

Christian Ullenboom



- ✓ Inkl. vollständiger HTML-Version der Java-Insel
- ✓ 300 Aufgaben und Musterlösungen
- ✓ Java 7, Eclipse 3.7.1, NetBeans 7
- ✓ Viele Zusatzttools

Java ist auch eine Insel

Das umfassende Handbuch

Aktuell
zu
Java 7



- ▶ Programmieren mit der Java Platform, Standard Edition 7
- ▶ Java von A bis Z: Einführung, Praxis, Referenz
- ▶ Von Klassen und Objekten zu Datenstrukturen und Algorithmen

10., aktualisierte und überarbeitete Auflage

Galileo Computing

Christian Ullenboom

Java ist auch eine Insel

Das umfassende Handbuch

Liebe Leserin, lieber Leser,

da ist sie endlich: die neue Java-Insel!

Sie halten unsere Insel in Händen, das Grundlagenwerk, das Standardwerk, das in keinem Regal eines Java-Programmierers fehlen darf.

Was genau ist neu an dieser Auflage? Kurz: eine Menge!

Eine wichtige Änderung ist Folgende: Um Java zufriedenstellend und nahezu vollständig zu beschreiben, braucht man mehrere tausend Seiten. Viele Leser jedoch interessiert nicht der ganze Themenkomplex. Im Schwerpunkt gibt es Leser, die einen fundierten Einstieg in Java brauchen mit wirklich guten Beispielen. Das sind Studenten oder Auszubildende, deren erste Wahl meist Java ist. Eine zweite Gruppe bilden Industrieprogrammierer mit konkreten Fragen an einen speziellen Themenbereich. Das sind Leser, die ein umfangreiches Nachschlagewerk zu den immer größer werdenden Bibliotheken für die tägliche Arbeit brauchen. Diesem Umstand der zwei Leserschaften haben wir Rechnung getragen und hoffen, sie künftig mit zwei Titeln optimal versorgen zu können.

Sie besitzen hiermit das Werk, das Ihnen vollständig die Funktionsweise von Java erklärt. Möglicherweise kommen Sie gut damit aus, weil Sie nach Beendigung der Ausbildung gar nicht als Softwareentwickler arbeiten werden. Tun Sie dies aber doch, so empfehle ich Ihnen unser Handbuch zu den Java-Bibliotheken, »Java 7 – Mehr als eine Insel«. Das wären dann insgesamt 2700 Seiten Java, womit Sie mehr als gut gerüstet sind für die Zukunft.

Jetzt wünsche ich Ihnen erst einmal viel Freude beim Lesen unseres Kult-Buches. Sie haben jederzeit die Möglichkeit, Anregungen und Kritik loszuwerden. Zögern Sie nicht, sich an Christian Ullenboom (c.ullenboom@tutego.com) oder an mich zu wenden. Wir sind gespannt auf Ihre Rückmeldung!

Judith Stevens-Lemoine
Lektorat Galileo Computing

judith.stevens@galileo-press.de
www.galileocomputing.de
Galileo Press · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

1	Java ist auch eine Sprache	47
2	Imperative Sprachkonzepte	113
3	Klassen und Objekte	241
4	Der Umgang mit Zeichenketten	341
5	Eigene Klassen schreiben	475
6	Exceptions	615
7	Äußere.innere Klassen	691
8	Besondere Klassen der Java SE	709
9	Generics<T>	781
10	Architektur, Design und angewandte Objektorientierung	847
11	Die Klassenbibliothek	873
12	Einführung in die nebenläufige Programmierung	933
13	Einführung in Datenstrukturen und Algorithmen	971
14	Einführung in grafische Oberflächen	1013
15	Einführung in Dateien und Datenströme	1085
16	Einführung in die <XML>-Verarbeitung mit Java	1135
17	Einführung ins Datenbankmanagement mit JDBC	1175
18	Bits und Bytes und Mathematisches	1199
19	Die Werkzeuge des JDK	1251
A	Die Klassenbibliothek	1277

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendar ist sein Ausspruch *Eppur si muove* (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

Lektorat Judith Stevens-Lemoine, Anne Scheibe

Korrektorat Friederike Daenecke, Zülpich

Cover Barbara Thoben, Köln

Titelbild Getty Images/Photographer's Choice RF/Frank Krahmer

Typografie und Layout Vera Brauner

Herstellung Norbert Englert

Satz SatzPro, Krefeld

Druck und Bindung Bercker Graphischer Betrieb, Kevelaer

Gerne stehen wir Ihnen mit Rat und Tat zur Seite:

judith.stevens@galileo-press.de bei Fragen und Anmerkungen zum Inhalt des Buches

service@galileo-press.de für versandkostenfreie Bestellungen und Reklamationen

britta.behrens@galileo-press.de für Rezensions- und Schulungsexemplare

Dieses Buch wurde gesetzt aus der TheAntiqua (9,5/14 pt) in FrameMaker.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN 978-3-8362-1802-3

© Galileo Press, Bonn 2012

10., aktualisierte Auflage 2012

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischem oder anderen Wegen und der Speicherung in elektronischen Medien. Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Inhalt

Vorwort	29
---------------	----

1 Java ist auch eine Sprache

1.1 Historischer Hintergrund	47
1.2 Warum Java gut ist: die zentralen Eigenschaften	50
1.2.1 Bytecode	50
1.2.2 Ausführung des Bytecodes durch eine virtuelle Maschine	50
1.2.3 Plattformunabhängigkeit	52
1.2.4 Java als Sprache, Laufzeitumgebung und Standardbibliothek	53
1.2.5 Objektorientierung in Java	54
1.2.6 Java ist verbreitet und bekannt	55
1.2.7 Java ist schnell: Optimierung und Just-in-Time Compilation	55
1.2.8 Das Java-Security-Modell	57
1.2.9 Zeiger und Referenzen	57
1.2.10 Bring den Müll raus, Garbage-Collector!	59
1.2.11 Ausnahmebehandlung	59
1.2.12 Einfache Syntax der Programmiersprache Java	60
1.2.13 Java ist Open Source	62
1.2.14 Wofür sich Java weniger eignet	64
1.2.15 Java im Vergleich zu anderen Sprachen	65
1.2.16 Java und das Web, Applets und JavaFX	68
1.2.17 Features, Enhancements (Erweiterungen) und ein JSR	71
1.2.18 Die Entwicklung von Java und seine Zukunftsaussichten	72
1.3 Java-Plattformen: Java SE, Java EE und Java ME	73
1.3.1 Die Java SE-Plattform	73
1.3.2 Java für die Kleinen	75
1.3.3 Java für die ganz, ganz Kleinen	76
1.3.4 Java für die Großen	76
1.3.5 Echtzeit-Java (Real-time Java)	77
1.4 Die Installation der Java Platform Standard Edition (Java SE)	78
1.4.1 Die Java SE von Oracle	78

1.4.2	Download des JDK	79
1.4.3	Java SE unter Windows installieren	80
1.5	Das erste Programm compilieren und testen	84
1.5.1	Ein Quadratzahlen-Programm	84
1.5.2	Der Compilerlauf	86
1.5.3	Die Laufzeitumgebung	88
1.5.4	Häufige Compiler- und Interpreterprobleme	88
1.6	Entwicklungsumgebungen im Allgemeinen	89
1.6.1	Die Entwicklungsumgebung Eclipse	89
1.6.2	NetBeans von Oracle	90
1.6.3	IntelliJ IDEA	91
1.6.4	Ein Wort zu Microsoft, Java und zu J++, J#	91
1.7	Eclipse im Speziellen	92
1.7.1	Eclipse starten	94
1.7.2	Das erste Projekt anlegen	96
1.7.3	Eine Klasse hinzufügen	100
1.7.4	Übersetzen und ausführen	102
1.7.5	JDK statt JRE *	103
1.7.6	Start eines Programms ohne Speicheraufforderung	103
1.7.7	Projekt einfügen, Workspace für die Programme wechseln	104
1.7.8	Plugins für Eclipse	106
1.8	NetBeans im Speziellen	106
1.8.1	NetBeans-Bundles	106
1.8.2	NetBeans installieren	107
1.8.3	NetBeans starten	108
1.8.4	Ein neues NetBeans-Projekt anlegen	108
1.8.5	Ein Java-Programm starten	110
1.8.6	Einstellungen	110
1.9	Zum Weiterlesen	111

2 Imperative Sprachkonzepte

2.1	Elemente der Programmiersprache Java	113
2.1.1	Token	113
2.1.2	Textkodierung durch Unicode-Zeichen	115
2.1.3	Bezeichner	115

2.1.4	Literale	117
2.1.5	Reservierte Schlüsselwörter	118
2.1.6	Zusammenfassung der lexikalischen Analyse	119
2.1.7	Kommentare	120
2.2	Von der Klasse zur Anweisung	122
2.2.1	Was sind Anweisungen?	122
2.2.2	Klassendeklaration	123
2.2.3	Die Reise beginnt am main()	124
2.2.4	Der erste Methodenaufruf: println()	125
2.2.5	Atomare Anweisungen und Anweisungssequenzen	126
2.2.6	Mehr zu print(), println() und printf() für Bildschirmausgaben	126
2.2.7	Die API-Dokumentation	128
2.2.8	Ausdrücke	130
2.2.9	Ausdrucksanweisung	131
2.2.10	Erste Idee der Objektorientierung	132
2.2.11	Modifizierer	133
2.2.12	Gruppieren von Anweisungen mit Blöcken	134
2.3	Datentypen, Typisierung, Variablen und Zuweisungen	135
2.3.1	Primitive Datentypen im Überblick	136
2.3.2	Variablen Deklarationen	139
2.3.3	Konsoleneingaben	142
2.3.4	Fließkommazahlen mit den Datentypen float und double	144
2.3.5	Ganzzahlige Datentypen	146
2.3.6	Wahrheitswerte	148
2.3.7	Unterstriche in Zahlen *	148
2.3.8	Alphanumerische Zeichen	149
2.3.9	Gute Namen, schlechte Namen	150
2.3.10	Initialisierung von lokalen Variablen	151
2.4	Ausdrücke, Operanden und Operatoren	152
2.4.1	Zuweisungsoperator	152
2.4.2	Arithmetische Operatoren	154
2.4.3	Unäres Minus und Plus	158
2.4.4	Zuweisung mit Operation	159
2.4.5	Präfix- oder Postfix-Inkrement und -Dekrement	160
2.4.6	Die relationalen Operatoren und die Gleichheitsoperatoren	162
2.4.7	Logische Operatoren: Nicht, Und, Oder, Xor	164
2.4.8	Kurzschluss-Operatoren	166

2.4.9	Der Rang der Operatoren in der Auswertungsreihenfolge	167
2.4.10	Die Typanpassung (das Casting)	170
2.4.11	Überladenes Plus für Strings	175
2.4.12	Operator vermisst *	176
2.5	Bedingte Anweisungen oder Fallunterscheidungen	177
2.5.1	Die if-Anweisung	177
2.5.2	Die Alternative mit einer if-else-Anweisung wählen	180
2.5.3	Der Bedingungsoperator	184
2.5.4	Die switch-Anweisung bietet die Alternative	187
2.6	Schleifen	192
2.6.1	Die while-Schleife	193
2.6.2	Die do-while-Schleife	195
2.6.3	Die for-Schleife	197
2.6.4	Schleifenbedingungen und Vergleiche mit ==	201
2.6.5	Ausbruch planen mit break und Wiedereinstieg mit continue	204
2.6.6	break und continue mit Marken *	208
2.7	Methoden einer Klasse	212
2.7.1	Bestandteil einer Methode	213
2.7.2	Signatur-Beschreibung in der Java-API	215
2.7.3	Aufruf einer Methode	216
2.7.4	Methoden ohne Parameter deklarieren	217
2.7.5	Statische Methoden (Klassenmethoden)	218
2.7.6	Parameter, Argument und Wertübergabe	219
2.7.7	Methoden vorzeitig mit return beenden	221
2.7.8	Nicht erreichbarer Quellcode bei Methoden *	222
2.7.9	Methoden mit Rückgaben	223
2.7.10	Methoden überladen	227
2.7.11	Sichtbarkeit und Gültigkeitsbereich	230
2.7.12	Vorgegebener Wert für nicht aufgeführte Argumente *	232
2.7.13	Finale lokale Variablen	232
2.7.14	Rekursive Methoden *	233
2.7.15	Die Türme von Hanoi *	237
2.8	Zum Weiterlesen	240

3 Klassen und Objekte

3.1 Objektorientierte Programmierung (OOP)	241
3.1.1 Warum überhaupt OOP?	241
3.1.2 Denk ich an Java, denk ich an Wiederverwendbarkeit	242
3.2 Eigenschaften einer Klasse	243
3.2.1 Die Klasse Point	244
3.3 Die UML (Unified Modeling Language) *	244
3.3.1 Hintergrund und Geschichte der UML	245
3.3.2 Wichtige Diagrammtypen der UML	246
3.3.3 UML-Werkzeuge	247
3.4 Neue Objekte erzeugen	249
3.4.1 Ein Exemplar einer Klasse mit dem new-Operator anlegen	249
3.4.2 Garbage-Collector (GC) – Es ist dann mal weg	251
3.4.3 Deklarieren von Referenzvariablen	251
3.4.4 Zugriff auf Objektattribute und -methoden mit dem »..«	252
3.4.5 Überblick über Point-Methoden	257
3.4.6 Konstruktoren nutzen	261
3.5 ZZZZZnake	262
3.6 Kompilationseinheiten, Imports und Pakete schnüren	265
3.6.1 Volle Qualifizierung und import-Deklaration	266
3.6.2 Mit import p1.p2.* alle Typen eines Pakets erreichen	267
3.6.3 Hierarchische Strukturen über Pakete	268
3.6.4 Die package-Deklaration	269
3.6.5 Unbenanntes Paket (default package)	270
3.6.6 Klassen mit gleichen Namen in unterschiedlichen Paketen *	271
3.6.7 Compilationseinheit (Compilation Unit)	272
3.6.8 Statischer Import *	272
3.6.9 Eine Verzeichnisstruktur für eigene Projekte *	274
3.7 Mit Referenzen arbeiten, Identität und Gleichheit	274
3.7.1 Die null-Referenz	274
3.7.2 null-Referenzen testen	276
3.7.3 Zuweisungen bei Referenzen	278
3.7.4 Methoden mit nicht-primitiven Parametern	279
3.7.5 Identität von Objekten	284
3.7.6 Gleichheit und die Methode equals()	285

3.8 Arrays	287
3.8.1 Grundbestandteile	287
3.8.2 Deklaration von Arrays	288
3.8.3 Arrays mit Inhalt	289
3.8.4 Die Länge eines Arrays über das Attribut length auslesen	289
3.8.5 Zugriff auf die Elemente über den Index	290
3.8.6 Array-Objekte mit new erzeugen	292
3.8.7 Typische Feldfehler	293
3.8.8 Feld-Objekte als Parametertyp	294
3.8.9 Vorinitialisierte Arrays	295
3.8.10 Die erweiterte for-Schleife	296
3.8.11 Arrays mit nicht-primitiven Elementen	298
3.8.12 Mehrdimensionale Arrays *	301
3.8.13 Nichtrechteckige Arrays *	306
3.8.14 Die Wahrheit über die Array-Initialisierung *	309
3.8.15 Mehrere Rückgabewerte *	310
3.8.16 Methode mit variabler Argumentanzahl (Vararg)	311
3.8.17 Klonen kann sich lohnen – Arrays vermehren *	313
3.8.18 Feldinhalte kopieren *	314
3.8.19 Die Klasse Arrays zum Vergleichen, Füllen, Suchen, Sortieren nutzen	315
3.8.20 Eine lange Schlange	325
3.9 Der Einstiegspunkt für das Laufzeitsystem: main()	328
3.9.1 Korrekte Deklaration der Startmethode	328
3.9.2 Kommandozeilenargumente verarbeiten	329
3.9.3 Der Rückgabetyp von main() und System.exit() *	330
3.10 Annotationen und Generics	332
3.10.1 Generics	332
3.10.2 Annotationen	333
3.10.3 Eigene Metadaten setzen	333
3.10.4 Annotationstypen @Override, @Deprecated, @SuppressWarnings	334
3.11 Zum Weiterlesen	339

4 Der Umgang mit Zeichenketten

4.1 Von ASCII über ISO-8859-1 zu Unicode	341
4.1.1 ASCII	341

4.1.2	ISO/IEC 8859-1	342
4.1.3	Unicode	343
4.1.4	Unicode-Zeichenkodierung	345
4.1.5	Escape-Sequenzen/Fluchtsymbole	346
4.1.6	Schreibweise für Unicode-Zeichen und Unicode-Escapes	347
4.1.7	Unicode 4.0 und Java *	349
4.2	Die Character-Klasse	350
4.2.1	Ist das so?	350
4.2.2	Zeichen in Großbuchstaben/Kleinbuchstaben konvertieren	353
4.2.3	Ziffern einer Basis *	355
4.3	Zeichenfolgen	357
4.4	Die Klasse String und ihre Methoden	359
4.4.1	String-Literale als String-Objekte für konstante Zeichenketten	359
4.4.2	Konkatenation mit +	361
4.4.3	String-Länge und Test auf Leerstring	361
4.4.4	Zugriff auf ein bestimmtes Zeichen mit charAt()	362
4.4.5	Nach enthaltenen Zeichen und Zeichenfolgen suchen	363
4.4.6	Das Hangman-Spiel	367
4.4.7	Gut, dass wir verglichen haben	369
4.4.8	Phonetische Vergleiche *	373
4.4.9	String-Teile extrahieren	374
4.4.10	Strings anhängen, Groß-/Kleinschreibung und Leerraum	378
4.4.11	Suchen und ersetzen	381
4.4.12	String-Objekte mit Konstruktoren neu anlegen *	383
4.5	Konvertieren zwischen Primitiven und Strings	389
4.5.1	Unterschiedliche Typen in String-Repräsentationen konvertieren	389
4.5.2	Stringinhalt in einen primitiven Wert konvertieren	391
4.5.3	String-Repräsentation im Format Binär, Hex, Oktal *	393
4.6	Veränderbare Zeichenketten mit StringBuilder und StringBuffer	397
4.6.1	Anlegen von StringBuilder/StringBuffer-Objekten	398
4.6.2	StringBuilder/StringBuffer in andere Zeichenkettenformate konvertieren	400
4.6.3	Zeichen(folgen) erfragen	400
4.6.4	Daten anhängen	400
4.6.5	Zeichen(folgen) setzen, löschen und umdrehen	402
4.6.6	Länge und Kapazität eines StringBuilder/StringBuffer-Objekts *	404

4.6.7	Vergleichen von String mit StringBuilder und StringBuffer	405
4.6.8	hashCode() bei StringBuilder/StringBuffer *	407
4.7	CharSequence als Basistyp *	407
4.8	Reguläre Ausdrücke	409
4.8.1	Pattern.matches() bzw. String#matches()	410
4.8.2	Die Klassen Pattern und Matcher	413
4.8.3	Finden und nicht matchen	419
4.8.4	Gierige und nicht gierige Operatoren *	420
4.8.5	Mit MatchResult alle Ergebnisse einsammeln *	421
4.8.6	Suchen und Ersetzen mit Mustern	423
4.8.7	Hangman Version 2	425
4.9	Zerlegen von Zeichenketten	427
4.9.1	Splitten von Zeichenketten mit split()	427
4.9.2	Die Klasse Scanner	429
4.9.3	Die Klasse StringTokenizer *	436
4.9.4	BreakIterator als Zeichen-, Wort-, Zeilen- und Satztrenner *	438
4.10	Zeichenkodierungen, XML/HTML-Entitys, Base64 *	442
4.10.1	Unicode und 8-Bit-Abbildungen	442
4.10.2	Das Paket java.nio.charset und der Typ Charset	443
4.10.3	Konvertieren mit OutputStreamWriter/InputStreamReader-Klassen *	445
4.10.4	XML/HTML-Entitys ausmaskieren	446
4.10.5	Base64-Kodierung	447
4.11	Ausgaben formatieren	449
4.11.1	Formatieren und Ausgeben mit format()	449
4.11.2	Die Formatter-Klasse *	455
4.11.3	Formatieren mit Masken *	457
4.11.4	Format-Klassen	459
4.11.5	Zahlen, Prozente und Währungen mit NumberFormat und DecimalFormat formatieren *	462
4.11.6	MessageFormat und Pluralbildung mit ChoiceFormat	465
4.12	Sprachabhängiges Vergleichen und Normalisierung *	467
4.12.1	Die Klasse Collator	467
4.12.2	Effiziente interne Speicherung für die Sortierung	471
4.12.3	Normalisierung	473
4.13	Zum Weiterlesen	474

