 - auslagern **145**
 - Diagramm **134, 199**
 - Element- **146**
 - Eltern- **251**
 - Exception- **296**
 - generische **332**
 - Hüll- **265, 355**
 - innere **133, 277, 481**
 - Instantiierung **135**
 - Instanz **133**
 - Instanzvariablen **133**
 - Kind- **251**
 - Komponentenvariablen **133**
 - Methoden **173, 212**
 - Namen **36**
 - Referenz **138**
 - Sub- **191, 249, 251**
 - Super- **191, 251**
 - Top-Level- **146**
 - Variablen **134, 212**
 - Wrapper- **265, 355**
- Klassendiagramm **134, 199**
- Klassenkonstanten **214**
- Klassenmethoden **173, 212**
- Klassenvariablen **134, 212**
- Klicken **717**
- Knöpfe **438, 440–443**
- Kommandozeilenparameter **169**
- Kommentare **41, 45**
 - mit javadoc **42**
- Kommunikation, Thread- **561**
- Kompatibilität **717**
- Komponenten **411, 412**
 - grafische Darstellung **507**
 - statische **212**
- Komponentenvariablen **133**
- Komposition **197**
- Konkatenation, String- **350**
- Konsole **51**
- Konsolenfenster **34, 718**
- Konstanten
 - Klassen- **215**
 - Literal- **44, 633, 634**
 - symbolische **66, 215**
 - Zeichenketten- **349**
- konstanter Ausdruck **70**
- Konstruktoren **218**
 - Default- **218**
 - Standard- **218**
 - Überladen **219, 220**
- Koordinatensystem **508**
- Kopie
 - Feld- **158**
 - flache **119, 127, 145**
 - mit clone **119**
 - Referenz- **158**
 - Tiefen- **119, 127, 144, 145**
- Labels **417, 438**
- Lambda-Ausdrücke **655, 659**
- Länge eines Feldes **113**
- Langzahlen **363, 367**
- last **397**
- lastIndexOf **400**
- Layout **418**
- Layout-Manager **412, 430**
- LayoutManager **430**
- Lebenszyklus, Thread- **558**
- Leere Anweisung **85**
- Leerzeichen **45**
- LEFT **431, 436, 452**
- leichtgewichtige Prozesse **550**
- length **354, 585**
- Leser/Schreiber-Problem **562**
- LIGHT_GRAY **427**
- lightweight **420**
- limit **683**
- line feed **35**
- lines **681**
- LinkedList **400**
- list **585, 681**
- List **391, 400**

- Liste 399
- Listener 412, 489
 - Registrierung 494
- ListSelectionEvent 487, 489
- ListSelectionListener 489
- ListSelectionModel 449
- Literale 44
- Literalkonstanten 44, 58–60, 633, 634
 - null 44
- Locale 389
- logische Aussagen 60
- logische Operatoren 73
 - Oder 71, 74, 644
 - Und 71, 74
- long 56**
- Long 357
- LONG 389
- long integer 33
- LongStream 682
- longValue 358
- Look and Feel 409, 652, 718

- MAGENTA 427
- main 36, 50, 151, 168
- map 683
- Marke 93
- markierte Anweisungen 93
- Maschinensprache 26, 718
- Math 174
- max 366, 372
- MAX_PRIORITY 560
- MAX_VALUE 359
- Maximieren 414
- Mebibyte 24
- MEDIUM 389
- Megabyte 24
- Mehrdimensionale Felder 121, 124
- Mehrfachvererbung 268
- Menü 465
- Menüleisten 465
- Menge 395
- menuDeselected 491
- MenuEvent 486, 487
- MenuListener 491
- menuSelected 491
- Message 296
- META_MASK 467
- Methoden 34, 36, 149, 150
 - Argument 151
 - Aufruf 154
 - Call by value 154
 - Default- 672
 - Deklaration 151
 - dynamisches Binden 260
 - Ergebnisrückgabe 152
 - generische 339
 - get- 208
 - Instanz- 176, 195, 205, 206
 - Klassen- 173, 212
 - Kopf 151
 - Name 151
 - Parameter 151
 - Referenzen 668
 - rekursive 163
 - Rückgabetypp 152
 - Rumpf 151
 - set- 208
 - statische 212, 672
 - synchronisierte 564
 - terminieren 165
 - Überladen 156
 - Überschreiben 196, 258
 - variable Argumente 157, 405
 - Wertaufruf 154
- MILLISECOND 383
- min 366, 372
- MIN_PRIORITY 560
- MIN_VALUE 359
- Minimieren 414
- MINUTE 383
- mkdir 585
- modal 462
- Modell 187
- Modellierung 28, 187
- Modellierungsphase 198
- Modifikatoren 276
- monadische Operatoren 46, 67
- Monitor 565
- MONTH 383
- mouseClicked 490
- mouseDragged 490
- mouseEntered 490
- MouseEvent 486
- mouseExited 490
- MouseListener 490
- MouseMotionListener 490