5 Eigene Klassen schreiben

5.1 Eigene Klassen mit Eigenschaften deklarieren	475
5.1.1 Attribute deklarieren	476
5.1.2 Methoden deklarieren	478
5.1.3 Die this-Referenz	483
5.2 Privatsphäre und Sichtbarkeit	487
5.2.1 Für die Öffentlichkeit: public	487
5.2.2 Kein Public Viewing – Passwörter sind privat	487
5.2.3 Wieso nicht freie Methoden und Variablen für alle?	489
5.2.4 Privat ist nicht ganz privat: Es kommt darauf an, wer's sieht *	490
5.2.5 Zugriffsmethoden für Attribute deklarieren	491
5.2.6 Setter und Getter nach der JavaBeans-Spezifikation	491
5.2.7 Paketsichtbar	494
5.2.8 Zusammenfassung zur Sichtbarkeit	496
5.3 Statische Methoden und statische Attribute	499
5.3.1 Warum statische Eigenschaften sinnvoll sind	499
5.3.2 Statische Eigenschaften mit static	500
5.3.3 Statische Eigenschaften über Referenzen nutzen? *	501
5.3.4 Warum die Groß- und Kleinschreibung wichtig ist *	502
5.3.5 Statische Variablen zum Datenaustausch *	503
5.3.6 Statische Eigenschaften und Objekteigenschaften *	505
5.4 Konstanten und Aufzählungen	505
5.4.1 Konstanten über öffentliche statische finale Variablen	506
5.4.2 Typ(un)sichere Aufzählungen *	507
5.4.3 Aufzählungen mit enum	509
5.5 Objekte anlegen und zerstören	513
5.5.1 Konstruktoren schreiben	513
5.5.2 Der vorgegebene Konstruktor (default constructor)	515
5.5.3 Parametrisierte und überladene Konstruktoren	517
5.5.4 Copy-Konstruktor	519
5.5.5 Einen anderen Konstruktor der gleichen Klasse mit this() aufrufen	521
5.5.6 Ihr fehlt uns nicht – der Garbage-Collector	525
5.5.7 Private Konstruktoren, Utility-Klassen, Singleton, Fabriken	527
5.6 Klassen- und Objektinitialisierung *	529
5.6.1 Initialisierung von Objektvariablen	530
5.6.2 Statische Blöcke als Klasseninitialisierer	532

5.6.3	Initialisierung von Klassenvariablen	534
5.6.4	Eincompilierte Belegungen der Klassenvariablen	534
5.6.5	Exemplarinitialisierer (Instanzinitialisierer)	535
5.6.6	Finale Werte im Konstruktor und in statischen Blöcken setzen	539
5.7	Assoziationen zwischen Objekten	541
5.7.1	Unidirektionale 1:1-Beziehung	542
5.7.2	Bidirektionale 1:1-Beziehungen	543
5.7.3	Unidirektionale 1:n-Beziehung	545
5.8	Vererbung	547
5.8.1	Vererbung in Java	548
5.8.2	Spielobjekte modellieren	548
5.8.3	Die implizite Basisklasse <code>java.lang.Object</code>	550
5.8.4	Einfach- und Mehrfachvererbung *	551
5.8.5	Die Sichtbarkeit <code>protected</code>	551
5.8.6	Konstruktoren in der Vererbung und <code>super()</code>	552
5.9	Typen in Hierarchien	559
5.9.1	Automatische und explizite Typanpassung	559
5.9.2	Das Substitutionsprinzip	561
5.9.3	Typen mit dem <code>instanceof</code> -Operator testen	563
5.10	Methoden überschreiben	566
5.10.1	Methoden in Unterklassen mit neuem Verhalten ausstatten	566
5.10.2	Mit <code>super</code> an die Eltern	570
5.10.3	Finale Klassen und finale Methoden	573
5.10.4	Kovariante Rückgabetypen	575
5.10.5	Array-Typen und Kovarianz *	576
5.11	Drum prüfe, wer sich ewig dynamisch bindet	577
5.11.1	Gebunden an <code>toString()</code>	578
5.11.2	Implementierung von <code>System.out.println(Object)</code>	580
5.11.3	Nicht dynamisch gebunden bei privaten, statischen und finalen Methoden	581
5.11.4	Dynamisch gebunden auch bei Konstruktoraufrufen *	583
5.11.5	Eine letzte Spielerei mit Javas dynamischer Bindung und überschatteten Attributen *	585
5.12	Abstrakte Klassen und abstrakte Methoden	587
5.12.1	Abstrakte Klassen	587
5.12.2	Abstrakte Methoden	589

5.13 Schnittstellen	593
5.13.1 Schnittstellen deklarieren	594
5.13.2 Implementieren von Schnittstellen	595
5.13.3 Markierungsschnittstellen *	597
5.13.4 Ein Polymorphie-Beispiel mit Schnittstellen	598
5.13.5 Die Mehrfachvererbung bei Schnittstellen *	599
5.13.6 Keine Kollisionsgefahr bei Mehrfachvererbung *	604
5.13.7 Erweitern von Interfaces – Subinterfaces	605
5.13.8 Konstantendeklarationen bei Schnittstellen	606
5.13.9 Initialisierung von Schnittstellenkonstanten *	609
5.13.10 Abstrakte Klassen und Schnittstellen im Vergleich	613
5.14 Zum Weiterlesen	614

6 Exceptions

6.1 Problembereiche einzäunen	615
6.1.1 Exceptions in Java mit try und catch	616
6.1.2 Eine NumberFormatException auffangen	617
6.1.3 Ablauf einer Ausnahmesituation	620
6.1.4 Eigenschaften vom Exception-Objekt	620
6.1.5 Wiederholung abgebrochener Bereiche *	622
6.1.6 Mehrere Ausnahmen auffangen	623
6.1.7 throws im Methodenkopf angeben	626
6.1.8 Abschlussbehandlung mit finally	627
6.2 RuntimeException muss nicht aufgefangen werden	633
6.2.1 Beispiele für RuntimeException-Klassen	634
6.2.2 Kann man abfangen, muss man aber nicht	635
6.3 Die Klassenhierarchie der Fehler	635
6.3.1 Die Exception-Hierarchie	636
6.3.2 Oberausnahmen auffangen	636
6.3.3 Schon gefangen?	638
6.3.4 Alles geht als Exception durch	639
6.3.5 Zusammenfassen gleicher catch-Blöcke mit dem multi-catch	641
6.4 Harte Fehler: Error *	645
6.5 Auslösen eigener Exceptions	646
6.5.1 Mit throw Ausnahmen auslösen	646

6.5.2	Vorhandene Runtime-Fehlertypen kennen und nutzen	649
6.5.3	Parameter testen und gute Fehlermeldungen	651
6.5.4	Neue Exception-Klassen deklarieren	653
6.5.5	Eigene Ausnahmen als Unterklassen von Exception oder RuntimeException?	655
6.5.6	Ausnahmen abfangen und weiterleiten *	657
6.5.7	Aufrufstack von Ausnahmen verändern *	659
6.5.8	Präzises rethrow *	660
6.5.9	Geschachtelte Ausnahmen *	664
6.6	Automatisches Ressourcen-Management (try mit Ressourcen)	667
6.6.1	try mit Ressourcen	668
6.6.2	Ausnahmen vom close() bleiben bestehen	669
6.6.3	Die Schnittstelle AutoCloseable	670
6.6.4	Mehrere Ressourcen nutzen	671
6.6.5	Unterdrückte Ausnahmen *	673
6.7	Besonderheiten bei der Ausnahmebehandlung *	676
6.7.1	Rückgabewerte bei ausgelösten Ausnahmen	677
6.7.2	Ausnahmen und Rückgaben verschwinden: Das Duo »return« und »finally«	677
6.7.3	throws bei überschriebenen Methoden	679
6.7.4	Nicht erreichbare catch-Klauseln	681
6.8	Den Stack-Trace erfragen *	682
6.8.1	StackTraceElement	683
6.8.2	printStackTrace()	684
6.8.3	StackTraceElement vom Thread erfragen	685
6.9	Assertions *	686
6.9.1	Assertions in eigenen Programmen nutzen	687
6.9.2	Assertions aktivieren	688
6.10	Zum Weiterlesen	690

7 Äußere.innere Klassen

7.1	Geschachtelte (innere) Klassen, Schnittstellen, Aufzählungen	691
7.2	Statische innere Klassen und Schnittstellen	692

7.3 Mitglieds- oder Elementklassen	694
7.3.1 Exemplare innerer Klassen erzeugen	695
7.3.2 Die this-Referenz	696
7.3.3 Vom Compiler generierte Klassendateien *	697
7.3.4 Erlaubte Modifizierer bei äußeren und inneren Klassen	698
7.3.5 Innere Klassen greifen auf private Eigenschaften zu	698
7.4 Lokale Klassen	700
7.5 Anonyme innere Klassen	701
7.5.1 Umsetzung innerer anonymer Klassen *	703
7.5.2 Nutzung innerer Klassen für Threads *	703
7.5.3 Konstruktoren innerer anonymer Klassen *	704
7.6 Zugriff auf lokale Variablen aus lokalen inneren und anonymen Klassen *	705
7.7 this in Unterklassen *	707

8 Besondere Klassen der Java SE

8.1 Vergleichen von Objekten	709
8.1.1 Natürlich geordnet oder nicht?	709
8.1.2 Die Schnittstelle Comparable	710
8.1.3 Die Schnittstelle Comparator	711
8.1.4 Rückgabewerte kodieren die Ordnung	711
8.1.5 Aneinanderreihung von Comparatoren *	713
8.2 Wrapper-Klassen und Autoboxing	718
8.2.1 Wrapper-Objekte erzeugen	719
8.2.2 Konvertierungen in eine String-Repräsentation	721
8.2.3 Die Basisklasse Number für numerische Wrapper-Objekte	722
8.2.4 Vergleiche durchführen mit compare(), compareTo(), equals()	724
8.2.5 Die Klasse Integer	727
8.2.6 Die Klassen Double und Float für Fließkommazahlen	729
8.2.7 Die Long-Klasse	730
8.2.8 Die Boolean-Klasse	731
8.2.9 Autoboxing: Boxing und Unboxing	732
8.3 Object ist die Mutter aller Klassen	737
8.3.1 Klassenobjekte	737
8.3.2 Objektidentifikation mit <code>toString()</code>	738

8.3.3	Objektgleichheit mit equals() und Identität	740
8.3.4	Klonen eines Objekts mit clone() *	745
8.3.5	Hashcodes über hashCode() liefern *	751
8.3.6	System.identityHashCode() und das Problem der nicht-eindeutigen Objektverweise *	757
8.3.7	Aufräumen mit finalize() *	761
8.3.8	Synchronisation *	764
8.4	Die Utility-Klasse java.util.Objects	764
8.5	Die Spezial-Oberklasse Enum	767
8.5.1	Methoden auf Enum-Objekten	767
8.5.2	enum mit eigenen Konstruktoren und Methoden *	771
8.6	Erweitertes for und Iterable	777
8.6.1	Die Schnittstelle Iterable	777
8.6.2	Einen eigenen Iterable implementieren *	778
8.7	Zum Weiterlesen	780

9 Generics<T>

9.1	Einführung in Java Generics	781
9.1.1	Mensch versus Maschine: Typprüfung des Compilers und der Laufzeitumgebung	781
9.1.2	Taschen	782
9.1.3	Generische Typen deklarieren	785
9.1.4	Generics nutzen	786
9.1.5	Diamonds are forever	789
9.1.6	Generische Schnittstellen	792
9.1.7	Generische Methoden/Konstruktoren und Typ-Inferenz	794
9.2	Umsetzen der Generics, Typlösung und Raw-Types	799
9.2.1	Realisierungsmöglichkeiten	799
9.2.2	Typlösung (Type Erasure)	800
9.2.3	Probleme aus der Typlösung	802
9.2.4	Raw-Type	807
9.3	Einschränken der Typen über Bounds	810
9.3.1	Einfache Einschränkungen mit extends	810
9.3.2	Weitere Obertypen mit &	813

9.4	Typparameter in der throws-Klausel *	814
9.4.1	Deklaration einer Klasse mit Typvariable <E extends Exception>	814
9.4.2	Parametrisierter Typ bei Typvariable <E extends Exception>	814
9.5	Generics und Vererbung, Invarianz	818
9.5.1	Arrays sind invariant	818
9.5.2	Generics sind kovariant	819
9.5.3	Wildcards mit ?	820
9.5.4	Bounded Wildcards	822
9.5.5	Bounded-Wildcard-Typen und Bounded-Typvariablen	826
9.5.6	Das LESS-Prinzip	829
9.5.7	Enum<E extends Enum<E>> *	832
9.6	Konsequenzen der Typlöschung: Typ-Token, Arrays und Brücken *	834
9.6.1	Typ-Token	834
9.6.2	Super-Type-Token	836
9.6.3	Generics und Arrays	837
9.6.4	Brückenmethoden	839

10 Architektur, Design und angewandte Objektorientierung

10.1	Architektur, Design und Implementierung	847
10.2	Design-Pattern (Entwurfsmuster)	848
10.2.1	Motivation für Design-Pattern	849
10.2.2	Das Beobachter-Pattern (Observer/Observable)	849
10.2.3	Ereignisse über Listener	856
10.3	JavaBean	861
10.3.1	Properties (Eigenschaften)	862
10.3.2	Einfache Eigenschaften	863
10.3.3	Indizierte Eigenschaften	863
10.3.4	Gebundene Eigenschaften und PropertyChangeListener	864
10.3.5	Veto-Eigenschaften – dagegen!	867
10.4	Zum Weiterlesen	872

11 Die Klassenbibliothek

11.1	Die Java-Klassenphilosophie	873
11.1.1	Übersicht über die Pakete der Standardbibliothek	873
11.2	Sprachen der Länder	876
11.2.1	Sprachen und Regionen über Locale-Objekte	876
11.3	Die Klasse Date	880
11.3.1	Objekte erzeugen und Methoden nutzen	880
11.3.2	Date-Objekte sind nicht immutable	882
11.4	Calendar und GregorianCalendar	883
11.4.1	Die abstrakte Klasse Calendar	883
11.4.2	Der gregorianische Kalender	885
11.4.3	Calendar nach Date und Millisekunden fragen	887
11.4.4	Abfragen und Setzen von Datumselementen über Feldbezeichner	888
11.5	Klassenlader (Class Loader)	892
11.5.1	Woher die kleinen Klassen kommen	892
11.5.2	Setzen des Klassenpfades	894
11.5.3	Die wichtigsten drei Typen von Klassenladern	895
11.5.4	Die Klasse java.lang.ClassLoader *	896
11.5.5	Hot Deployment mit dem URL-Classloader *	897
11.5.6	Das Verzeichnis jre/lib/endorsed *	901
11.6	Die Utility-Klasse System und Properties	902
11.6.1	Systemeigenschaften der Java-Umgebung	903
11.6.2	line.separator	905
11.6.3	Eigene Properties von der Konsole aus setzen *	905
11.6.4	Umgebungsvariablen des Betriebssystems *	908
11.6.5	Einfache Zeitmessung und Profiling *	910
11.7	Einfache Benutzereingaben	913
11.7.1	Grafischer Eingabedialog über JOptionPane	913
11.7.2	Geschützte Passwort-Eingaben mit der Klasse Console *	915
11.8	Ausführen externer Programme *	916
11.8.1	ProcessBuilder und Prozesskontrolle mit Process	917
11.8.2	Einen Browser, E-Mail-Client oder Editor aufrufen	922
11.9	Benutzereinstellungen *	924
11.9.1	Benutzereinstellungen mit der Preferences-API	924
11.9.2	Einträge einfügen, auslesen und löschen	926

11.9.3	Auslesen der Daten und Schreiben in einem anderen Format	929
11.9.4	Auf Ereignisse horchen	929
11.9.5	Zugriff auf die gesamte Windows-Registry	931
11.10	Zum Weiterlesen	932

12 Einführung in die nebenläufige Programmierung

12.1	Nebenläufigkeit	933
12.1.1	Threads und Prozesse	934
12.1.2	Wie parallele Programme die Geschwindigkeit steigern können	935
12.1.3	Was Java für Nebenläufigkeit alles bietet	937
12.2	Threads erzeugen	937
12.2.1	Threads über die Schnittstelle Runnable implementieren	938
12.2.2	Thread mit Runnable starten	939
12.2.3	Die Klasse Thread erweitern	941
12.3	Thread-Eigenschaften und -Zustände	944
12.3.1	Der Name eines Threads	944
12.3.2	Wer bin ich?	944
12.3.3	Schläfer gesucht	945
12.3.4	Mit yield() auf Rechenzeit verzichten	947
12.3.5	Der Thread als Dämon	948
12.3.6	Das Ende eines Threads	950
12.3.7	Einen Thread höflich mit Interrupt beenden	951
12.3.8	UncaughtExceptionHandler für unbehandelte Ausnahmen	953
12.4	Der Ausführer (Executor) kommt	954
12.4.1	Die Schnittstelle Executor	955
12.4.2	Die Thread-Pools	957
12.5	Synchronisation über kritische Abschnitte	958
12.5.1	Gemeinsam genutzte Daten	959
12.5.2	Probleme beim gemeinsamen Zugriff und kritische Abschnitte	959
12.5.3	Punkte parallel initialisieren	961
12.5.4	Kritische Abschnitte schützen	963
12.5.5	Kritische Abschnitte mit ReentrantLock schützen	966
12.6	Zum Weiterlesen	969

13 Einführung in Datenstrukturen und Algorithmen

13.1 Datenstrukturen und die Collection-API	971
13.1.1 Designprinzip mit Schnittstellen, abstrakten und konkreten Klassen	972
13.1.2 Die Basis-Schnittstellen Collection und Map	972
13.1.3 Die Utility-Klassen Collections und Arrays	973
13.1.4 Das erste Programm mit Container-Klassen	973
13.1.5 Die Schnittstelle Collection und Kernkonzepte	975
13.1.6 Schnittstellen, die Collection erweitern und Map	978
13.1.7 Konkrete Container-Klassen	981
13.1.8 Generische Datentypen in der Collection-API	982
13.1.9 Die Schnittstelle Iterable und das erweiterte for	983
13.2 Listen	983
13.2.1 Erstes Listen-Beispiel	984
13.3 Mengen (Sets)	987
13.3.1 Ein erstes Mengen-Beispiel	988
13.3.2 Methoden der Schnittstelle Set	991
13.4 Assoziative Speicher	992
13.4.1 Die Klassen HashMap und TreeMap	992
13.4.2 Einfügen und Abfragen der Datenstruktur	995
13.4.3 Über die Bedeutung von equals() und hashCode()	998
13.5 Mit einem Iterator durch die Daten wandern	998
13.5.1 Die Schnittstelle Iterator	999
13.5.2 Der Iterator kann (eventuell auch) löschen	1001
13.6 Algorithmen in Collections	1002
13.6.1 Die Bedeutung von Ordnung mit Comparator und Comparable	1004
13.6.2 Sortieren	1005
13.6.3 Den größten und kleinsten Wert einer Collection finden	1008
13.7 Zum Weiterlesen	1011

14 Einführung in grafische Oberflächen

14.1 Das Abstract Window Toolkit und Swing	1013
14.1.1 SwingSet-Demos	1013
14.1.2 Abstract Window Toolkit (AWT)	1014

14.1.3	Java Foundation Classes	1015
14.1.4	Was Swing von AWT unterscheidet	1018
14.1.5	GUI-Builder für AWT und Swing	1019
14.2	Mit NetBeans zur ersten Oberfläche	1020
14.2.1	Projekt anlegen	1020
14.2.2	Eine GUI-Klasse hinzufügen	1022
14.2.3	Programm starten	1024
14.2.4	Grafische Oberfläche aufbauen	1025
14.2.5	Swing-Komponenten-Klassen	1028
14.2.6	Funktionalität geben	1030
14.3	Fenster zur Welt	1033
14.3.1	Swing-Fenster mit javax.swing.JFrame darstellen	1034
14.3.2	Fenster schließbar machen – setDefaultCloseOperation()	1035
14.3.3	Sichtbarkeit des Fensters	1036
14.4	Beschriftungen (JLabel)	1037
14.5	Es tut sich was – Ereignisse beim AWT	1039
14.5.1	Die Ereignisquellen und Horcher (Listener) von Swing	1039
14.5.2	Listener implementieren	1041
14.5.3	Listener bei dem Ereignisauslöser anmelden/abmelden	1044
14.5.4	Adapterklassen nutzen	1046
14.5.5	Innere Mitgliedsklassen und innere anonyme Klassen	1048
14.5.6	Aufrufen der Listener im AWT-Event-Thread	1051
14.6	Schaltflächen	1051
14.6.1	Normale Schaltflächen (JButton)	1051
14.6.2	Der aufmerksame ActionListener	1054
14.7	Alles Auslegungssache: die Layoutmanager	1055
14.7.1	Übersicht über Layoutmanager	1055
14.7.2	Zuweisen eines Layoutmanagers	1056
14.7.3	Im Fluss mit FlowLayout	1057
14.7.4	BoxLayout	1059
14.7.5	Mit BorderLayout in alle Himmelsrichtungen	1060
14.7.6	Rasteranordnung mit GridLayout	1063
14.8	Textkomponenten	1065
14.8.1	Text in einer Eingabezeile	1066
14.8.2	Die Oberklasse der Text-Komponenten (JTextComponent)	1067

14.9 Zeichnen von grafischen Primitiven	1068
14.9.1 Die paint()-Methode für das AWT-Frame	1068
14.9.2 Die ereignisorientierte Programmierung ändert Fensterinhalte	1070
14.9.3 Zeichnen von Inhalten auf ein JFrame	1072
14.9.4 Linien	1073
14.9.5 Rechtecke	1074
14.9.6 Zeichenfolgen schreiben	1075
14.9.7 Die Font-Klasse	1076
14.9.8 Farben mit der Klasse Color	1079
14.10 Zum Weiterlesen	1083

15 Einführung in Dateien und Datenströme

15.1 Datei und Verzeichnis	1085
15.1.1 Dateien und Verzeichnisse mit der Klasse File	1086
15.1.2 Verzeichnis oder Datei? Existiert es?	1089
15.1.3 Verzeichnis- und Dateieigenschaften/-attribute	1090
15.1.4 Umbenennen und Verzeichnisse anlegen	1091
15.1.5 Verzeichnisse auflisten und Dateien filtern	1092
15.1.6 Dateien und Verzeichnisse löschen	1093
15.2 Dateien mit wahlfreiem Zugriff	1094
15.2.1 Ein RandomAccessFile zum Lesen und Schreiben öffnen	1095
15.2.2 Aus dem RandomAccessFile lesen	1096
15.2.3 Schreiben mit RandomAccessFile	1097
15.2.4 Die Länge des RandomAccessFile	1098
15.2.5 Hin und her in der Datei	1098
15.3 Dateisysteme unter NIO.2	1100
15.3.1 FileSystem und Path	1100
15.3.2 Die Utility-Klasse Files	1102
15.4 Stream-Klassen und Reader/Writer am Beispiel von Dateien	1104
15.4.1 Mit dem FileWriter Texte in Dateien schreiben	1105
15.4.2 Zeichen mit der Klasse FileReader lesen	1107
15.4.3 Kopieren mit FileOutputStream und FileInputStream	1108
15.4.4 Datenströme über Files mit NIO.2 beziehen	1110
15.5 Basisklassen für die Ein-/Ausgabe	1113
15.5.1 Die abstrakten Basisklassen	1113

15.5.2	Übersicht über Ein-/Ausgabeklassen	1113
15.5.3	Die abstrakte Basisklasse OutputStream	1116
15.5.4	Die Schnittstellen Closeable, AutoCloseable und Flushable	1117
15.5.5	Die abstrakte Basisklasse InputStream	1118
15.5.6	Ressourcen aus dem Klassenpfad und aus Jar-Archiven laden	1120
15.5.7	Die abstrakte Basisklasse Writer	1120
15.5.8	Die abstrakte Basisklasse Reader	1122
15.6	Datenströme filtern und verketten	1125
15.6.1	Streams als Filter verketten (verschachteln)	1125
15.6.2	Gepufferte Ausgaben mit BufferedWriter und BufferedOutputStream ...	1126
15.6.3	Gepufferte Eingaben mit BufferedReader/BufferedInputStream	1128
15.7	Vermittler zwischen Byte-Streams und Unicode-Strömen	1131
15.7.1	Datenkonvertierung durch den OutputStreamWriter	1131
15.7.2	Automatische Konvertierungen mit dem InputStreamReader	1132

16 Einführung in die <XML>-Verarbeitung mit Java

16.1	Auszeichnungssprachen	1135
16.1.1	Die Standard Generalized Markup Language (SGML)	1136
16.1.2	Extensible Markup Language (XML)	1136
16.2	Eigenschaften von XML-Dokumenten	1137
16.2.1	Elemente und Attribute	1137
16.2.2	Beschreibungssprache für den Aufbau von XML-Dokumenten	1140
16.2.3	Schema – eine Alternative zu DTD	1144
16.2.4	Namensraum (Namespace)	1147
16.2.5	XML-Applikationen *	1148
16.3	Die Java-APIs für XML	1149
16.3.1	Das Document Object Model (DOM)	1150
16.3.2	Simple API for XML Parsing (SAX)	1150
16.3.3	Pull-API StAX	1150
16.3.4	Java Document Object Model (JDOM)	1150
16.3.5	JAXP als Java-Schnittstelle zu XML	1151
16.3.6	DOM-Bäume einlesen mit JAXP *	1152
16.4	Java Architecture for XML Binding (JAXB)	1153
16.4.1	Bean für JAXB aufbauen	1153

16.4.2	JAXBContext und die Marshaller	1154
16.4.3	Ganze Objektgraphen schreiben und lesen	1156
16.5	XML-Dateien mit JDOM verarbeiten	1158
16.5.1	JDOM beziehen	1159
16.5.2	Paketübersicht *	1159
16.5.3	Die Document-Klasse	1161
16.5.4	Eingaben aus der Datei lesen	1162
16.5.5	Das Dokument im XML-Format ausgeben	1163
16.5.6	Elemente	1164
16.5.7	Zugriff auf Elementinhalte	1167
16.5.8	Attributinhalte lesen und ändern	1170
16.6	Zum Weiterlesen	1173

17 Einführung ins Datenbankmanagement mit JDBC

17.1	Relationale Datenbanken	1175
17.1.1	Das relationale Modell	1175
17.2	Einführung in SQL	1176
17.2.1	Ein Rundgang durch SQL-Abfragen	1177
17.2.2	Datenabfrage mit der Data Query Language (DQL)	1179
17.2.3	Tabellen mit der Data Definition Language (DDL) anlegen	1181
17.3	Datenbanken und Tools	1182
17.3.1	HSQLDB	1182
17.3.2	Eclipse-Plugins zum Durchschauen von Datenbanken	1184
17.4	JDBC und Datenbanktreiber	1188
17.5	Eine Beispielabfrage	1190
17.5.1	Schritte zur Datenbankabfrage	1190
17.5.2	Ein Client für die HSQLDB-Datenbank	1190
17.5.3	Datenbankbrowser und eine Beispielabfrage unter NetBeans	1192

18 Bits und Bytes und Mathematisches

18.1	Bits und Bytes *	1199
18.1.1	Die Bit-Operatoren Komplement, Und, Oder und Xor	1200

18.1.2	Repräsentation ganzer Zahlen in Java – das Zweierkomplement	1202
18.1.3	Das binäre (Basis 2), oktale (Basis 8), hexadezimale (Basis 16) Stellenwertsystem	1203
18.1.4	Auswirkung der Typanpassung auf die Bitmuster	1204
18.1.5	byte als vorzeichenlosen Datentyp nutzen	1207
18.1.6	Die Verschiebeoperatoren	1208
18.1.7	Ein Bit setzen, löschen, umdrehen und testen	1210
18.1.8	Bit-Methoden der Integer- und Long-Klasse	1211
18.2	Fließkommaarithmetik in Java	1213
18.2.1	Spezialwerte für Unendlich, Null, NaN	1213
18.2.2	Standard-Notation und wissenschaftliche Notation bei Fließkommazahlen *	1216
18.2.3	Mantisse und Exponent *	1217
18.3	Die Eigenschaften der Klasse Math	1218
18.3.1	Attribute	1220
18.3.2	Absolutwerte und Vorzeichen	1220
18.3.3	Maximum/Minimum	1221
18.3.4	Runden von Werten	1222
18.3.5	Wurzel- und Exponentialmethoden	1225
18.3.6	Der Logarithmus *	1226
18.3.7	Rest der ganzzahligen Division *	1227
18.3.8	Winkelmethoden *	1228
18.3.9	Zufallszahlen	1229
18.4	Genaugigkeit, Wertebereich eines Typs und Überlaufkontrolle *	1230
18.4.1	Behandlung des Überlaufs	1230
18.4.2	Was bitte macht ein ulp?	1233
18.5	Mathe bitte strikt *	1234
18.5.1	Strikte Fließkommaberechnungen mit strictfp	1235
18.5.2	Die Klassen Math und StrictMath	1235
18.6	Die Random-Klasse	1236
18.6.1	Objekte mit dem Samen aufbauen	1236
18.6.2	Zufallszahlen erzeugen	1237
18.6.3	Pseudo-Zufallszahlen in der Normalverteilung *	1238
18.7	Große Zahlen *	1238
18.7.1	Die Klasse BigInteger	1239
18.7.2	Methoden von BigInteger	1241
18.7.3	Ganz lange Fakultäten	1244

18.7.4	Große Fließkommazahlen mit BigDecimal	1246
18.7.5	Mit MathContext komfortabel die Rechengenauigkeit setzen	1248
18.8	Zum Weiterlesen	1250

19 Die Werkzeuge des JDK

19.1	Java-Quellen übersetzen	1252
19.1.1	Java-Compiler vom JDK	1252
19.1.2	Native Compiler	1253
19.1.3	Java-Programme in ein natives ausführbares Programm einpacken	1253
19.2	Die Java-Laufzeitumgebung	1254
19.3	Dokumentationskommentare mit JavaDoc	1259
19.3.1	Einen Dokumentationskommentar setzen	1259
19.3.2	Mit dem Werkzeug javadoc eine Dokumentation erstellen	1262
19.3.3	HTML-Tags in Dokumentationskommentaren *	1263
19.3.4	Generierte Dateien	1263
19.3.5	Dokumentationskommentare im Überblick *	1264
19.3.6	JavaDoc und Doclets *	1266
19.3.7	Veraltete (deprecated) Typen und Eigenschaften	1266
19.4	Das Archivformat Jar	1269
19.4.1	Das Dienstprogramm jar benutzen	1270
19.4.2	Das Manifest	1273
19.4.3	Applikationen in Jar-Archiven starten	1273

A Die Klassenbibliothek

A.1	java.lang-Paket	1285
A.1.1	Schnittstellen	1285
A.1.2	Klassen	1286
Index		1289



Vorwort

»Mancher glaubt, schon darum höflich zu sein, weil er sich überhaupt noch der Worte und nicht der Fäuste bedient.«
– Friedrich Hebbel (1813–1863)

Am 23. Mai 1995 stellten auf der SunWorld in San Francisco der Chef vom damaligen Science Office von Sun Microsystems, John Gage, und Netscape-Mitbegründer Marc Andreessen die neue Programmiersprache Java und deren Integration in den Webbrowser Netscape vor. Damit begann der Siegeszug einer Sprache, die uns elegante Wege eröffnet, um plattformunabhängig zu programmieren und objektorientiert unsere Gedanken abzubilden. Die Möglichkeiten der Sprache und Bibliothek sind an sich nichts Neues, aber so gut verpackt, dass Java angenehm und flüssig zu programmieren ist. Dieses Buch beschäftigt sich in 19 Kapiteln mit Java, den Klassen, der Design-Philosophie und der objektorientierten Programmierung.

Über dieses Buch

Die Zielgruppe

Die Kapitel dieses Buchs sind für Einsteiger in die Programmiersprache Java wie auch für Fortgeschrittene konzipiert. Kenntnisse in einer strukturierten Programmiersprache wie C, Delphi oder Visual Basic und Wissen über objektorientierte Technologien sind hilfreich, weil das Buch nicht explizit auf eine Rechnerarchitektur eingeht oder auf die Frage, was Programmieren eigentlich ist. Wer also schon in einer beliebigen Sprache programmiert hat, der liegt mit diesem Buch genau richtig!

Was dieses Buch nicht ist

Dieses Buch darf nicht als Programmierbuch für Anfänger verstanden werden. Wer noch nie programmiert hat und mit dem Wort »Übersetzen« in erster Linie »Dolmetschen« verbindet, der sollte besser ein anderes Tutorial bevorzugen oder parallel lesen. Viele Bereiche aus dem Leben eines Industrieprogrammierers behandelt »die Insel« bis

zu einer allgemein verständlichen Tiefe, doch sie ersetzt nicht die *Java Language Specification* (JLS: <http://java.sun.com/docs/books/jls/>).

Die Java-Technologien sind in den letzten Jahren explodiert, sodass die anfängliche Überschaubarkeit einer starken Spezialisierung gewichen ist. Heute ist es kaum mehr möglich, alles in einem Buch zu behandeln, und das möchte ich mit der Insel auch auf keinen Fall. Ein Buch, das sich speziell mit der grafischen Oberfläche *Swing* beschäftigt, ist genauso umfangreich wie die jetzige Insel. Nicht anders verhält es sich mit den anderen Spezialthemen, wie etwa objektorientierter Analyse/Design, UML, verteilter Programmierung, Enterprise JavaBeans, Datenbankanbindung, OR-Mapping, Web-Services, dynamischen Webseiten und vielen anderen Themen. Hier muss ein Spezialbuch die Neugier befriedigen.

Die Insel trainiert die Syntax der Programmiersprache, den Umgang mit den wichtigen Standardbibliotheken, Entwicklungstools und Entwicklungsumgebungen, objektorientierte Analyse und Design, Entwurfsmuster und Programmkonventionen. Sie hilft aber weniger, am Abend bei der Party die hübschen Mädels und coolen IT-Geeks zu beeindrucken und mit nach Hause zu nehmen. Sorry.

Mein Leben und Java, oder warum es noch ein Java-Buch gibt

Meine ursprüngliche Beschäftigung mit Java hängt eng mit einer universitären Pflichtveranstaltung zusammen. In unserer Projektgruppe befassten wir uns 1997 mit einer objektorientierten Dialogspezifikation. Ein Zustandsautomat musste programmiert werden, und die Frage nach der Programmiersprache stand an. Da ich den Seminarteilnehmern Java vorstellen wollte, arbeitete ich einen Foliensatz für den Vortrag aus. Parallel zu den Folien erwartete der Professor eine Ausarbeitung in Form einer Seminararbeit. Die Beschäftigung mit Java machte mir Spaß und war etwas ganz anderes, als ich bis dahin gewohnt war. Vor Java kodierte ich rund 10 Jahre in Assembler, später dann mit den Hochsprachen Pascal und C, vorwiegend Compiler. Ich probierte aus, schrieb meine Erfahrungen auf und lernte dabei Java und die Bibliotheken kennen. Die Arbeit wuchs mit meinen Erfahrungen. Während der Projektgruppe sprach mich ein Kommilitone an, ob ich nicht Lust hätte, als Referent eine Java-Weiterbildung zu geben. Lust hatte ich – aber keine Unterlagen. So schrieb ich weiter, um für den Kurs Schulungsunterlagen zu haben. Als der Professor am Ende der Projektgruppe nach der Seminararbeit fragte, war die Vorform der Insel schon so umfangreich, dass die vorliegende Einleitung mehr oder weniger zur Seminararbeit wurde.

Das war 1997, und natürlich hätte ich mit dem Schreiben sofort aufhören können, nachdem ich die Seminararbeit abgegeben habe. Doch bis heute schule ich in Java, und das Schreiben ist eine Lernstrategie für mich. Wenn ich mich in neue Gebiete einarbeite, lese ich erst einmal auf Masse und beginne dann, Zusammenfassungen zu schreiben. Erst beim Schreiben wird mir richtig bewusst, was ich noch nicht weiß. Dieses Lernprinzip hat auch zu meinem ersten Buch über Amiga-Maschinensprachprogrammierung geführt. Doch das MC680x0-Buch kam nicht auf den Markt, denn die Verlage konnten mir nur mitteilen, dass die Zeit der Homecomputer vorbei sei.¹ Mit Java war das anders, denn hier war ich zur richtigen Zeit am richtigen Ort. Die Prognosen für Java stehen ungebrochen gut, weil der Einsatz von Java mittlerweile so gefestigt ist wie der von COBOL bei Banken und Versicherungen.

Heute sehe ich die Insel als ein sehr facettenreiches Java-Buch für die ambitionierten Entwickler an, die hinter die Kulissen schauen wollen. Der Detailgrad der Insel wird von keinem anderen (mir bekannten) deutsch- oder englischsprachigen Grundlagenbuch erreicht.² Die Erweiterung der Insel macht mir Spaß, auch wenn viele Themen kaum in einem normalen Java-Kurs angesprochen werden.



-
- 1 Damit habe ich eine Wette gegen Georg und Thomas verloren – sie durften bei einer großen Imbisskette so viel essen, wie sie wollten. Ich hatte später meinen Spaß, als wir mit dem Auto nach Hause fuhren und dreimal anhalten mussten.
 - 2 Und vermutlich gibt es weltweit kein anderes IT-Fachbuch, das so viele unanständige Wörter im Text versteckt.

Software und Versionen

Als Grundlage für dieses Buch dient die *Java Platform Standard Edition* (Java SE) in der Version 7 in der Implementierung von Oracle, die *Java Development Kit* (Oracle JDK, kurz JDK) genannt wird. Das JDK besteht im Wesentlichen aus einem Compiler und einer Laufzeitumgebung (JVM) und ist für die Plattformen Windows, Linux und Solaris erhältlich. Ist das System kein Windows, Linux oder Solaris, gibt es Laufzeitumgebungen von anderen Unternehmen bzw. vom Hersteller der Plattform: Für Apples Mac OS X gibt es die Java-Laufzeitumgebung von Apple selbst (die bald Teil des OpenJDK sein wird), und IBM bietet für IBM System i (ehemals iSeries) ebenfalls eine Laufzeitumgebung. Die Einrichtung dieser Exoten wird in diesem Buch nicht besprochen.