- mouseMoved 490
- mousePressed 490
- mouseReleased 490
- move 650
- multiply 366, 371

- Namen 43
 - Datei- 36
 - für Threads 558
 - Klassen- 36
 - Methoden- 151
- NaN 359
- negate 366, 371
- Negation 70
- NEGATIVE_INFINITY 359
- net 25
- Netz 25
- Netzwerk 607
- Netzwerk-Programmierung 607
- Netzwerkschicht 608
- new** 111, 135
- newInputStream 650
- newLine 592
- newOutputStream 650
- newPriority 551
- next 394
- nextElement 406
- nextToken 406, 593
- nextTokens 407
- Nimbus** 652
- NOFOLLOW_LINKS 650
- NORM_PRIORITY 560
- NORTH 433
- Notation 67
 - Infix 67
 - Postfix 67
 - Präfix 67
- notify 559, 569
- notifyAll 559, 569
- Null-Literal** 44
- Null-Referenz** 141
- Nullstellen 373
- NumberFormat 376

- Oberfläche, grafische** 409, 411
- Object 569
- Object 262
- ObjectInputStream 601
- ObjectOutputStream 601

- Objekte 133, 187, 718
- objektorientiert 35
- objektorientierte Datenbank 691
- objektorientierte Programmierung 190, 719

- Oder
 - bedingtes logisches 74
 - exklusives 71
 - logisches 71, 74, 644
- of 681
- oktale Zahlen 57, 634, 719
- OOP** 719
- Open Source 87, 690, 719
- openConnection 626
- openStream 626
- Operand 67
- Operationen
 - Abschluss- 680, 683, 684
 - Pipeline- 676, 679
 - Stream- 679
 - Zwischen- 679, 682
- Operatoren 45, 46, 67
 - abkürzende Schreibweise 72
 - arithmetische 68
 - Auswertungsreihenfolge 76
 - binäre 67
 - Bit- 70
 - Dekrement- 75
 - Diamond 639
 - dreistellige 67
 - dyadische 46, 67
 - einstellige 67
 - Inkrement- 75
 - logische 73
 - monadische 46, 67
 - Notation 67
 - Prioritäten 76
 - Reihenfolge 67
 - Schiebe- 71
 - ternäre 67
 - triadische 46, 67
 - unäre 67
 - Vergleichs- 73
 - Zuweisungs- 65, 72
 - zweistellige 67
- Operatorsymbole 46
- ORANGE 427
- Ordner** 25

- OutOfMemoryError 318
- OutputStream 584, 599
- OutputStreamWriter 588

- pack 460, 461, 463
- package 275**
- paint 508
- paintBorder 508
- paintChildren 508
- paintComponent 508
- Pakete 274**
 - java.awt 412, 426
 - java.awt.event 466, 467, 486
 - java.io 300, 584, 593, 603
 - java.lang 174, 274, 349, 352
 - java.math 363
 - java.net 537, 607, 610, 613, 626
 - java.nio 604, 648
 - java.nio.file 648
 - java.text 376, 386
 - java.util 278, 379, 389, 391, 402, 404, 406, 486
 - java.util.functions 663
 - java.util.stream 676
 - javax.swing 412, 425, 467
 - javax.swing.event 486
 - javax.swing.text 451
 - Prog1Tools 269, 274, 702
- Paradigmen 188**
- parallele Programmierung 547
- parallelStream 681
- Parameter**
 - aktueller 154
 - formaler 151
 - Kommandozeilen- 169
- Parameterliste 151, 152**
- parse 389
- parseByte 358
- parseDouble 358
- parseFloat 358
- parseInteger 358
- parseLong 358
- parseShort 358
- paste 451
- Path 648
- Paths 648
- Peer 419**
- Peripheriegeräte 24**