Eine grafische Entwicklungsoberfläche (IDE) ist kein Teil des JDK. Zwar verlasse ich mich ungern auf einen Hersteller, weil die Hersteller unterschiedliche Entwicklergruppen ansprechen, doch sollen in diesem Buch die freien Entwicklungsumgebungen *Eclipse* und *NetBeans* Verwendung finden. Die Beispielprogramme lassen sich grundsätzlich mit beliebigen anderen Entwicklungsumgebungen, wie etwa IntelliJ IDEA oder Oracle JDeveloper, verarbeiten oder mit einem einfachen ASCII-Texteditor, wie Notepad (Windows) oder vi (Unix), eingeben und auf der Kommandozeile übersetzen. Diese Form der Entwicklung ist allerdings nicht mehr zeitgemäß, sodass ein grafischer Kommandozeilen-Aufsatz die Programmerstellung vereinfacht.

Welche Java-Version verwenden wir?

Seit Oracle (damals noch von Sun geführt) die Programmiersprache Java 1995 mit Version 1.0 vorgestellt hat, drehte sich die Versionsspirale bis Version 7 (was gleichbedeutend mit Versionsnummer 1.7 ist). Besonders für Java-Buch-Autoren stellt sich die Frage, auf welcher Java-Version ihr Text aufbauen muss und welche Bibliotheken es beschreiben soll. Ich habe das Problem so gelöst, dass ich immer die Möglichkeiten der neuesten Version beschreibe, was zur Drucklegung die Java SE 7 war. Für die Didaktik der objektorientierten Programmierung ist die Versionsfrage glücklicherweise unerheblich.

Da viele Unternehmen noch unter Java 5 entwickeln, wirft die breite Nutzung von Features der Java-Version 7 unter Umständen Probleme auf, denn nicht jedes Beispielprogramm aus der Insel lässt sich per Copy & Paste fehlerfrei in das eigene Projekt übertragen. Da Java 7 fundamentale neue Möglichkeiten in der Programmiersprache bietet, kann ein Java 5- oder Java 6-Compiler natürlich nicht alles übersetzen. Um das Problem zu entschärfen und um nicht viele Beispiele im Buch ungültig zu machen, werden die Bibliotheksänderungen und Sprachneuerungen von Java 7 zwar ausführlich in den einzelnen Kapiteln beschrieben, aber die Beispielprogramme in anderen Kapiteln bleiben

auf dem Sprachniveau von Java 5 bzw. Java 6.³ So laufen mehrheitlich alle Programme unter den verbreiteten Versionen Java 5 und Java 6. Auch wenn die Sprachänderungen von Java 7 zu einer Verkürzung führen, ist es unrealistisch anzunehmen, dass jedes Unternehmen kurz nach der Herausgabe der neuen Java-Version und der 10. Auflage der Insel auf Java 7 wechselt. Es ist daher nur naheliegend, das Buch für eine breite Entwicklergemeinde auszulegen, anstatt für ein paar Wenige, die sofort mit der neusten Version arbeiten können. Erst in der nächsten Auflage, die synchron mit Java 8 folgt, werden die Beispiele überarbeitet und wenn möglich an die neuen Sprachmittel von Java 7 angepasst.

Das Buch in der Lehre einsetzen

»Die Insel« eignet sich ideal zum Selbststudium. Das erste Kapitel dient zum Warmwerden und plaudert ein wenig über dieses und jenes. Wer auf dem Rechner noch keine Entwicklungsumgebung installiert hat, der sollte zuerst das JDK von Oracle installieren.

Weil das JDK nur Kommandozeilentools installiert, sollte jeder Entwickler eine grafische IDE (*Integrated Development Environment*) installieren, da eine IDE die Entwicklung von Java-Programmen deutlich komfortabler macht. Eine IDE bietet gegenüber der rohen Kommandozeile einige Vorteile:

- Das Editieren, Kompilieren und Laufenlassen eines Java-Programms ist schnell und einfach über einen Tastendruck oder Mausklick möglich.
- Ein Editor sollte die Syntax von Java farbig hervorheben (Syntax-Highlighting).
- Eine kontextsensitive Hilfe zeigt bei Methoden die Parameter an, und gleichzeitig verweist sie auf die API-Dokumentation.

Weitere Vorteile wie GUI-Builder, Projektmanagement und Debuggen sollen jetzt keine Rolle spielen. Wer neu in die Programmiersprache Java einsteigt, wird an Eclipse seine Freude haben. Es wird im ersten Kapitel ebenfalls beschrieben.

Zum Entwickeln von Software ist die Hilfe unerlässlich. Sie ist von der Entwicklungsumgebung in der Regel über einen Tastendruck einsehbar oder online zu finden. Unter welcher URL sie verfügbar ist, erklärt ebenfalls Kapitel 1.

Richtig los geht es mit Kapitel 2, und von da an geht es didaktisch Schritt für Schritt weiter. Wer Kenntnisse in C hat, kann Kapitel 2 überblättern. Wer schon in C++/C# objekt-

³ In Java 6 wurden keine neuen Spracheigenschaften hinzugefügt, nur eine Kleinigkeit bei der Gültigkeit der @Override-Annotation änderte sich.

orientiert programmiert hat, kann Kapitel 3 überfliegen und dann einsteigen. Objekt-orientierter Mittelpunkt des Buchs ist Kapitel 5: Es vermittelt die OO-Begriffe Klasse, Methode, Assoziation, Vererbung, dynamisches Binden... Nach Kapitel 5 ist die objekt-orientierte Grundausbildung abgeschlossen, und nach Kapitel 9 sind die Grundlagen von Java bekannt. Es folgen Vertiefungen in einzelne Bereiche der Java-Bibliothek.

Mit diesem Buch und einer Entwicklungsumgebung Ihres Vertrauens können Sie die ersten Programme entwickeln. Um eine neue Programmiersprache zu erlernen, reicht das Lesen aber nicht aus. Mit den Übungsaufgaben auf der DVD können Sie deshalb auch Ihre Fingerfertigkeit trainieren. Da Lösungen beigelegt sind, lassen sich die eigenen Lösungen gut mit den Musterlösungen vergleichen. Vielleicht bietet die Buchlösung noch eine interessante Lösungsidee oder Alternative an.

Persönliche Lernstrategien

Wer das Buch im Selbststudium nutzt, wird wissen wollen, was eine erfolgreiche Lernstrategie ist. Der Schlüssel zur Erkenntnis ist, wie so oft, die Lernpsychologie, die untersucht, unter welchen Lesebedingungen ein Text optimal verstanden werden kann. Die Methode, die ich vorstellen möchte, heißt PQ4R-Methode, benannt nach den Anfangsbuchstaben der Schritte, die die Methode vorgibt:

- **Vorschau (Preview):** Zunächst sollten Sie sich einen ersten Überblick über das Kapitel verschaffen, etwa durch Blättern im Inhaltsverzeichnis und in den Seiten der einzelnen Kapitel. Schauen Sie sich die Abbildungen und Tabellen etwas länger an, da sie schon den Inhalt verraten und Lust auf den Text vermitteln.
- **Fragen (Question):** Jedes Kapitel versucht, einen thematischen Block zu vermitteln. Vor dem Lesen sollten Sie sich überlegen, welche Fragen das Kapitel beantworten soll.
- **Lesen (Read):** Jetzt geht's los, der Text wird durchgelesen. Wenn es nicht gerade ein geliehenes Bücherei-Buch ist, sollten Sie Passagen, die Ihnen wichtig erscheinen, mit vielen Farben hervorheben und mit Randbemerkungen versehen. Gleches gilt für neue Begriffe. Die zuvor gestellten Fragen sollte jeder beantworten können. Sollten neue Fragen auftauchen – im Gedächtnis abspeichern!
- **Nachdenken (Reflect):** Egal, ob motiviert oder nicht – das ist ein interessantes Ergebnis einer anderen Studie –, lernen kann jeder immer. Der Erfolg hängt nur davon ab, wie tief das Wissen verarbeitet wird (elaborierte Verarbeitung). Dazu müssen die Themen mit anderen Themen verknüpft werden. Überlegen Sie, wie die Aussagen mit den anderen Teilen zusammenpassen. Dies ist auch ein guter Zeitpunkt für praktische Übungen. Für die angegebenen Beispiele im Buch sollten Sie sich eigene Bei-

spiele überlegen. Wenn der Autor eine if-Abfrage am Beispiel des Alters beschreibt, wäre eine eigene Idee etwa eine if-Abfrage zur Hüpfballgröße.

- *Wiedergeben (Recite)*: Die zuvor gestellten Fragen sollten sich nun beantworten lassen, und zwar ohne den Text. Für mich ist das Schreiben eine gute Möglichkeit, um über mein Wissen zu reflektieren, doch sollte dies jeder auf seine Weise tun. Allelal ist es lustig, sich während des Duschens über alle Schlüsselwörter und ihre Bedeutung, den Zusammenhang zwischen abstrakten Klassen und Schnittstellen usw. klar zu werden. Ein Tipp: Lautes Erklären hilft bei vielen Arten der Problemlösung – quatschen Sie einfach mal den Toaster zu. Noch schöner ist es, mit jemandem zusammen zu lernen und sich gegenseitig die Verfahren zu erklären. Eine interessante Visualisierungstechnik ist die Mind-Map. Sie dient dazu, den Inhalt zu gliedern.
- *Rückblick (Review)*: Nun gehen Sie das Kapitel noch einmal durch und schauen, ob Sie alles ohne weitere Fragen verstanden haben. Manche »schnellen« Erklärungen haben sich vielleicht als falsch herausgestellt. Vielleicht klärt der Text auch nicht alles. Dann ist ein an mich gerichteter Hinweis (c.ullenboom@tutego.de) angebracht.

Fokus auf das Wesentliche

Einige Unterkapitel sind für erfahrene Programmierer oder Informatiker geschrieben. Besonders der Neuling wird an einigen Stellen den sequenziellen Pfad verlassen müssen, da spezielle Kapitel mehr Hintergrundinformationen und Vertrautheit mit Programmiersprachen erfordern. Verweise auf C(++), C# oder andere Programmiersprachen dienen aber nicht wesentlich dem Verständnis, sondern nur dem Vergleich.

Einsteiger in Java können noch nicht zwischen dem absolut notwendigen Wissen und einer interessanten Randnotiz unterscheiden. Die Insel gewichtet aus diesem Grund das Wissen auf zwei Arten. Zunächst gibt es vom Text abgesetzte Boxen, die zum Teil spezielle und fortgeschrittene Informationen bereitstellen. Des Weiteren enden einige Überschriften auf ein *, was bedeutet, dass dieser Abschnitt übersprungen werden kann, ohne dass dem Leser etwas Wesentliches für die späteren Kapitel fehlt.

Organisation der Kapitel

Kapitel 1, »Java ist auch eine Sprache«, zeigt die Besonderheiten der Sprache Java auf. Einige Vergleiche mit anderen populären objektorientierten Sprachen werden gezogen. Die Absätze sind nicht besonders technisch und beschreiben auch den historischen Ablauf der Entwicklung von Java. Das Kapitel ist nicht didaktisch aufgebaut, sodass einige Begriffe erst in den weiteren Kapiteln vertieft werden; Einsteiger sollten es querlesen.

Ebenso wird hier dargestellt, wie das Java JDK von Oracle zu beziehen und zu installieren ist, damit die ersten Programme übersetzt und gestartet werden können.

Richtig los geht es in **Kapitel 2**, »Imperative Sprachkonzepte«. Es hebt Variablen, Typen und die imperativen Sprachelemente hervor und schafft mit Anweisungen und Ausdrücken die Grundlagen für jedes Programm. Hier finden auch Fallanweisungen, die diversen Schleifentypen und Methoden ihren Platz. Das alles geht noch ohne große Objektorientierung.

Objektorientiert wird es dann in **Kapitel 3**, »Klassen und Objekte«. Dabei kümmern wir uns erst einmal um die in der Standardbibliothek vorhandenen Klassen und entwickeln eigene Klassen später. Die Bibliothek ist so reichhaltig, dass allein mit den vordefinierten Klassen schon viele Programme entwickelt werden können. Speziell die bereitgestellten Datenstrukturen lassen sich vielfältig einsetzen.

Wichtig ist für viele Probleme auch der in **Kapitel 4** vorgestellte »Umgang mit Zeichenketten«. Die beiden notwendigen Klassen `Character` für einzelne Zeichen und `String`, `StringBuffer/StringBuilder` für Zeichenfolgen werden eingeführt, und auch ein Abschnitt über reguläre Ausdrücke fehlt nicht. Bei den Zeichenketten müssen Teile ausgeschnitten, erkannt und konvertiert werden. Ein `split()` vom `String` und der `Scanner` zerlegen Zeichenfolgen anhand von Trennern in Teilzeichenketten. Format-Objekte bringen beliebige Ausgaben in ein gewünschtes Format. Dazu gehört auch die Ausgabe von Dezimalzahlen.

Mit diesem Vorwissen über Objekterzeugung und Referenzen kann der nächste Schritt erfolgen: In **Kapitel 5** werden wir »Eigene Klassen schreiben«. Anhand von Spielen und Räumen modellieren wir Objekteigenschaften und zeigen Benutz- und Vererbungsbeziehungen auf. Wichtige Konzepte – wie statische Eigenschaften, dynamisches Binden, abstrakte Klassen und Schnittstellen (Interfaces) sowie Sichtbarkeit – finden dort ihren Platz. Da Klassen in Java auch innerhalb anderer Klassen liegen können (innere Klassen), setzt sich ein eigenes Unterkapitel damit auseinander.

Ausnahmen, die wir in **Kapitel 6**, »Exceptions«, behandeln, bilden ein wichtiges Rückgrat in Programmen, da sich Fehler kaum vermeiden lassen. Da ist es besser, die Behandlung aktiv zu unterstützen und den Programmierer zu zwingen, sich um Fehler zu kümmern und diese zu behandeln.

Kapitel 7, »Äußere.innere Klassen«, beschreibt, wie sich Klassen ineinander verschachteln lassen. Das verbessert die Kapselung, denn auch Implementierungen können dann sehr lokal sein.

Kapitel 8, »Besondere Klassen der Java SE«, geht auf die Klassen ein, die für die Java-Bibliothek zentral sind, etwa Vergleichsklassen, Wrapper-Klassen oder die Klasse `Object`, die die Oberklasse aller Java-Klassen ist.

Mit Generics lassen sich Klassen, Schnittstellen und Methoden mit einer Art Typ-Platzhalter deklarieren, wobei der konkrete Typ erst später festgelegt wird. **Kapitel 9**, »Generics<T>«, gibt einen Einblick in die Technik.

Danach sind die Fundamente gelegt, und die verbleibenden Kapitel dienen dazu, das bereits erworbene Wissen auszubauen. **Kapitel 10**, »Architektur, Design und angewandte Objektorientierung«, zeigt Anwendungen guter objektorientierter Programmierung und stellt Entwurfsmuster (Design-Pattern) vor. An unterschiedlichen Beispielen demonstriert das Kapitel, wie Schnittstellen und Klassenhierarchien gewinnbringend in Java eingesetzt werden. Es ist der Schlüssel dafür, nicht nur im Kleinen zu denken, sondern auch große Applikationen zu schreiben.

Nach den ersten zehn Kapiteln haben die Leser die Sprache Java nahezu komplett kennengelernt. Da Java aber nicht nur eine Sprache ist, sondern auch ein Satz von Standardbibliotheken, konzentriert sich die zweite Hälfte des Buchs auf die grundlegenden APIs. Jeweils am Ende eines Kapitels findet sich ein Unterkapitel »Zum Weiterlesen« mit Verweisen auf interessante Internetadressen – in der Java-Sprache `finally{}` genannt. Hier kann der Leser den sequenziellen Pfad verlassen und sich einzelnen Themen widmen, da die Themen in der Regel nicht direkt voneinander abhängen.

Die Java-Bibliothek besteht aus mehr als 4.000 Klassen, Schnittstellen, Aufzählungen, Ausnahmen und Annotationen. Das **Kapitel 11**, »Die Klassenbibliothek«, (und auch der Anhang A) gibt eine Übersicht über die wichtigsten Pakete und greift einige Klassen aus der Bibliothek heraus, etwa zum Laden von Klassen. Hier sind auch Klassen zur Konfiguration von Anwendungen oder Möglichkeiten zum Ausführen externer Programme zu finden.

Die Kapitel 11 bis 16 geben einen Überblick über spezielle APIs auf. Die Bibliotheken sind sehr umfangreich, und das aufbauende Java-Expertenbuch vertieft die API weiter. **Kapitel 12** gibt eine »Einführung in die nebenläufige Programmierung«. **Kapitel 13**, »Einführung in Datenstrukturen und Algorithmen«, zeigt praxisnah geläufige Datenstrukturen wie Listen, Mengen und Assoziativspeicher. Einen Kessel Buntes bietet **Kapitel 14**, »Einführung in grafische Oberflächen«, wo es um die Swing-Bibliothek geht. Darüber, wie aus Dateien gelesen und geschrieben wird, gibt **Kapitel 15**, »Einführung in Dateien und Datenströme«, einen Überblick. Da Konfigurationen und Daten oftmals im XML-Format vorliegen, zeigt **Kapitel 16**, »Einführung in die <XML>-Verarbeitung mit Java«, auf, welche Möglichkeiten zur XML-Verarbeitung die Bibliothek bietet. Wer stattdessen eine

relationale Datenbank bevorzugt, der findet in **Kapitel 17**, »Einführung ins Datenbankmanagement mit JDBC«, Hilfestellungen. Alle genannten Kapitel geben aufgrund der Fülle nur einen Einblick, und ihre Themen werden im zweiten Band noch tiefer aufgefächert.

Kapitel 18 stellt »Bits und Bytes und Mathematisches« vor. Die Klasse `Math` hält typische mathematische Methoden bereit, um etwa trigonometrische Berechnungen durchzuführen. Mit einer weiteren Klasse können Zufallszahlen erzeugt werden. Auch behandelt das Kapitel den Umgang mit beliebig langen Ganz- oder Fließkommazahlen. Die meisten Entwickler benötigen nicht viel Mathematik, daher ist es das Schlusskapitel.

Abschließend liefert **Kapitel 19**, »Die Werkzeuge des JDK«, eine Kurzübersicht der Kommandozeilenwerkzeuge `javac` zum Übersetzen von Java-Programmen und `java` zum Starten der JVM und Ausführen der Java-Programme.

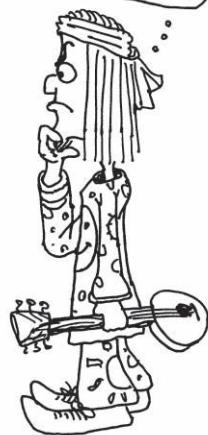
Anhang A, »Die Klassenbibliothek« erklärt alle Java-Pakete kurz mit einem Satz und zudem alle Typen im absolut essenziellen Paket `java.lang`.

Konventionen

In diesem Buch werden folgende Konventionen verwendet:

- Neu eingeführte Begriffe sind *kursiv* gesetzt, und der Index verweist genau auf diese Stelle. Des Weiteren sind *Dateienamen*, *HTTP-Adressen*, *Namen ausführbarer Programme*, *Programmoptionen* und *Dateiendungen (.txt)* kursiv. Einige Links führen nicht direkt zur Ressource, sondern werden über <http://www.tutego.de/go> zur tatsächlichen Quelle umgeleitet, was Änderungen erleichtert.
- Begriffe der Benutzeroberfläche stehen in **KAPITÄLCHEN**.
- Listings, Methoden und sonstige Programmelemente sind in nicht-proportionaler Schrift gesetzt. An einigen Stellen wurde hinter eine Listingzeile ein abgeknickter Pfeil als Sonderzeichen gesetzt, das den Zeilenumbruch markiert. Der Code aus der nächsten Zeile gehört also noch zur vorigen.
- Um im Programmcode Compilerfehler oder Laufzeitfehler anzuzeigen, steht in der Zeile ein `✗`. So ist auf den ersten Blick abzulesen, dass die Zeile nicht compiliert wird oder zur Laufzeit aufgrund eines Programmierfehlers eine Ausnahme auslöst. Beispiel:

Konventionen???



```
int p = new java.awt.Point();      // ☹ Compilerfehler: Type mismatch
```

- Bei Compilerfehlern – wie im vorangehenden Punkt – kommen die Fehlermeldungen in der Regel von Eclipse. Sie sind dort anders benannt als in NetBeans bzw. dem Kommandozeilencompiler *javac*. Aber natürlich führen beide Compiler zu ähnlichen Fehlern.
- Bei Methodennamen im Fließtext folgt immer ein Klammerpaar. Die Parameter werden nur dann aufgeführt, wenn sie wichtig sind.
- Um eine Gruppe von Methoden anzugeben, symbolisiert die Kennung `XXX` einen Platzhalter. So zeigt zum Beispiel `printXXX()` die Methoden `println()`, `print()` und `printf()` an. Aus dem Kontext geht hervor, welche Methoden gemeint sind.
- Raider heißt jetzt Twix, und Sun ging Anfang 2010 an Oracle. Auch wenn es für langjährige Entwickler hart ist: Der Name »Sun« verschwindet, und der geliebte Datenbankhersteller tritt an seine Stelle. Er taucht immer nur dann auf, wenn es um eine Technologie geht, die von Sun initiiert wurde und in der Zeit auf den Markt kam, in der Sun sie verantwortete.

Programmlistings

Komplette Programmlistings sind wie folgt aufgebaut:

Listing 0.1: Person.java

```
class Person
{}
```

Der abgebildete Quellcode befindet sich in der Datei *Person.java*. Befindet sich der Typ (Klasse, Aufzählung, Schnittstelle, Annotation) in einem Paket, steht die Pfadangabe beim Dateinamen:

Listing 0.2: com/tutego/insel/Person.java

```
package com.tutego.insel;
class Person { }
```

Um Platz zu sparen, stellt das Buch oftmals Quellcode-Ausschnitte dar. Der komplette Quellcode ist auf der DVD beziehungsweise im Internet verfügbar. Hinter dem Typ folgen in dem Fall Kennungen des abgedruckten Teils. Ist nur die Typdeklaration einer Datei ohne package- oder import-Deklaration aufgelistet, so steht hinter dem Dateinamen der Typ, etwa so:

Listing 0.3: Person.java, Person

Listing 0.4: Person.java, House

Im folgenden Fall wird nur die `main()`-Methode abgebildet:

Listing 0.5: Person.java, main()

Wird ein Ausschnitt einer Datei *Person.java* abgebildet, steht »Ausschnitt« oder »Teil 1«, »Teil 2«... dabei:

Listing 0.6: Person.java, Ausschnitt

Listing 0.7: Person.java, main() Teil 1

Gibt es Beispielprogramme für bestimmte Klassen, so enden die Klassennamen dieser Programme im Allgemeinen auf *-Demo*. Für die Java-Klasse `DateFormat` heißt somit ein Beispielprogramm, das die Funktionalität der Klasse `DateFormat` vorführt, *DateFormat-Demo*.

API-Dokumentation im Buch

Attribute, Konstruktoren und Methoden finden sich in einer speziellen Auflistung, die es ermöglicht, sie leicht im Buch zu finden und die Insel als Referenzwerk zu nutzen.

```
abstract class java.text.DateFormat  
extends Format  
implements Cloneable, Serializable
```

- `Date parse(String source) throws ParseException`
Parst einen Datum- oder einen Zeit-String.

Im Rechteck steht der vollqualifizierte Klassen- oder Schnittstellenname (etwa die Klasse `DateFormat` im Paket `java.text`) beziehungsweise der Name der Annotation. In den nachfolgenden Zeilen sind die Oberklasse (`DateFormat` erbt von `Format`) und die implementierten Schnittstellen (`DateFormat` implementiert `Cloneable` und `Serializable`) aufgeführt. Da jede Klasse, die keine explizite Oberklasse hat, automatisch von `Object` erbt, ist diese nicht extra angegeben. Die Sichtbarkeit ist, wenn nicht anders angegeben, `public`, da dies für Bibliotheksmethoden üblich ist. Wird eine Schnittstelle beschrieben, sind die Methoden automatisch abstrakt und öffentlich, und die Schlüsselwörter `abstract` und `public` werden nicht zusätzlich angegeben. In der anschließenden Aufzählung folgen Konstruktoren, Methoden und Attribute. Wenn nicht anders angegeben, ist die Sichtbarkeit `public`. Sind mit `throws` Fehler angegeben, dann handelt es sich nicht um

RuntimeExceptions, sondern nur um geprüfte Ausnahmen. Veraltete (deprecated) Methoden sind nicht aufgeführt, lediglich, wenn es überhaupt keine Alternative gibt.

Ausführbare Programme

Ausführbare Programme auf der Kommandozeile sind durch ein allgemeines Dollarzeichen am Anfang zu erkennen (auch wenn andere Betriebssysteme und Kommandozeilen ein anderes Prompt anzeigen). Die vom Anwender einzugebenden Zeichen sind fett gesetzt, die Ausgabe nicht:

```
$ java FirstLuck
```

Hart arbeiten hat noch nie jemanden getötet. Aber warum das Risiko auf sich nehmen?

Über die richtige Programmierer-»Sprache«

Die Programmierer-Sprache in diesem Buch ist Englisch, um ein Vorbild für »echte« Programme zu sein. Bezeichner wie Klassennamen, Methodenamen und auch eigene API-Dokumentationen sind auf Englisch, um eine Homogenität mit der englischen Java-Bibliothek zu schaffen. Zeichenketten und Konsolenausgaben sowie die Zeichenketten in Ausnahmen (Exceptions) sind in der Regel auf Deutsch, da es in realistischen Programmen kaum hart einkodierte Meldungen gibt – spezielle Dateien halten unterschiedliche Landessprachen vor. Zeilenkommentare sind als interne Dokumentation ebenfalls auf Deutsch vorhanden.

Online-Informationen und -Aufgaben

Dieses Buch ist in der aktuellen Version im Internet unter der Adresse <http://www.tutego.de/javabuch/> und <http://www.galileocomputing.de/> erhältlich. Die Webseiten informieren umfassend über das Buch und über die kommenden Versionen, etwa Erscheinungsdatum oder Bestellnummer. Der Quellcode der Beispielprogramme ist entweder komplett oder mit den bedeutenden Ausschnitten im Buch abgebildet. Ein Zip-Archiv mit allen Beispielen ist auf der Buch-Webseite erhältlich sowie auf die Buch-DVD gepresst. Alle Programmteile sind frei von Rechten und können ungefragt in eigene Programme übernommen und modifiziert werden.

Wer eine Programmiersprache erlernen möchte, muss sie wie eine Fremdsprache sprechen. Begleitend gibt es eine Aufgabensammlung unter <http://www.tutego.de/aufgaben/j/>, die ebenfalls auf der DVD ist. Viele Musterlösungen sind dabei. Die Seite wird in regelmäßigen Abständen mit neuen Aufgaben und Lösungen aktualisiert.

Passend zur Online-Version verschließt sich das Buch nicht den Kollaborationsmöglichkeiten des Web 2.0. Neue Kapitel und Abschnitte des Buches werden immer im Java-Insel-Blog <http://javainselblog.tutego.de/> veröffentlicht.



Abbildung 1: Der Blog zum Buch und mit tagesaktuellen Java-News

Leser erfahren im Blog von allen Aktualisierungen im Buch und können das Geschehen kommentieren. Neben den reinen Updates aus dem Buch publiziert der Blog auch tagesaktuelle Nachrichten über die Java-Welt und Java-Tools. Facebook-Nutzer können ein Fan der Insel werden (<http://www.facebook.com/pages/Javainsel/157203814292515>), und Twitter-Nutzer können den Nachrichtenstrom unter <http://twitter.com/javabuch> abonnieren.

Weiterbildung durch tutego

Unternehmen, die zur effektiven Weiterbildung ihrer Mitarbeiter IT-Schulungen wünschen, können einen Blick auf <http://www.tutego.de/seminare/> werfen. tutego bietet über hundert IT-Seminare zu Java-Themen, C++, C#/.NET, Datenbanken (Oracle, MySQL), XML (XSLT, Schema), Netzwerken, Internet, Office etc. Zu den Java-Themen zählen unter anderem:

- Java-Einführung, Java für Fortgeschrittene, Java für Umsteiger
- Softwareentwicklung mit Eclipse
- nebenläufiges Programmieren mit Threads
- JavaServer Faces (JSF), JavaServer Pages (JSP), Servlets und weitere Web-Technologien
- Datenbankanbindung mit JDBC, OR-Mapping mit JPA und Hibernate
- Java EE, EJB
- grafische Oberflächen mit Swing und JFC; Eclipse RPC, SWT
- Java und XML, JAXB
- Android

Danksagungen

Der größte Dank gebührt Sun Microsystems, die 1991 mit der Entwicklung begannen. Ohne Sun gäbe es kein Java, und ohne Java gäbe es auch nicht dieses Java-Buch. Dank gehört auch der Oracle Company als Käufer von Sun, denn vielleicht wäre ohne die Übernahme Java bald am Ende gewesen.

Die professionellen, aufheiternden Comics stammen von Andreas Schultze (*Akws@aol.com*). Ich danke auch den vielen Buch- und Artikelautoren für ihre interessanten Werke, aus denen ich mein Wissen über Java schöpfen konnte. Ich danke meinen Eltern für ihre Liebe und Geduld und meinen Freunden und Freundinnen für ihr Vertrauen. Ein weiteres Dankeschön geht an verschiedene treue Leser, deren Namen aufzulisten viel Platz kosten würde; ihnen ist die Webseite <http://www.tutego.de/javabuch/korrekteure.htm> gewidmet.

Java lebt – vielleicht sollte ich sogar »überlebt« sagen ... – durch viele freie gute Tools und eine aktive Open-Source-Community. Ein Dank geht an alle Entwickler, die großartige Java-Tools wie Eclipse, NetBeans, Ant, Maven, GlassFish, Tomcat, JBoss und Hunderte andere Bibliotheken schreiben und warten: Ohne Sie wäre Java heute nicht da, wo es ist.

Abschließend möchte ich dem Verlag Galileo Press meinen Dank für die Realisierung und die unproblematische Zusammenarbeit aussprechen. Für die Zusammenarbeit mit meiner Lektorin Judith bin ich sehr dankbar.

Feedback

Auch wenn wir die Kapitel noch so sorgfältig durchgegangen sind, ist es nicht auszuschließen, dass es noch Unstimmigkeiten⁴ gibt; vielmehr ist es bei 1.000 Seiten wahrscheinlich. Wer Anmerkungen, Hinweise, Korrekturen oder Fragen zu bestimmten Punkten oder zur allgemeinen Didaktik hat, der sollte sich nicht scheuen, mir eine E-Mail unter der Adresse c.ullenboom@tutego.de zu senden. Ich bin für Anregung, Lob und Tadel stets empfänglich.

In der Online-Version des Buchs haben wir eine besondere Möglichkeit zur Rückmeldung: Unter jedem Kapitel gibt es eine Textbox, sodass Leser uns schnell einen Hinweis schicken können. In der Online-Version können wir zudem Fehler schnell korrigieren, denn es gibt zum Teil bedauerliche Konvertierungsprobleme vom Buch ins HTML-Format, und einige Male blieb das Hochzeichen (^) auf der Strecke, sodass statt » 2^{16} « im Text ein »216« die Leser verwunderte.

Und jetzt wünsche ich Ihnen viel Spaß beim Lesen und Lernen von Java!

Sonsbeck im Jahr 2011, Jahr 1 nach Oracles Übernahme

Christian Ullenboom

Vorwort zur 10. Auflage

Die größte Neuerung in der 10. Auflage ist die Aufspaltung der Insel in ein Einführungsbuch und ein Fortgeschrittenenbuch. Eine Aufteilung wurde aus zwei Gründen nötig: a) Die Insel kam mit mehr als 1.400 Seiten an ihr druckbares Ende. Da die Java-Bibliotheken aber immer größer wurden und die Syntax (in langsamem Tempo) ebenso zunahm, mussten mehr und mehr Absätze aus der Insel ausgelagert werden. Dadurch verlor die Insel an Tiefe, eine Eigenschaft, die Leser aber an diesem Buch liebten. Der zweite Grund für ein Splitting ist, dass Spracheinsteiger, die beginnen, Variablen zu deklarieren und Klassen zu modellieren, nicht in einem Rutsch gleich mit Generics beginnen, RMI-Aufrufe starten oder mit JNI auf C-Funktionen zugreifen – für Einsteiger wäre das eine Art Bulimie-Wissen: da rein, da raus. Daher adressieren die beiden Bücher zwei unterschiedliche Zielgruppen: Dieses Buch spricht Einsteiger in Java an, die die Sprache und ihre Standardbibliothek praxisnah und in vielen Facetten lernen möchten. Fortgeschrittene Java-Entwickler mit längerer Praxiserfahrung bekommen im zweiten Buch einen tiefe-

⁴ Bei mir wird gerne ein »wir« zum »wie« – wie(r) dumm, dass die Tasten so eng beieinanderliegen.

ren Einblick in Generics und in die Java SE-Bibliotheken sowie einen Ausblick auf Web- und Swing-Programmierung.

Gegenüber der 9. Auflage ergeben sich folgende Änderungen: Das zweite Kapitel mit den imperativen Konzepten ist um ein Zahlenratespiel erweitert worden, sodass die Beispiele nicht so trocken, sondern anschaulicher sind. Zudem war in den Türmen von Hanoi die Silber- und Gold-Säule vertauscht, was mehrere Jahre keinem aufgefallen ist und mich dazu verleitet anzunehmen, dass die Rekursion mit den Türmen nur auf ein geringes Interesse stößt. Die Einführung in Unicode, die ebenfalls in Kapitel 2 stand, ist ins Kapitel 4 gewandert, was sich nun komplett um Zeichen und Zeichenketten kümmert; vorher war das Thema unnötig gespalten und das Kapitel am Anfang zu detaillastig.

Durch die Version 7 gibt es nur wenige Änderungen, und hier sind die Zuwächse eher minimal. Die Neuerungen in der Sprache lassen sich an einer Hand abzählen: Unterstriche in Literalen, `switch` mit String, Binär/Short-Präfixe, Diamanten-Typ, Multi-Catch bei Ausnahmen, präzisiertes Auslösen von Ausnahmen, ARM-Blöcke ... Im Buch macht das vielleicht 1 % der Änderungen aus.

Vorwort zur 9. Auflage

Neben Detailverbesserungen habe ich das Generics-Kapitel komplett neu geschrieben, und viele Abschnitte und Kapitel umsortiert, um sie didaktisch leichter zugänglich zu machen. Auch sprachlich ist die Insel wieder etwas präziser geworden: Der Begriff »Funktion« für eine statische Methode ist abgesetzt, und es heißt jetzt »statische Methode« oder eben »Objektmethode«, wenn der Unterschied wichtig ist, und einfach nur »Methode«, wenn der Unterschied nicht relevant ist. Dass Java von Sun zu Oracle übergegangen ist und vollständig Open Source ist, bleibt auch nicht unerwähnt, genauso wie neue Technologien, zu denen etwa JavaFX gehört. Durch diesen erhöhten Detailgrad mussten leider einige Kapitel (wie JNI, Java ME) aus der Insel fallen. Weiterhin gibt es Bezüge zu der kommenden Version Java 7 und viele interessante Sprachvergleiche, wie Features in anderen Programmiersprachen aussehen und inwiefern sie sich von Java unterscheiden.

Nach dem Vorwort ist es jetzt jedoch an Zeit, zur Sache zu kommen und dem griechischen Philosophen Platon zu folgen, der sagte: »Der Beginn ist der wichtigste Teil der Arbeit.«



Kapitel 1

Java ist auch eine Sprache

»Wir produzieren heute Informationen en masse, so wie früher Autos.«
– John Naisbitt (*1929)

Nach fast 20 Jahren hat sich Java als Plattform etabliert. Über 9 Millionen Softwareentwickler verdienen weltweit mit der Sprache ihre Brötchen, 3 Milliarden Mobiltelefone führen Java-Programme aus,¹ 1,1 Milliarden Desktops und alle Blu-ray-Player. Es gibt 10 Millionen Downloads von Oracles Laufzeitumgebung in jeder Woche, was fast 1 Milliarde Downloads pro Jahr ergibt.

Dabei war der Erfolg nicht unbedingt vorhersehbar. Java² hätte einfach nur eine schöne Insel, eine reizvolle Wandfarbe oder eine Pinte mit brasilianischen Rhythmen in Paris sein können, so wie Heuschrecken einfach nur grüne Hüpfen hätten bleiben können. Doch als robuste objektorientierte Programmiersprache mit einem großen Satz von Bibliotheken ist Java als Sprache für Softwareentwicklung im Großen angekommen und im Bereich plattformunabhängiger Programmiersprachen konkurrenzlos.

1.1 Historischer Hintergrund

In den 1970er-Jahren wollte Bill Joy eine Programmiersprache schaffen, die alle Vorteile von *MESA* und *C* vereinigen sollte. Diesen Wunsch konnte sich Joy zunächst nicht erfüllen, und erst Anfang der 1990er-Jahre beschrieb er in dem Artikel »Further«, wie eine neue objektorientierte Sprache aussehen könnte; sie sollte in den Grundzügen auf *C++* aufbauen. Erst später wurde ihm bewusst, dass *C++* als Basissprache ungeeignet und für große Programme unhandlich ist.

1 So verkündet es Thomas Kurian auf der JavaOne 2010-Konferenz. Auch <http://www.oracle.com/us/corporate/press/193190>.

2 *Just Another Vague Acronym* (etwa »bloß ein weiteres unbestimmtes Akronym«)

Zu jener Zeit arbeitete James Gosling am SGML-Editor *Imagination*. Er entwickelte in C++ und war mit dieser Sprache ebenfalls nicht zufrieden. Aus diesem Unmut heraus entstand die neue Sprache *Oak*. Der Name fiel Gosling ein, als er aus dem Fenster seines Arbeitsraums schaute – und eine Eiche erblickte (engl. *oak*), doch vielleicht ist das nur eine Legende, denn Oak steht auch für *Object Application Kernel*. Patrick Naughton startete im Dezember 1990 das Green-Projekt, in das Gosling und Mike Sheridan involviert waren. Überbleibsel aus dem Green-Projekt ist der *Duke*, der zum bekannten Symbol wurde.³

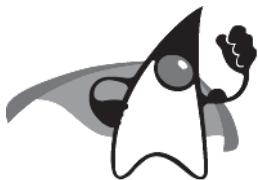


Abbildung 1.1: Der Duke, Symbol für Java

Die Idee hinter diesem Projekt war, Software für interaktives Fernsehen und andere Geräte der Konsumelektronik zu entwickeln. Bestandteile dieses Projekts waren das Betriebssystem Green-OS, Goslings Interpreter Oak und einige Hardwarekomponenten. Joy zeigte den Mitgliedern des Green-Projekts seinen Further-Aufsatz und begann mit der Implementierung einer grafischen Benutzeroberfläche. Gosling schrieb den Original-Compiler in C, und anschließend entwarfen Naughton, Gosling und Sheridan den Runtime-Interpreter ebenfalls in C – die Sprache C++ kam nie zum Einsatz. Oak führte die ersten Programme im August 1991 aus. So entwickelte das Green-Dream-Team ein Gerät mit der Bezeichnung *7 (*Star Seven*), das es im Herbst 1992 intern vorstellte. Der ehemalige Sun-Chef Scott McNealy (der nach der Übernahme von Oracle im Januar 2010 das Unternehmen verließ) war von *7 beeindruckt, und aus dem Team wurde im November die Firma *First Person, Inc.* Nun ging es um die Vermarktung von Star Seven.