- Philosophenproblem 572
- physikalische Schicht 608
- PI 215
- PINK 427
- Pipeline-Operationen 676, 679
- PLAIN 429
- Polymorphismus 196, 249, 254
- Popup-Menü 467
- Port 611**
- Portabilität 720**
- POSITIVE_INFINITY 359
- Postfix 67
- pow 366
- pow 174
- präemptives Scheduling 560
- Präfix 67
- Predicate 664
- print 589, 595
- printf 378
- println 34, 35, 589, 595
- PrintStream 603
- PrintWriter 595
- Prioritäten 47**
 - der Operatoren 76
 - von Threads 560
- Problemanalyse 28**
- problemorientierte Programmiersprache 26**
- Prog1Tools 269, 274, 702
- Programm 24**
- Programmieren 27**
- Programmiersprache 26**
- Programmierung**
 - Client/Server- 607
 - imperative 188
 - Netzwerk- 607
 - objektorientierte 190
 - parallele 547
- Prompt 79, 703
- protected 276**
- Protokoll 608, 720**
- Prozesse, leichtgewichtige 550
- Prozessor 24, 720
- Pulldown-Menü 465

- Quellcode 27, 720
- Quellprogramm 27
- Quelltext 27, 720

- Quicksort 165

- Rahmen 413, 415
- RAM 24, 721
- RandomAccessFile 604
- range 681, 682
- rangeClosed 682
- RCS 689
- read 300, 587, 588, 598, 599
- readBoolean 600
- readByte 600
- readChar 600, 702
- readDouble 600, 702
- Reader 584, 587
- readFloat 600
- readInt 600, 702
- readInteger 702
- readLine 591
- readLong 600
- readObject 601
- readShort 600
- RED 427
- reduce 683
- Referenz 106, 118, 138
 - Null- 141
- Referenzdatentypen 105, 117, 158
- Referenzen
 - Methoden- 668
- Referenzkopie 118, 126, 144, 158
- Regel, Syntax- 39, 40
- Registrierung eines Listeners 494
- Rekursion 163
 - Nachteile 165
 - Vorteile 164
- rekursive Methoden 163
- relationale Datenbank 691
- remainder 366
- remove 393, 394, 400, 425, 444, 466
- removeAll 393, 466
- removeAllItems 446
- removeItem 446
- removeItemAt 446
- renameTo 585
- repaint 507
- Repaint-Manager 507
- replace 354
- REPLACE_EXISTING 650
- replaceAll 666

- Reservierte Wörter 44
- Rest 57
- Resultatstyp 151
- retainAll 393
- return** 95, 152
- revalidate 516
- RGB-Farbmodell 427, 721
- RIGHT 431, 436, 452
- round 174
- ROUND_CEILING 371
- ROUND_DOWN 372
- ROUND_FLOOR 372
- ROUND_HALF_DOWN 372
- ROUND_HALF_EVEN 372
- ROUND_HALF_UP 372
- ROUND_UNNECESSARY 372
- ROUND_UP 372
- Routinen 150
- Routing 609
- RTFM 721
- Rückgabetypp 151, 152
- Rumpf
 - einer Methode 151
 - einer Schleife 89
- run 549–551
- Rundungsfehler 58
- Runnable 550, 554
- RuntimeException 302

- Sandkasten-Prinzip 538
- Scanner 53, 597
- Schaltflächen 438, 440–443
- Scheduler 559, 560, 721
- Scheduling bei Threads 560
- Schiebeoperatoren 71
- Schleifen 89, 90, 127
 - abweisende 91
 - do** 91
 - Endlos- 92
 - for** 89
 - nicht-abweisende 91
 - Rumpf 89
 - unendliche 92
 - vereinfachte Notation 170, 393, 395
 - while** 91
- Schließen 414
- Schlüsselwörter 44
 - abstract** 250