Anfang 1993 hörte das Team, dass Time Warner ein System für Set-Top-Boxen suchte (Set-Top-Boxen sind elektronische Geräte für Endbenutzer). First Person richtete den Blick vom Consumer-Markt auf die Set-Top-Boxen. Leider zeigte sich Time Warner später nicht mehr interessiert, aber First Person entwickelte (sich) weiter. Nach vielen Richtungswechseln konzentrierte sich die Entwicklung auf das *World Wide Web* (kurz *Web* genannt, selten *W3*). Die Programmiersprache sollte Programmcode über das Netzwerk empfangen können, und fehlerhafte Programme sollten keinen Schaden anrichten.

³ Er sieht ein bisschen wie ein Zahn aus und könnte deshalb auch die Werbung eines Zahnarztes sein. Das Design stammt übrigens von Joe Palrang.

Damit konnten die meisten Konzepte aus C(++) schon abgehakt werden – Zugriffe über ungültige Zeiger, die wild den Speicher beschreiben, sind ein Beispiel. Die Mitglieder des ursprünglichen Projektteams erkannten, dass Oak alle Eigenschaften aufwies, die nötig waren, um es im Web einzusetzen – perfekt, obwohl ursprünglich für einen ganz anderen Zweck entwickelt. Die Sprache Oak erhielt den Namen *Java*, da der Name Oak, wie sich später herausstellte, aus Gründen des Copyrights nicht verwendet werden konnte: Eine andere Programmiersprache schmückte sich bereits mit diesem Namen. Nach der Überlieferung fiel die Entscheidung für den Namen Java in einem Coffeeshop. In Java führte Patrick Naughton den Prototyp des Browsers *WebRunner* vor, der an einem Wochenende entstanden sein soll. Nach geringfügiger Überarbeitung durch Jonathan Payne wurde der Browser *HotJava* getauft und im Mai auf der SunWorld '95 der Öffentlichkeit vorgestellt.

Zunächst konnten sich nur wenige Anwender mit HotJava anfreunden. So war es ein großes Glück, dass Netscape sich entschied, die Java-Technologie zu lizenziieren. Sie wurde in der Version 2.0 des *Netscape Navigators* implementiert. Der Navigator kam im Dezember 1995 auf den Markt. Im Januar 1996 wurde das JDK 1.0 freigegeben, was den Programmierern die erste Möglichkeit gab, Java-Applikationen und Web-Applets (Applet: »A Mini Application«) zu programmieren. Kurz vor der Fertigstellung des JDK 1.0 gründeten die verbliebenen Mitglieder des Green-Teams die Firma *JavaSoft*. Und so begann der Siegeszug.

Wo ist die Sonne? Oracle übernimmt Sun Microsystems 2010

Die Entwicklung von Java stammte ursprünglich von Sun Microsystems, einem Unternehmen mit langer Tradition im Bereich Betriebssysteme und Hardware. Sun hat viele Grundlagen für moderne IT-Systeme geschaffen, aber vielen war es nur durch Java bekannt. Das führte auch dazu, dass an der Wertpapierbörse im August 2007 die Kursbezeichnung der Aktie SUNW durch das neue Aktiensymbol JAVA ersetzt wurde.

Sun Microsystems ging es als Unternehmen nie so wirklich gut. Bekannt und respektiert für seine Produkte, fehlte es Sun am Geschick, aus den Produkten und Services Bares zu machen. 2008/2009 häufte sich ein Verlust von 2,2 Milliarden US-Dollar an, was im März 2009 zu Übernahmefantasien von IBM führte. Letztendlich schlug einen Monat später die Oracle Corporation zu und übernahm Sun Microsystems für 7,4 Milliarden Dollar, zusammen mit allen Rechten und Patenten für Java, MySQL, Solaris, Open-Office, VirtualBox und allen anderen Produkten. Einige Open-Source-Projekte hat Oracle mittlerweile eingestellt, aber die großen und kommerziell interessanten erfreuen sich bester Gesundheit.

1.2 Warum Java gut ist: die zentralen Eigenschaften

Java ist eine objektorientierte Programmiersprache, die sich durch einige zentrale Eigenschaften auszeichnet. Diese machen sie universell einsetzbar und für die Industrie als robuste Programmiersprache interessant. Da Java objektorientiertes Programmieren ermöglicht, können Entwickler moderne und wiederverwertbare Softwarekomponenten programmieren.

Zum Teil wirkt Java sehr konservativ, aber das liegt daran, dass die Sprachdesigner nicht alles das sofort einbauen, was im Moment gerade hipp ist (XML-Literale sind so ein Beispiel). Java nahm schon immer das, was sich in anderen Programmiersprachen als sinnvoll und gut herausgestellt hat, in den Sprachkern auf, verhinderte aber, Dinge aufzunehmen, die nur von sehr wenigen Entwicklern eingesetzt werden bzw. die öfter zu Fehlern führen. In den Anfängen stand C++ als Vorbild da, heute schiebt Java auf C# und Skriptsprachen.

Einige der zentralen Eigenschaften wollen wir uns im Folgenden anschauen und dabei auch zentrale Begriffe und Funktionsweisen beleuchten.

1.2.1 Bytecode

Zunächst ist Java eine Programmiersprache wie jede andere. Doch im Gegensatz zu herkömmlichen Übersetzern einer Programmiersprache, die in der Regel Maschinencode für eine spezielle Plattform (etwa Linux oder Windows) und einen bestimmten Prozessor (zum Beispiel für x86er-Mikroprozessoren oder Prozessoren der ARM-Architektur) generieren, erzeugt der Java-Compiler Programmcode, den sogenannten *Bytecode*, für eine virtuelle Maschine. Bytecode ist vergleichbar mit Mikroprozessorcode für einen erdachten Prozessor, der Anweisungen wie arithmetische Operationen, Sprünge und Weiteres kennt. Der Java-Compiler von Oracle und der Java-Compiler der Entwicklungsumgebung Eclipse sind selbst in Java implementiert und generieren diesen Bytecode (es gibt aber auch Java-Compiler in C++, wie den Jikes-Compiler⁴).

1.2.2 Ausführung des Bytecodes durch eine virtuelle Maschine

Damit der Programmcode des virtuellen Prozessors ausgeführt werden kann, führt nach der Übersetzungsphase die *Laufzeitumgebung* (auch *Runtime-Interpreter* ge-

⁴ <http://tutego.de/go/jikes>

nannt), also die *Java Virtual Machine (JVM)*, den Bytecode aus.⁵ Die Laufzeitumgebung lädt den Bytecode, prüft ihn und führt ihn in einer kontrollierten Umgebung aus. Die JVM bietet eine ganze Reihe von Zusatzdiensten wie einen Garbage-Collector, der Speicher aufräumt, sowie eine starke Typprüfung unter einem klar definierten Speicher- und Threading-Modell.

Es gibt unterschiedliche virtuelle Maschinen verschiedener Hersteller. Die wichtigste ist die JVM im *Oracle JDK* – kurz *JDK* – bzw. im OpenJDK. Die JVM ist frei und für Windows, Linux und Solaris erhältlich. Hersteller eigener Betriebssysteme wie IBM oder HP haben eigene Java-Laufzeitumgebungen, und auch Apple pflegte lange Zeit eine eigene JVM, bis Apple den Code an Oracle für das *OpenJDK* übergab. Weiterhin gibt es quelloffene JVMs, die zum Teil aus akademischen Arbeiten entstanden sind. Eine virtuelle Maschine selbst ist in der Regel in C++ programmiert, genauso wie einige Bibliotheken, und hat eine nicht zu unterschätzende Komplexität.

Java on a chip

Neben einer Laufzeitumgebung, die den Java-Bytecode interpretiert und in den Maschinencode eines Wirtssystems übersetzt, wurde auch ein Prozessor konstruiert, der in der Hardware Bytecode ausführt. Die Entwicklung ging damals verstärkt von Sun aus, und einer der ersten Prozessoren war *PicoJava*. Bei der Entwicklung des Prozessors stand nicht die maximale Geschwindigkeit im Vordergrund, sondern die Kosten pro Chip, um ihn in jedes Haushaltsgerät einbauen zu können. Das Interesse an Java auf einem Chip zieht nun nach einer Flaute wieder an, denn viele mobile Endgeräte müssen mit schnellen Ausführungseinheiten versorgt werden.

Die ARM-Technologie des Unternehmens *ARM Limited* erlaubt durch *Jazelle DBX* eine sehr schnelle Ausführung von Java-Bytecode. Mit dem Prozessor S5L8900 hat Samsung die ARM-Technologie ARM1176JZ(F)-S zusammen mit Speicherschnittstellen und Teilen für Connectivity, Peripherie und Multimedia-Möglichkeiten in Silizium gegossen, und als 667-MHz-Prozessor sitzt er nun in Apples iPhone. Die Ironie des Schicksals ist dabei, dass Apple im iPhone bisher keine Java-Unterstützung vorsieht.

Der *aj-102* und *aj-200* von *ajile Systems Inc.* sind weitere Prozessoren, die Java-Bytecode direkt ausführen; der aj-200 unterstützt auch direktes Threading. Und wenn wir

⁵ Die Idee des Bytecodes (das Satzprogramm *FrameMaker* schlägt hier als Korrektur »Bote Gottes« vor) ist schon alt. Die Firma *Datapoint* schuf um 1970 die Programmiersprache PL/B, die Programme auf Bytecode abbildet. Auch verwendet die Originalimplementierung von UCSD-Pascal, die etwa Anfang 1980 entstand, einen Zwischencode – kurz *p-code*.

den Pressemitteilungen von Azul Systems⁶ glauben können, gibt es auch bald einen 64-Bit-Prozessor mit 48 Kernen, der Java- und auch .NET-Bytecode ausführt. Ein Doppelherz tut auch Java gut.



1.2.3 Plattformunabhängigkeit

Eine zentrale Eigenschaft von Java ist seine *Plattformunabhängigkeit* bzw. *Betriebssystemunabhängigkeit*. Diese wird durch zwei zentrale Konzepte erreicht. Zum einen bindet sich Java nicht an einen bestimmten Prozessor oder eine bestimmte Architektur, sondern der Compiler generiert Bytecode, den eine Laufzeitumgebung dann abarbeitet. Zum anderen abstrahiert Java von den Eigenschaften eines konkreten Betriebssystems, schafft etwa eine Schnittstelle zum Ein-/Ausgabesystem oder eine API für grafische Oberflächen. Entwickler programmieren immer gegen eine Java-API aber nie gegen die API der konkreten Plattform, etwa die Windows- oder Unix-API. Die Java-Laufzeitumgebung bildet Aufrufe etwa auf Dateien für das jeweilige System ab, ist also Vermittler zwischen den Java-Programmen und der eigentlichen Betriebssystem-API.

Zwar ist das Konzept einer plattformneutralen Programmiersprache schon recht alt, doch erst in den letzten 10 Jahren kam mehr und mehr hinzu. Neben Java sind plattformunabhängige Programmiersprachen und Laufzeitumgebungen .NET-Sprachen wie C# auf der der CLR (*Common Language Runtime* – entspricht der Java VM), Perl, Python

⁶ <http://www.azulsystems.com/>

oder Ruby. Plattformunabhängigkeit ist schwer, denn die Programmiersprache und ein Bytecode produzierender Compiler ist nur ein Teil – der größere Teil ist die Laufzeitumgebung und eine umfangreiche API. Zwar ist auch C an sich eine portable Sprache, und ANSI C-Programme lassen sich von jedem C-Compiler auf jedem Betriebssystem mit Compiler übersetzen, aber das Problem sind die Bibliotheken, die über ein paar simple Dateioperationen nicht hinauskommen.

In Java 7 ändert sich die Richtung etwas, was sich besonders an der neuen API für die Dateisystemunterstützung ablesen lässt. Vor Java 7 war die Datei-Klasse so aufgebaut, dass die Semantik gewisser Operationen nicht ganz genau spezifiziert war und auf diese Weise sehr plattformabhängig war. Es gibt aber in der Datei-Klasse keine Operation, die nur auf einer Plattform zur Verfügung steht und andere Plattformen ausschließt. Das Credo lautete immer: Was nicht auf allen Plattformen existiert, kommt nicht in die Bibliothek.⁷ Mit Java 7 gibt es einen Wechsel: Nun sind plattformspezifische Dateieigenschaften zugänglich. Es bleibt abzuwarten, ob in der Zukunft in anderen API-Bereichen – vielleicht bei grafischen Oberflächen – noch weitere Beispiele hinzukommen.

1.2.4 Java als Sprache, Laufzeitumgebung und Standardbibliothek

Java ist nicht nur eine Programmiersprache, sondern ebenso ein Laufzeitsystem, was Oracle durch den Begriff »Java Platform« klarstellen will. So gibt es neben der Programmiersprache Java durchaus andere Sprachen, die eine Java-Laufzeitumgebung ausführen, etwa diverse Skriptsprachen wie *Groovy* (<http://groovy.codehaus.org/>), *JRuby* (<http://jruby.org/>), *Jython* (<http://www.jython.org/>) oder *Scala* (<http://www.scala-lang.org/>). Skriptsprachen auf der Java-Plattform werden immer populärer; sie etablieren eine andere Syntax, nutzen aber die JVM und die Bibliotheken.

Zu der Programmiersprache und JVM kommt ein Satz von Standardbibliotheken für grafische Oberflächen, Ein-/Ausgabe und Netzwerkoperationen. Das bildet die Basis für höherwertige Dienste wie Datenbankanbindungen oder Web-Services. Integraler Bestandteil der Standardbibliothek seit Java 1.0 sind weiterhin *Threads*. Sie sind leicht zu erzeugende Ausführungsstränge, die unabhängig voneinander arbeiten können. Mittlerweile unterstützen alle populären Betriebssysteme diese »leichtgewichtigen Prozesse« von Haus aus, sodass die JVM diese parallelen Programmteile nicht nachbilden muss, sondern auf das Betriebssystem verweisen kann. Bei den neuen Multi-Core-Pro-

⁷ Es gibt sie durchaus, die Methoden, die nur zum Beispiel auf Windows zur Verfügung stehen. Aber dann liegen sie nicht in einem java- oder javax-Paket, sondern in einem internen Paket.

zessoren sorgt das Betriebssystem für eine optimale Ausnutzung der Rechenleistung, da Threads wirklich nebenläufig arbeiten können.

Zu den Standardbibliotheken kommen dann weitere kommerzielle oder quelloffene Bibliotheken hinzu. Egal, ob es darum geht, PDF-Dokumente zu schreiben, Excel-Dokumente zu lesen, in SAP Daten zu übertragen oder bei einem Wincor-Bankautomaten den Geldauswurf zu steuern – für all das gibt es Java-Bibliotheken.

1.2.5 Objektorientierung in Java

Java ist als Sprache entworfen worden, die es einfach machen sollte, große, fehlerfreie Anwendungen zu schreiben. In C-Programmen erwartet uns statistisch gesehen alle 55 Programmzeilen ein Fehler. Selbst in großen Softwarepaketen (ab einer Million Codezeilen) findet sich, unabhängig von der zugrunde liegenden Programmiersprache, im Schnitt alle 200 Programmzeilen ein Fehler. Selbstverständlich gilt es, diese Fehler zu beheben, obwohl bis heute noch keine umfassende Strategie für die Softwareentwicklung im Großen gefunden wurde. Viele Arbeiten der Informatik beschäftigen sich mit der Frage, wie Tausende Programmierer über Jahrzehnte miteinander arbeiten und Software entwerfen können. Dieses Problem ist nicht einfach zu lösen und wurde im Zuge der Softwarekrise Mitte der 1960er-Jahre heftig diskutiert.

Eine Laufzeitumgebung eliminiert viele Probleme technischer Natur. *Objektorientierte Programmierung* versucht, die Komplexität des Software-Problems besser zu modellieren. Die Philosophie ist, dass Menschen objektorientiert denken und eine Programmierumgebung diese menschliche Denkweise abbilden sollte. Genauso wie Objekte in der realen Welt verbunden sind und kommunizieren, muss es auch in der Softwarewelt möglich sein. Objekte bestehen aus *Eigenschaften*; das sind Dinge, die ein Objekt »hat« und »kann«. Ein Auto »hat« Räder und einen Sitz und »kann« beschleunigen und bremsen. Objekte entstehen aus *Klassen*, das sind Beschreibungen für den Aufbau von Objekten.

Die Sprache Java ist nicht bis zur letzten Konsequenz objektorientiert, so wie Smalltalk es vorbildlich demonstriert. Primitive Datentypen wie Ganzzahlen oder Fließkomma-zahlen werden nicht als Objekte verwaltet. Als Grund für dieses Design wird genannt, dass der Compiler und die Laufzeitumgebung mit der Trennung besser in der Lage wären, die Programme zu optimieren. Allerdings zeigt die virtuelle Maschine von Microsoft für die .NET-Plattform und andere moderne Programmiersprachen, dass auch ohne die Trennung eine gute Performance möglich ist.

1.2.6 Java ist verbreitet und bekannt

Unabhängig von der Leistungsfähigkeit einer Sprache zählen am Ende doch nur betriebswirtschaftliche Faktoren: Wie schnell und billig lässt sich ein vom Kunden gewünschtes System bauen, und wie stabil und änderungsfreundlich ist es? Dazu kommen Fragen wie: Wie sieht der Literaturmarkt aus, wie die Ausbildungswege, woher bekommt ein Team einen Entwickler oder Consultant, wenn es brennt? Dies sind nicht unbedingt Punkte, die Informatiker beim Sprachvergleich auf die erste Stelle setzen, sie sind aber letztendlich für den Erfolg einer Software-Plattform entscheidend. Fast jede Universität lehrt Java, und mit Java ist ein Job sicher. Konferenzen stellen neue Trends vor und schaffen Trends. Diese Kette ist nicht zu durchbrechen, und selbst wenn heute eine neue Super-Sprache mit dem Namen »Bali« auftauchen würde, würde es Jahre dauern, bis ein vergleichbares System geschaffen wäre. Wohlgernekt: Das sagt nichts über die Innovations- oder Leistungsfähigkeit aus, nur über die Marktsättigung, aber dadurch wird Java eben für so viele interessant.

Java-Entwickler sind glücklich



Andrew Vos hat sich Kommentare angeschaut, mit denen Entwickler ihre Programme in die Versionsverwaltung einpflegen.⁸ Dabei zählt er, wie viele »böse« Wörter wie »shit«, »omg«, »wtf« beim Check-in vorkommen. Seine Herangehensweise ist zwar statistisch nicht ganz ordentlich, aber bei seinen untersuchten Projekten stehen Java-Entwickler recht gut da und haben wenig zu fluchen. Die Kommentare sind amüsant zu lesen und geben unterschiedliche Erklärungen, etwa dass JavaScript-Programmierer eigentlich nur über den IE fluchen, aber nicht über die JavaScript an sich, und dass Python-Programmierer zum Fluchen zu anständig sind.

1.2.7 Java ist schnell: Optimierung und Just-in-Time Compilation

Die Laufzeitumgebung von Java 1.0 startete mit einer puren Interpretation des Bytecodes. Das bereitete massive Geschwindigkeitsprobleme, denn beim Interpretieren muss die Arbeit eines Prozessors – das Erkennen, Dekodieren und Ausführen eines Befehls – noch einmal in Software wiederholt werden; das kostet viel Zeit. Java-Programme der ersten Stunde waren daher deutlich langsamer als übersetzte C(++)-Programme und brachten Java den Ruf ein, eine langsame Sprache zu sein.

⁸ <http://andrewvos.com/2011/02/21/amount-of-profanity-in-git-commit-messages-per-programming-language/>

Die Technik der *Just-in-Time-(JIT-)Compiler*⁹ war er erste Schritt, das Problem anzugehen. Ein JIT-Compiler beschleunigt die Ausführung der Programme, indem er zur Laufzeit den Bytecode, also die Programmanweisungen der virtuellen Maschine, in Maschinencode der jeweiligen Plattform übersetzt. Anschließend steht ein an die Architektur angepasstes Programm im Speicher, das der physikalische Prozessor ohne Interpretation schnell ausführt. Mit dieser Technik entspricht die Geschwindigkeit der von anderen übersetzten Sprachen. Jedoch übersetzt ein guter JIT nicht alles, sondern versucht über diverse Heuristiken herauszufinden, ob sich eine Übersetzung – die ja selbst Zeit kostet – überhaupt lohnt. Die JVM beginnt daher immer mit einer Interpretation und wechselt dann in einen Compiler-Modus, wenn es nötig wird. Somit ist Java im Grunde eine kompilierte, aber auch interpretierte Programmiersprache – von der Ausführung durch Hardware einmal abgesehen. Vermutlich ist der Java-Compiler in der JVM der am häufigsten laufende Compiler überhaupt.

Der JIT-Compiler von Sun wurde immer besser und entwickelte sich weiter zu einer Familie von virtuellen Maschinen, die heute unter dem Namen *HotSpot* bekannt sind. Das Besondere ist, dass HotSpot die Ausführung zur Laufzeit überwacht und »heiße« (sprich: kritische) Stellen findet, etwa Schleifen mit vielen Wiederholungen – daher auch der Name »HotSpot«. Daraufhin steuert die JVM ganz gezielt Übersetzungen und Optimierungen. Zu den Optimierungen gehören Klassiker, wie das Zusammenfassen von Ausdrücken, aber auch viele dynamische Optimierungen fallen in diesen Bereich, zu denen ein statischer C++-Compiler nicht in der Lage wäre, weil ihm der Kontext fehlt.¹⁰ Zudem kann die JVM Bytecode zu jeder Zeit nachladen, der wie alle schon geladenen Teile genauso optimiert wird. Der neu eingeführte Programmcode kann sogar alte Optimierungen und Maschinencode ungültig machen, den dann die JVM neu übersetzt.

HotSpot steht genauso wie das Laufzeitsystem unter der freien GPL-Lizenz und ist für jeden einsehbar. Die JVM ist hauptsächlich in C++ programmiert, aber aus Performance-Gründen befinden sich dort auch Teile in Maschinencode, was die Portierung nicht ganz einfach macht. Das *Zero-Assembler Project* (<http://openjdk.java.net/projects/zero/>) hat sich zum Ziel gesetzt, HotSpot ohne Maschinencode zu realisieren, sodass eine Portierung einfacher ist. Seit dem JDK 1.6.0_04-b12, das Ende 2007 veröffentlicht wurde, hat die HotSpot VM eine eigene Entwicklung und Versionsnummer, die für die letzten Java 6-Versionen bei über 20 liegt.

⁹ Diese Idee ist auch schon alt: HP hatte um 1970 JIT-Compiler für BASIC-Maschinen.

¹⁰ Dynamische Methodenaufrufe sind in der Regel sehr schnell, weil die JVM die Hierarchie kennt.

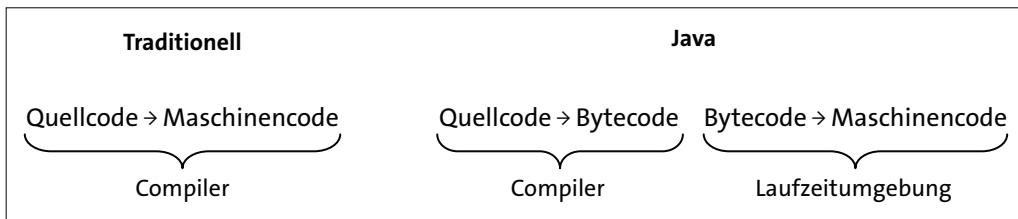


Abbildung 1.2: Traditioneller Compiler und Java-Compiler mit Laufzeitumgebung

1.2.8 Das Java-Security-Modell

Das Java-Security-Modell gewährleistet den sicheren Programmablauf auf den verschiedensten Ebenen. Der Verifier liest Code und überprüft die strukturelle Korrektheit und Typsicherheit. Weist der Bytecode schon Fehler auf, kommt der Programmcode erst gar nicht zur Ausführung. Die Prüfung ist wichtig, denn ein *Klassenlader* (engl. *class loader*) kann Klassendateien von überall her laden. Während vielleicht dem Bytecode aus dem lokalen Laufwerk vertraut werden kann, gilt das mitunter nicht für Code, der über ein ungesichertes Netzwerk übertragen wurde, wo ein Dritter plötzlich Schadcode einfügt (*Man-in-the-Middle-Angriff*). Ist der Bytecode korrekt in der virtuellen Maschine angemeldet, folgen weitere Prüfungen. So sind etwa (mit entsprechender Anpassung) keine Lese-/Schreibzugriffe auf private Variablen möglich. Treten Sicherheitsprobleme auf, werden diese durch Exceptions zur Laufzeit gemeldet – so kommt es etwa zu keinen Pufferüberläufen. Auf der Programmebene überwacht ein *Security-Manager* Zugriffe auf das Dateisystem, die Netzwerk-Ports, externe Prozesse und weitere Systemressourcen. Das Sicherheitsmodell kann vom Programmierer erweitert und über Konfigurationsdateien einfach konfiguriert werden.

1.2.9 Zeiger und Referenzen

In Java gibt es keine Zeiger (engl. *pointer*), wie sie aus anderen Programmiersprachen bekannt und gefürchtet sind. Da eine objektorientierte Programmiersprache ohne Verweise aber nicht funktioniert, werden *Referenzen* eingeführt. Eine Referenz repräsentiert ein Objekt, und eine Variable speichert diese Referenz. Die Referenz hat einen Typ, der sich nicht ändern kann. Ein Auto bleibt ein Auto und kann nicht als Laminiersystem angesprochen werden. Eine Referenz unter Java ist nicht als Zeiger auf Speicherbereiche zu verstehen, obwohl sie intern durchaus so implementiert werden kann; das ist für den Benutzer aber nie sichtbar.

zB Beispiel *

Das folgende Programm zeigt, dass das Pfuschen in C++ leicht möglich ist und wir Zugriff auf private Elemente über eine Zeigerarithmetik bekommen können. Für uns Java-Programmierer ist dies ein abschreckendes Beispiel.

```
#include <cstring>
#include <iostream>

using namespace std;

class VeryUnsafe
{
public:
    VeryUnsafe() { strcpy( password, "HaL9124f/aa" ); }
private:
    char password[ 100 ];
};

int main()
{
    VeryUnsafe badguy;
    char *pass = reinterpret_cast<char*>( & badguy );
    cout << "Password: " << pass << endl;
}
```

Dieses Beispiel demonstriert, wie problematisch der Einsatz von Zeigern sein kann. Der zunächst als Referenz auf die Klasse `VeryUnsafe` gedachte Zeiger `badguy` mutiert durch die explizite Typumwandlung zu einem Char-Pointer `pass`. Problemlos können über diesen die Zeichen byteweise aus dem Speicher ausgelesen werden. Dies erlaubt auch einen indirekten Zugriff auf die privaten Daten.

In Java ist es nicht möglich, auf beliebige Teile des Speichers zuzugreifen. Auch sind private Variablen erst einmal sicher.¹¹ Der Compiler bricht mit einer Fehlermeldung ab – beziehungsweise löst das Laufzeitsystem eine Ausnahme (Exception) aus –, wenn das Programm einen Zugriff auf eine private Variable versucht.

¹¹ Ganz stimmt das allerdings nicht. Mit Reflection lässt sich da schon etwas machen, wenn die Sicherheitseinstellungen das nicht verhindern.

1.2.10 Bring den Müll raus, Garbage-Collector!

In Programmiersprachen wie C++ lässt sich etwa die Hälfte der Fehler auf falsche Speicher-Allokation zurückführen. Mit Objekten zu arbeiten, bedeutet unweigerlich, sie anzulegen und zu löschen. Die Java-Laufzeitumgebung kümmert sich jedoch selbstständig um die Verwaltung dieser Objekte – die Konsequenz: Sie müssen nicht freigegeben werden, ein *Garbage-Collector* (kurz GC) entfernt sie. Der GC ist Teil des Laufzeitsystems von Java. Nach dem expliziten Generieren eines Objekts überwacht Java permanent, ob das Objekt noch gebraucht wird, also referenziert wird. Umgekehrt bedeutet das aber auch: Wenn auf einem Objekt vielleicht noch ein heimlicher Verweis liegt, kann der GC das Objekt nicht löschen. Diese sogenannten *hängenden Referenzen* sind ein Ärgernis und zum Teil nur durch längere Debugging-Sitzungen zu finden.

Der GC ist ein nebenläufiger Thread im Hintergrund, der nicht referenzierte Objekte findet, markiert und dann von Zeit zu Zeit entfernt. Damit macht der Garbage-Collector die Funktionen `free()` aus C oder `delete()` aus C++ überflüssig. Wir können uns über diese Technik freuen, da viele Probleme damit verschwunden sind. Nicht freigegebene Speicherbereiche gibt es in jedem größeren Programm, und falsche Destruktoren sind vielfach dafür verantwortlich. An dieser Stelle sollte nicht verschwiegen werden, dass es auch ähnliche Techniken für C(++) gibt¹² und dass alle modernen Programmiersprachen (bzw. deren Laufzeitumgebungen) einen GC besitzen.

1.2.11 Ausnahmebehandlung

Java unterstützt ein modernes System, um mit Laufzeitfehlern umzugehen. In die Programmiersprache wurden *Ausnahmen* (engl. *exceptions*) eingeführt: Objekte, die zur Laufzeit generiert werden und einen Fehler anzeigen. Diese Problemstellen können durch Programmkonstrukte gekapselt werden. Die Lösung ist in vielen Fällen sauberer als die mit Rückgabewerten und unleserlichen Ausdrücken im Programmfluss. In C++ gibt es ebenso Exceptions, die aber nicht so intensiv wie in Java benutzt werden.

Aus Geschwindigkeitsgründen überprüft C(++)¹³ die Array-Grenzen (engl. *range checking*) standardmäßig nicht, was ein Grund für viele Sicherheitsprobleme ist. Ein fehler-

¹² Ein bekannter Garbage-Collector stammt von Hans-J. Boehm, Alan J. Demers und Mark Weiser. Er ist unter <http://tutego.de/go/boehmgc> zu finden. Der Algorithmus arbeitet jedoch konservativ, das heißt, er findet nicht garantiert alle unerreichbaren Speicherbereiche, sondern nur einige. Eingesetzt wird der Boehm-Demers-Weiser-GC unter anderem in der X11-Bibliothek. Dort sind die `malloc()`- und `free()`-Funktionen einfach durch neue Methoden ausgetauscht worden.

¹³ In C++ ließe sich eine Variante mit einem überladenen Operator lösen.

hafter Zugriff auf das Element $n + 1$ eines Feldes der Größe n kann zweierlei bewirken: Ein Zugriffsfehler tritt auf, oder – viel schlimmer – andere Daten werden beim Schreibzugriff überschrieben, und der Fehler ist nicht mehr nachvollziehbar.

Das Laufzeitsystem von Java überprüft automatisch die Grenzen eines Arrays. Diese Überwachungen können auch nicht abgeschaltet werden, wie es Compiler anderer Programmiersprachen mitunter erlauben. Eine clevere Laufzeitumgebung findet heraus, ob keine Überschreitung möglich ist, und optimiert diese Abfrage dann weg; Feldüberprüfungen kosten daher nicht mehr die Welt und machen sich nicht automatisch in einer schlechteren Performance bemerkbar.

1.2.12 Einfache Syntax der Programmiersprache Java

Die Syntax von Java ist bewusst einfach gehalten worden und strotzt nicht vor Operatoren oder Komplexität wie C++ oder Perl. Java erbte eine einfache und grundlegende Syntax wie die geschweiften Klammern von C, vermied es aber, die Syntax mit allen möglichen Dingen zu überladen. Da Programme häufiger gelesen als geschrieben werden, muss eine Syntax klar und durchgängig sein – je leichter Entwickler auf den ersten Blick sehen, was passiert, desto besser ist es.

»Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live« – John Woods

Es hat keinen Wert an sich, wenn Programme einfach nur kompakt sind, aber die Zeit, die ein Mensch zum Verstehen braucht, exponentiell steigt. Auch wenn das folgende Perl-Beispiel aus den »The Fifth Obfuscated Perl Contest Results«¹⁴ ganz bewusst als unleserliches Programm entworfen wurde, so wird jedem Betrachter schon beim Anblick schwummerig:

```
#::: ::-| ::-| .-. :||-: 0-| .-| ::||-| ..|-. :||  
open(Q,$0);while(<Q>){if(/^#(.*)$/){for(split('-', $1)){\$q=0;for(split){\$s/\|  
/..:/xg;\$s/://g;\$Q=\$_?length:\$_;\$q+=\$q?\$Q:\$Q*20;}print chr(\$q);}}}}print"\n";  
#.. : :||-| .||-| :||-| ::||-| ||-: :||-| ..|
```

Eine einfachere Syntax lässt sich als Fluch (mehr Schreibarbeit) und auch als Segen (in der Regel leichter verständlich) auffassen. Java macht vieles richtig, aber es gibt ganz klar Stellen, an denen es hätte auch einfacher sein können (Stichwort *Generics*). Den-

¹⁴ http://www.foo.be/docs/tpj/issues/vol5_3/tpj0503-0014.html

noch haben die Java-Entwickler gut daran getan, auf Konstrukte zu verzichten; zwei aus C(++) ausgelassene Fähigkeiten sollen exemplarisch vorgestellt werden.

In Java gibt es keine benutzerdefinierten überladenen Operatoren

Wenn wir einen Operator wie das Pluszeichen verwenden und damit Ausdrücke addieren, tun wir dies meistens mit bekannten Rechengrößen wie Fließkommazahlen (Gleitkommazahlen) oder Ganzzahlen. Da das gleiche Operatorzeichen auf unterschiedlichen Datentypen gültig ist, nennt sich so ein Operator »überladen«. Operatoren wie +, -, *, / sind für Ganzzahlen und Gleitkommazahlen ebenso überladen wie die Operatoren Oder, Und oder Xor für Ganzzahlen und boolesche Werte. Der Vergleichsoperator == beziehungsweise != ist ebenfalls überladen, denn er lässt sich bei allen Zahlen, aber auch bei Wahrheitswerten oder Objektverweisen verwenden. Ein auffälliger überladener Operator ist das Pluszeichen bei Zeichenketten. Strings können damit leicht zusammengesetzt werden. Informatiker verwenden in diesem Zusammenhang auch gern das Wort *Konkatenation* (selten *Katenation*). Bei den Strings "Hallo" + " " + "du da" ist "Hallo du da" die Konkatenation der Zeichenketten.

Einige Programmiersprachen erlauben es, die vorhandenen Operatoren mit neuer Bedeutung zu versehen. In Java ist das nicht möglich. In C++ ist das Überladen von Operatoren erlaubt, sodass etwa das Pluszeichen dafür genutzt werden kann, geometrische Punktobjekte zu addieren, Brüche zu teilen oder eine Zeile in eine Datei zu schreiben. Repräsentieren die Objekte mathematische Konstrukte, ist es ganz praktisch, wenn Operationen über kurze Operatorzeichen benannt werden und nicht über längere Methoden – ein `matrix1.add(matrix2)` ist sperriger als ein `matrix1 + matrix2`. Obwohl benutzerdefinierte überladene Operatoren zuweilen ganz praktisch sind, verführte die Möglichkeit oft zu unsinnigem Gebrauch in C++. Daher haben die Sprachdesigner das für Java nicht vorgesehen, aber einige alternative Sprachen auf der JVM ermöglichen dies, denn es ist eine Sprachbeschränkung und keine Beschränkung der virtuellen Maschine.

Kein Präprozessor für Textersetzung *

Viele C(++)-Programme enthalten Präprozessor-Direktiven wie `#define`, `#include` oder `#if` zum Einbinden von Prototyp-Definitionen oder zur bedingten Compilierung. Einen solchen Präprozessor gibt es in Java aus unterschiedlichen Gründen nicht:

- Header-Dateien sind in Java nicht nötig, da der Compiler die benötigten Informationen wie Methodensignaturen direkt aus den Klassendateien liest.

- Da in Java die Datentypen eine feste, immer gleiche Länge haben, entfällt die Notwendigkeit, abhängig von der Plattform unterschiedliche Längen zu definieren.
- Pragma-Steuerungen sind im Programmcode unnötig, da die virtuelle Maschine ohne äußere Steuerung Programmoptimierungen vornimmt.

Ohne den Präprozessor sind schmutzige Tricks wie `#define private public` oder Makros, die Fehler durch eine doppelte Auswertung erzeugen, von vornherein ausgeschlossen. Im Übrigen findet sich der Private/Public-Hack im Quellcode von StarOffice. Die obere Definition ersetzt jedes Auftreten von `private` durch `public` – mit der Konsequenz, dass der Zugriffsschutz ausgehebelt ist.

Ohne Präprozessor ist auch die bedingte Kompilierung mit `#ifdef` nicht mehr möglich. Innerhalb von Anweisungsblöcken können wir uns in Java damit behelfen, Bedingungen der Art `if (true)` oder `if (false)` zu formulieren; über den Schalter `-D` auf der Kommandozeile lassen sich Variablen einführen, die dann eine if-Anweisung über `System.getProperty()` zur Laufzeit prüfen kann.¹⁵

1.2.13 Java ist Open Source

Schon seit Java 1.0 gibt es den Quellcode der Standardbibliotheken (falls er beim JDK mitinstalliert wurde, befindet er sich im Wurzelverzeichnis unter dem Namen `src.zip`), und jeder Interessierte konnte einen Blick auf die Implementierung werfen. Zwar legte Sun damals also die Implementierungen offen, doch weder die Laufzeitumgebung noch der Compiler oder die Bibliotheken standen unter einer akzeptierten Open-Source-Lizenz. Zehn Jahre seit der ersten Freigabe von Java gab es Forderungen an Sun, die gesamte Java-Plattform unter eine bekanntere Lizenzform wie die *GNU General Public License* (GPL) oder die BSD-Lizenz zu stellen. Dabei deutete Jonathan Schwartz in San Francisco bei der JavaOne-Konferenz 2006 schon an: »It's not a question of whether we'll open source Java, now the question is how.« War die Frage also statt des »Ob« ein »Wie«, kündigte Rich Green bei der Eröffnungsrede der JavaOne-Konferenz im Mai 2007 die endgültige Freigabe von Java als *OpenJDK*¹⁶ unter der Open-Source-Lizenz GPL 2 an. Dem war Ende 2006 die Freigabe des Compilers und der virtuellen Maschine vorausgegangen.

¹⁵ Da aber besonders bei mobilen Endgeräten Präprozessor-Anweisungen für unterschiedliche Geräte praktisch sind, gibt es Hersteller-Erweiterungen wie die von NetBeans (<http://tutego.de/gonebpreprocessor>).