- assert** 320
- catch** 299
- default** 87, 672
- extends** 194, 251
- final** 66, 214, 255, 261
- finally** 316
- implements** 267
- protected** 276
- static** 134, 213
- super** 258
- synchronized** 564
- this** 207
- catch** 298
- throw** 298, 306
- throws** 301
- transient** 601
- try** 298
- Schnittstellen 195, 266
- SECOND 383
- Seiteneffekt 158
- Semantik 721
- semantische Ereignisse 486
- Semikolon 32, 46
- Sequenzdiagramm 199
- Serialisierung 601
- Serializable 601
- Server 612, 722
- Server-Host 612
- Server-Rechner 612
- ServerSocket 613
- Servlets 691
- set 382, 383, 400
- Set 391, 395
- set-Methoden 208
- setAccelerator 466
- setActionCommand 480
- setBackground 424
- setCharAt 354
- setDaemon 551
- setDefaultCloseOperation 416, 460, 463
- setEditable 446, 451
- setEnabled 424
- setFocusPainted 439
- setFont 424
- setForeground 424
- setHorizontalAlignment 436, 452
- setHorizontalScrollBarPolicy 456
- setHorizontalTextPosition 437
- setIcon 436
- setJMenuBar 460, 463
- setLayout 418, 425
- setLineWrap 454
- setLocation 424
- setMnemonic 466
- setModal 463
- setName 551
- setOpaque 425
- setPriority 560
- setSelected 439
- setSelectedIndex 446, 449
- setSelectedIndices 449
- setSelectedItem 446
- setSelectionMode 449
- setSize 413, 415, 424
- setSoTimeout 624
- setText 436, 451
- setTime 380, 382
- setTimeInMillis 384
- setTitle 413, 415, 460, 463
- setToolTipText 425
- setVerticalAlignment 436
- setVerticalScrollBarPolicy 456
- setVerticalTextPosition 437
- setVisible 413, 415, 424
- setWrapStyleWord 454
- SHIFT_MASK 467
- short** 56
- Short 357
- SHORT 389
- shortValue 358
- showConfirmDialog 464
- showDocument 536
- showInputDialog 464
- showMessageDialog 464
- Sicherheitseinschränkungen 538
- Sichtbarkeit 160
- signierte Applets 539
- Simple Mail Transfer Protocol 608
- SimpleDateFormat 386
- SINGLE_SELECTION 449
- size 393
- sleep 552, 559
- SMTP 608

- SOAP 692
- Socket 612
- Socket 613, 624
- SocketChannel 604
- Software 24
- sort 403, 404, 666
- sorted 683
- SortedSet 397
- Sortieren 402
- Sourcecode 722
- SOUTH 433
- Speicher, externer 24
- Speicherkapazität 24
- Speicherzelle 24
- Sperre 565
- Spezialisierung 191
- Sprungbefehle 93
- sqrt 174
- Standard-Ausgabe 375
- Standard-Konstruktor 218
- Standardwerte 225
- start 527, 548–551, 559
- Starvation 560
- stateChanged 489
- static** 134, 213
- static-Block** 222
- statische Importe 79, 175
- statische Initialisierer 222
- statische Komponenten 212
- statische Methoden 212, 672
- stop 527
- Ströme
 - gepufferte 591
- stream 681
- Stream 676, 681–683
- Stream-Operationen 679
- Streams 676, 679
- StreamTokenizer 593
- String 168, 177, 349, 636
 - Addition 68, 70
 - Konkatenation 68, 70
- StringBuffer 352
- StringTokenizer 406
- Subklassen 191, 249, 251
- subSet 398
- subtract 366, 371
- Suchen 402
- sum 684
- super** 258
- Superklassen 191, 251
- Supplier 664
- SVN 689
- Swing 409, 415, 420, 517
- SwingUtilities 497, 580, 654
- switch** 87, 636
- symbolische Konstanten 66, 215
- Synchronisation, Thread- 561
- Synchronisieren 722
- synchronisierte Methoden 564
- synchronized** 564
- Syntax 39, 40, 722
 - Regel 39, 40
- System 589
- Systemsoftware 24
- Tabulatorzeichen 45
- Tags 524
- tailSet 398
- Targets 554
- Tastatureingaben 701
- Tastaturfokus 439
- Tastaturkommandos 414
- TCP 608, 609, 722
- Telnet-Programm 617
- Terabyte 24
- terminieren 165
- ternäre Operatoren 67
- test 664
- Texteditor 28
- Textkomponenten 451, 452, 454
- this** 207
- Thread 550, 622
- Threads 547, 722
 - Applets 573
 - Dämon- 560
 - Deadlock 560
 - Frames 573
 - Gruppen 561
 - Kommunikation 561
 - Lebenszyklus 558
 - Namen 558
 - Scheduling 560
 - Sicherheit 579
 - Starvation 560
 - Synchronisation 561
 - vorzeitig beenden 556

- throw** 298, 306
- Throwable 317
- throws** 301
- Tibibyte 24
- Tiefenkopie 119, 127, 144, 145
- toArray 393
- Token 406
- Toolbars 467
- Tooltip 426, 723
- TOP 436
- Top-Level-Container 413, 415
- Top-Level-Klasse 146
- toString 211, 262, 353, 366, 372, 375, 380
- transient** 601
- Transmission Control Protocol 608
- Transportschicht 608
- TreeSet 398
- Trennzeichen 45
- triadische Operatoren 46, 67
- true** 60
- TRUNCATE_EXISTING 650
- try** 298
 - mit Ressourcen 646
- TT_EOF 593
- TT_EOL 593
- TT_NUMBER 593
- TT_WORD 593
- Typ 34, 68
 - Daten- 34
 - Ergebnis- 68
 - Rückgabe- 151, 152
- Typ-Parameter 329, 334, 639
- Typecast 60
- Typkonvertierung 60
 - automatische 60, 155, 254
 - explizite 61
 - implizite 60, 155, 254
- Typsicherheit bei Collections 392
- Typumwandlung 60
 - automatische 60, 155, 254
 - implizite 60, 155, 254
- Typ-Variable 332
- Typwandlung bei Wrapper-Klassen 360

- Überladen 156
 - von Konstruktoren 219, 220
 - von Methoden 155
- Überschreiben von Methoden 196, 258

- Übersetzer 26, 723
- UDP 608, 609, 723
- UIManager 497, 654
- Umgebungsvariable 723
- UML 199, 723
- Umlaute 43
- unäre Operatoren 67
- UnaryOperator 664
- Unboxing 360
- Und
 - bedingtes logisches 74
 - logisches 71, 74
- unendliche Streams 681, 682
- Unicode 59
 - Schreibweise 59
- Unified Modeling Language 199, 724
- Uniform Resource Locator 536
- Unterprogramm 149
- Unterstrich 43, 634
- Update 724
- URL 536, 724
- URL 537, 625, 626
- URLConnection 626
- use cases 198
- Use-Case-Diagramm 199
- User Datagram Protocol 608

- validate 515
- valueOf 357
- variable Methodenargumente 405
- Variablen 34, 63
 - Deklaration 34, 65
 - Gültigkeitsbereich 85
 - indizierte 109
 - Initialisierung 65
 - Instanz- 133
 - Klassen- 134, 212
 - Name 34, 65
- Variablendeklaration 34
- Verdecken 160
- vereinfachte Eingabe 597
- Vererbung 192, 249, 254
- Vergleichsoperatoren 73
- Verkettung, String- 350
- vernetzt 25
- Versionsverwaltungen 689
- Verteilungsdiagramm 199
- VERTICAL 467

- VERTICAL_SCROLLBAR_ALWAYS 456
- VERTICAL_SCROLLBAR_AS_NEEDED 456
- VERTICAL_SCROLLBAR_NEVER 456
- Verzeichnis 25, 584
- VirtualMachineError 318
- virtuelle Maschine 27
- void 151, 152, 168
- Vorzeichen 55

- Wahrheitswert 60
- wait 559, 568
- walk 681
- web 25
- Werkzeuggesteigen 467
- Wertaufruf 154
- Wertebereich 55
- WEST 433
- while** 91
- WHITE 427
- Wiederholungsanweisungen 89
- Wildcards 336
 - Bounded 337
- windowActivated 492
- WindowAdapter 493
- windowClosed 491
- windowClosing 491
- windowDeactivated 492
- windowDeiconified 492
- WindowEvent 487
- WindowFocusListener 492
- windowGainedFocus 492
- windowIconified 491
- WindowListener 491
- windowLostFocus 492
- windowOpened 491
- windowStateChanged 492
- WindowStateListener 492
- Workaround 724
- Wortschatz 724
- Wortsymbole 44
- Wrapper-Klassen 265, 355
 - Autoboxing 361
 - Autounboxing 361
 - Boxing 360
 - Typwandlung 360
 - Unboxing 360
- write 587, 588, 599
- WRITE 650
- writeBoolean 600
- writeByte 600
- writeChar 600
- writeDouble 600
- writeFloat 600
- writeInt 600
- writeLong 600
- writeObject 601
- Writer 584, 587
- writeShort 600

- XML 691, 724

- YEAR 383
- YELLOW 427
- yield 552, 559

- Zahlen
 - binäre 634
 - ganze 33
 - Gleitkomma- 33, 58
 - hexadezimale 57, 634
 - negative 56
 - oktale 57, 634
- Zeichen 59
- Zeichenketten-Literale 349
- Zeichenströme 584
- Zeichnen 507
- Zeile 121
- Zeilenendezeichen 45
- Zeilenvorschub 35
- Zeitangaben 379, 382, 386
- Zeitpunkte 380
- Zeitscheiben-Verfahren 560
- Zentraleinheit 24
- Zielprogramm 27
- Ziffern 43
- ZipInputStream 604
- ZipOutputStream 604
- Zugriffsmethoden 205
- Zugriffsrechte 205, 276
- Zusicherungen 320
- Zuweisung 65
- zuweisungskompatibel 153
- Zuweisungsoperator 65, 72
- Zweierkomplement 56
- zweistellige Operatoren 67
- Zwischenoperationen 679, 682