¹⁶ <http://openjdk.java.net/>

Die Geschichte ist allerdings noch ein wenig komplizierter. Obwohl OpenJDK nun unter der GPL stand, enthielt es doch Teile wie den Font-Renderer, Sound-Unterstützung, Farbmanagement oder SNMP-Code, die als binäre Pakete beigelegt wurden, weil etwa die Rechte zur Veröffentlichung fehlten. Sun nennt diese Teile, die etwa 4 % vom JDK 6 ausmachen, *belasteten Code* (engl. *encumbered code*)¹⁷. Das hinderte puristische Linux-Distributoren daran, OpenJDK auszuliefern. RedHat startete im Juni 2007 das Projekt *IcedTea*, um diese binären Teile auf der Basis des OpenJDK durch GPL-Software zu ersetzen. So basiert der Font-Renderer zum Beispiel auf *FreeType*¹⁸ und das Farbmanagement auf *little CMS*¹⁹. Mit diesen Ersetzungen erfüllte das OpenJDK mit IcedTea im Juni 2008 die Anforderungen des *Technology Compatibility Kit* (TCK) von Sun und ist in der Öffentlichkeit seither unter dem Namen *OpenJDK 6* bekannt. Daraufhin floss das OpenJDK 6 plus den Ersetzungen unter der GPLv2 in Linux-Distributionen wie *Fedora* und *Debian* ein.

Das OpenJDK bildet die Basis von Java 7, und jeder Entwickler kann sein eigenes Java zusammenstellen und beliebige Erweiterungen veröffentlichen. Damit ist der Schritt vollzogen, dass auch Java auf Linux-Distributionen Platz finden darf, die Java vorher aus Lizenzgründen nicht integrieren wollten.

Auch wenn es sich so anhört, als ob das Oracle JDK bzw. OpenJDK das Gleiche sei, ist das nicht ganz richtig: Zwar basieren Oracle JDK und OpenJDK auf den gleichen Quellen (bei der Version 7 etwa zu 95 %), doch sind beim Oracle JDK immer noch proprietäre Dinge enthalten, und nicht alles ist hundertprozentig quelloffen und GPL. Das gilt für die Version 7 wie für die Version 6. Das Oracle JDK steht unter der *Binary Code License*; genau die muss jeder abnicken, der das JDK von der Webseite laden möchte.

Bei der 6er-Reihe kommt noch eine Besonderheit dazu, wie es die Versionsnummern²⁰ ganz gut zeigen. Während das Oracle JDK zum Beispiel im Juni 2011 bei Versionsnummer 1.6.0_26-b03 steht, ist das OpenJDK bei Version 6 b22. Die Versionsnummern sind deshalb völlig unabhängig, weil beide Projekte auch unabhängig voneinander laufen. Das hat mit der Geschichte zu tun. Nach der Entwicklung des JDK 6, das *nicht* unter der GPL steht, ging es mit dem JDK 7 logisch weiter. Aus dem JDK 7 (Build 10) entstand dann OpenJDK, das heute mit der Versionsnummer *OpenJDK 7* genannt wird. OpenJDK 7 und

17 <http://www.sun.com/software/opensource/java/faq.jsp#h>

18 <http://www.freetype.org/>

19 <http://www.littlecms.com/>

20 <https://gist.github.com/925323>

JDK 7 entwickeln sind Hand in Hand, und Code-Änderungen gehen mal in die eine Richtung und mal in die andere.

Jetzt kommt die Besonderheit: Das OpenJDK 6 entstand nicht, wie vermutet werden könnte, aus dem Oracle JDK 1.6, sondern aus dem OpenJDK 7 (Build 20). Es wurden nur Java 7-Eigenschaften entfernt. So läuft das auch bis heute: Die meisten Änderungen am OpenJDK 6 sind Backports von OpenJDK 7. Änderungen am OpenJDK 7 stammen überwiegend von Oracle, und häufig ist es die Firma RedHat, die diese Änderungen in OpenJDK 6 portiert. Zwischen dem OpenJDK 6 und dem JDK 1.6 gibt es einen Quellcodeaustausch bei Bug-Fixes, doch die Codebasis ist unterschiedlich. Oracle JDK 6 ist im Wartungsmodus, und großartige Veränderungen passieren bis auf Fehlerbereinigungen nicht. Oracle JDK 6 und JDK 7 sind immer noch die zentrale Versionen, die die Downloadseite von Oracle anbietet; das OpenJDK liegt auf einem extra Server <http://openjdk.java.net/>, die Binaries für Linux und Windows finden sich unter <http://jdk7.java.net/java-se-7-ri/>. Interessanterweise ist das OpenJDK auch die Referenzimplementierung für Java SE, nicht das Oracle JDK.

1.2.14 Wofür sich Java weniger eignet

Java ist als Programmiersprache für allgemeine Probleme entworfen worden und deckt große Anwendungsgebiete ab (*general-purpose language*). Das heißt aber auch, dass es für ausreichend viele Anwendungsfälle deutlich bessere Programmiersprachen gibt, etwa im Bereich Skripting, wo die Eigenschaft, dass jedes Java-Programm mindestens eine Klasse und eine Methode benötigt, eher störend ist, oder im Bereich von automatisierter Textverarbeitung, wo andere Programmiersprachen eleganter mit regulären Ausdrücken arbeiten können.

Auch dann, wenn extrem maschinen- und plattformabhängige Anforderungen bestehen, wird es in Java umständlich. Java ist plattformunabhängig entworfen worden, sodass alle Methoden auf allen Systemen lauffähig sein sollen. Sehr systemnahe Eigenschaften wie die Taktfrequenz sind nicht sichtbar, und sicherheitsproblematische Manipulationen wie der Zugriff auf bestimmte Speicherzellen (das PEEK und POKE) sind ebenso untersagt. Hier ist eine bei Weitem unvollständige Aufzählung von Dingen, die Java standardmäßig nicht kann:

- CD auswerfen
- Bildschirm auf der Textkonsole löschen, Cursor positionieren und Farben setzen
- auf niedrige Netzwerk-Protokolle wie ICMP zugreifen

- Microsoft Office fernsteuern
- Zugriff auf USB²¹ oder Firewire

Aus den genannten Nachteilen, dass Java nicht auf die Hardware zugreifen kann, folgt, dass die Sprache nicht so ohne Weiteres für die Systemprogrammierung eingesetzt werden kann. Treibersoftware, die Grafik-, Sound- oder Netzwerkkarten anspricht, lässt sich in Java nur über Umwege realisieren. Genau das Gleiche gilt für den Zugriff auf die allgemeinen Funktionen des Betriebssystems, die Windows, Linux oder ein anderes System bereitstellt. Typische System-Programmiersprachen sind C(++) oder Objective-C.

Aus diesen Beschränkungen ergibt sich, dass Java eine hardwarenahe Sprache wie C(++) nicht ersetzen kann. Doch das muss die Sprache auch nicht! Jede Sprache hat ihr bevorzugtes Terrain, und Java ist eine allgemeine Applikationsprogrammiersprache; C(++) darf immer noch für Hardwaretreiber und virtuelle Java-Maschinen herhalten.

Soll ein Java-Programm trotzdem systemnahe Eigenschaften nutzen – und das kann es mit entsprechenden Bibliotheken ohne Probleme –, bietet sich zum Beispiel der *native Aufruf* einer Systemfunktion an. Native Methoden sind Unterprogramme, die nicht in Java implementiert werden, sondern in einer anderen Programmiersprache, häufig in C(++)). In manchen Fällen lässt sich auch ein externes Programm aufrufen und so etwa die Windows-Registry manipulieren oder Dateirechte setzen. Es läuft aber immer darauf hinaus, dass die Lösung für jede Plattform immer neu implementiert werden muss.

1.2.15 Java im Vergleich zu anderen Sprachen

Beschäftigen sich Entwickler mit dem Design von Programmiersprachen, werden häufig existierende Spracheigenschaften auf ihre Tauglichkeit hin überprüft und dann in das Konzept aufgenommen. Auch Java ist eine sich entwickelnde Sprache, die Merkmale anderer Sprachen aufweist.

Java und C(++)

Java basiert syntaktisch stark auf C(++)), etwa bei den Datentypen, Operatoren oder Klammern, hat aber nicht alle Eigenschaften übernommen. In der geschichtlichen Kette wird Java gern als Nachfolger von C++ (und als Vorgänger von C#) angesehen, doch die Programmiersprache Java verzichtet bewusst auf problematische Konstrukte wie Zeiger.

²¹ Eigentlich sollte es Unterstützung für den Universal Serial Bus geben, doch Sun hat hier – wie leider auch an anderer Stelle – das Projekt *JSR-80: Java USB API* nicht weiterverfolgt.

Das Klassenkonzept – und damit der objektorientierte Ansatz – wurde nicht unwe sentlich durch SIMULA und Smalltalk inspiriert. Die Schnittstellen (engl. *interfaces*), die eine elegante Möglichkeit der Klassenorganisation bieten, sind an Objective-C angelehnt – dort heißen sie *Protocols*. Während Smalltalk alle Objekte dynamisch verwaltet und in C++ der Compiler statisch Klassen zu einem Programm kombiniert, mischt Java in sehr eleganter Form dynamisches und statisches Binden. Alle Klassen – optional auch von einem anderen Rechner über das Netzwerk – lädt die JVM zur Laufzeit. Selbst Methodenaufrufe sind über das Netz möglich.²² In der Summe lässt sich sagen, dass Java bekannte und bewährte Konzepte übernimmt und die Sprache sicherlich keine Revolution darstellt; moderne Skriptsprachen sind da weiter und übernehmen auch Konzepte aus funktionalen Programmiersprachen.

Java und JavaScript

Obacht ist beim Gebrauch des Namens »Java« geboten. Nicht alles, bei dem Java im Wortstamm auftaucht, hat tatsächlich mit Java zu tun: JavaScript hat keinen großen Bezug zu Java – bis auf Ähnlichkeiten bei den imperativen Konzepten. Die Programmiersprache wurde von Netscape entwickelt. Dazu ein Zitat aus dem Buch »The Java Developer's Resource«²³:

»Java and JavaScript are about as closely related as the Trump Taj Mahal in Atlantic City is to the Taj Mahal in India. In other words Java and Java-Script both have the word Java in their names. JavaScript is a programming language from Netscape which is incorporated in their browsers. It is superficially similar to Java in the same way C is similar to Java but differs in all important respects.«

Die Klassennutzung ist mit einem Prototyp-Ansatz in JavaScript völlig anders als in Java, und JavaScript lässt sich zu den funktionalen Programmiersprachen zählen, was Java nun wahrlich nicht ist.

Java und C#/.NET

Da C# kurz nach Java und nach einem Streit zwischen Microsoft und Sun erschien und die Sprachen zu Beginn syntaktisch sehr ähnlich gewesen sind, könnte leicht angenom-

²² Diese Möglichkeit ist unter dem Namen RMI (*Remote Method Invocation*) bekannt. Bestimmte Objekte können über das Netz miteinander kommunizieren.

²³ <http://www.cafeaulait.org/books/jdr/>, Kapitel 1. Das Buch wurde bei Prentice-Hall verlegt (ISBN 0135707897).

men werden, dass Java Pate für die Programmiersprache C#²⁴ stand. Doch das ist lange her. Mittlerweile hat C# eine so starke Eigendynamik entwickelt, dass Microsofts Programmiersprache viel innovativer ist als Java. C# ist im Laufe der Jahre komplex geworden, und Microsoft integriert ohne großen Abstimmungsprozess Elemente in die Programmiersprache, wo in der Java-Welt erst eine Unmenge von Personen diskutieren und abstimmen. Zeitweilig macht es den Eindruck, als könne Java nun auch endlich das, was C# bietet. So gesehen, profitiert Java heute von den Erfahrungen aus der C#-Welt.

Während Oracle für Java eine Aufteilung in das *Java SE* für die »allgemeinen« Programme und das *Java EE* als Erweiterung für die »großen« Enterprise-Systeme vornimmt, fließt bei Microsoft alles in *ein* Framework. Das .NET Framework ist natürlich größer als das Java-Framework, da sich mit .NET alles programmieren lässt, was Windows hergibt. Diese Eigenschaft fällt im Bereich GUI besonders auf, und das plattformunabhängige Java gibt dort weniger her.

Wäre nicht die Plattformunabhängigkeit, wäre es wohl ziemlich egal, ob große Systeme in Java oder einer .NET-Sprache entwickelt würden. *Mono* ist eine interessante Alternative zur Microsoft .NET-Entwicklungs- und Laufzeitumgebung, denn die quelloffene Nach-Implementierung läuft unter Linux, BSD, MacOS, iOS, Android und anderen Systemen. Allerdings bringt die Patentunsicherheit Unternehmen vom großen Mono-Einsatz ab, und wichtige Teile aus dem .NET-Framework wie *Windows Presentation Foundation* (WPF), *Windows Workflow Foundation* (WWF) oder *Entity Framework* fehlen. Zudem übernahm im April 2011 die Firma *Attachmate* das Unternehmen Novell, das Mono bisher leitete, und setzte erst einmal viele Mono-Entwickler auf die Straße. Einige ehemals von Novell bezahlte Mono-Entwickler schlüpften daraufhin bei der im Mai 2011 gegründeten Firma *Xamarin* unter.

Etwas zynisch lässt sich bemerken, dass Java vielleicht nur deshalb noch lebt, weil Microsoft Windows attraktiv machen möchte, nicht aber andere Plattformen stärken möchte, indem es C# und das .NET Framework quołoffen unter eine Open-Source-Lizenz stellt und die Laufzeitumgebung auf unterschiedliche Plattformen bringt. Ein Hoch auf Industriepolitik! Microsoft brachte mit *Silverlight* eine Art abgespecktes .NET auf unterschiedlichen Systemen heraus, doch es hatte keine Auswirkungen auf die Client-Applikationen und zog keine Java-Entwickler an. Microsoft selbst ist auch nicht mehr sonderlich von Silverlight überzeugt und sieht für Smartphone- und Web-Anwendungen eine Kombination von HTML5, CSS3 und JavaScript vor.

²⁴ In Microsoft-Dokumenten findet sich über Java kein Wort. Dort wird immer nur davon gesprochen, dass C# andere Sprachen, wie etwa C++, VB und Delphi, als Vorbilder hatte.

1.2.16 Java und das Web, Applets und JavaFX

Es ist nicht untertrieben, dem Web eine Schlüsselposition bei der Verbreitung von Java zuzuschreiben. Populär wurde Java in erster Linie durch die *Applets* – Java-Programme, die vom Browser dargestellt werden. Applets werden in einer HTML-Datei referenziert; der Browser holt sich eigenständig die Klassen und Ressourcen über das Netz und führt sie in einer virtuellen Maschine aus. Applets brachten erstmals Dynamik in die bis dahin statischen Webseiten – JavaScript kam erst später.

Obwohl Applets ganz normale Java-Programme sind, gibt es verständlicherweise einige Einschränkungen. So dürfen Applets nicht – es sei denn, sie sind signiert – auf das Dateisystem zugreifen und beliebig irgendwelche Dateien löschen, was Java-Applikationen problemlos können.

Netscape war eine der ersten Firmen, die einen Java-Interpreter in ihren Webbrowser integrierten. Heute bietet jeder Browser Java-Unterstützung, oft auch durch Oracles Hilfe, das mit dem Java-Plugin die jeweils neusten Java-Versionen in den Browser integriert. Ohne das Java-Plugin sähe die Unterstützung wohl anders aus. Der Internet Explorer (IE) von Microsoft akzeptiert zum Beispiel Applets, doch kommt er mit der MS-eigenen JVM nicht über die Version 1.1.4 hinaus. Oracle patcht hier fleißig, um auf allen populären Browsern immer die aktuellste und sicherste Java-Version anbieten zu können.

Java auf der Serverseite statt auf der Clientseite und im Browser

Obwohl das Web Java bekannt gemacht hat und dort viele Einsatzgebiete liegen, ist es nicht auf dieses Medium beschränkt. Nahezu alle großen IT-Unternehmen haben ihre Zuneigung zu dieser Sprache entdeckt. Es hat sich gezeigt, dass die Devise »write once, run anywhere« (WORA) auf der Serverseite weitgehend zutrifft. Java ist inzwischen wohl die wichtigste Sprache für die Gestaltung von Internet-Applikationen auf dem Server. Sie unterstützt strukturiertes und objektorientiertes Programmieren und ist ideal für größere Projekte, bei denen die Unsicherheiten von C++ vermieden werden sollen.

Nach dem anfänglichen Hype heißt es heute paradoxerweise oft, dass Java zu langsam für Client-Anwendungen sei. Dabei sind die virtuellen Maschinen aufgrund der Entwicklung von JIT-Compilern und der HotSpot-Technologie in den letzten Jahren sehr viel schneller geworden. Auch die Geschwindigkeit der Prozessoren ist ständig weiter gewachsen. Anwendungen, wie in Java geschriebene Entwicklungsumgebungen oder der SAP-Client, zeigen, dass auch auf der Clientseite Programme in angemessener Geschwindigkeit laufen können – entsprechend viel Arbeitsspeicher vorausgesetzt. Da ist Java nämlich mindestens so anspruchsvoll wie neue, bunte Betriebssysteme von MS.

Dennoch kommt Java auf der Clientseite nicht so richtig in Fahrt, und es ist ein bisschen so wie Linux, bei dem auch immer wieder gesagt wurde, »Das Jahr 2000+x ist DAS Jahr des Linux-Desktops«, woraus trotzdem nie etwas wurde.

RIA mit JavaFX

Java ist auf dem Desktop nicht besonders stark, und Applets spielen kaum (mehr) eine Rolle im Internet. Oracle nahm daher einen erneuten Anlauf, um im Bereich der Entwicklung von *Rich Internet Applications* (RIA) mitmischen zu können. Unter RIA wollen wir grafisch aufwändige Webanwendungen verstehen, die Daten aus dem Internet beziehen. Zwar beherrscht *Adobe Flash* hier fast zu 100 % das Feld, doch Microsoft ist mit Silverlight ebenfalls ein Marktteilnehmer, und Oracle möchte als strategischen Gründen nicht das Feld den Mitbewerbern überlassen. Daher veröffentlichte Oracle nach langer interner Projektphase Ende 2008 die JavaFX-Plattform. JavaFX ist ein ganz neuer GUI-Stack und komplett von Swing und AWT entkoppelt.

In der ersten Version gehörte die Programmiersprache *JavaFX Script* mit dazu, doch ab Version JavaFX 2 hat Oracle die Richtung geändert: JavaFX ist nun eine pure Java-Bibliothek, und die eigenwillige Programmiersprache JavaFX Script ist Vergangenheit. Das Open-Source-Projekt *Visage* (<http://code.google.com/p/visage/>) ist ein Fork von Oracles JavaFX-Script-Code unter der GPL-Lizenz und möchte die interessante Programmiersprache am Leben halten. Wie die Chancen stehen, ist jedoch im Moment noch völlig offen.

Konkurrenz für Flash, Silverlight und JavaFX kommt aus dem Bereich der Web-Standards: auch aufwändige Webanwendungen sind heute schon mit JavaScript und HTML 5 realisierbar. Auch Microsoft fuhr das Engagement für Silverlight zurück und bevorzugt nun Lösungen auf der Basis von JavaScript + HTML 5, insbesondere für mobile Endgeräte, da den Redmontern klar ist, dass es nie Silverlight auf dem iPhone oder Androids geben wird.²⁵ Adobe selbst beginnt mit Konvertern von Flash nach HTML 5/JavaScript und zeigt damit auch die Zukunft auf. Es ist daher unwahrscheinlich, dass JavaFX einen ernsthaften Anteil im Web gewinnen kann, wohl aber auf einem anderen Gebiet, das weder JavaScript, Flash noch Silverlight abdecken: Blu-ray-Player und Settop-Boxen. Die *Blu-ray Disc Association* (BDA) hat mit *Blu-ray Disc Java* (kurz BD-J) eine Spezifikation verabschiedet, die beschreibt, wie Blu-ray-Player Menüs oder andere interaktive Anwendungen ausführen können (das Ganze basiert auf einer abgespeckten Java-Version, der Java ME). Im Moment wird diskutiert, ob zukünftig die Anforderungen an die Blu-ray-

²⁵ Mit <http://mono-android.net/> gibt es interessante Ansätze.

Player so hochgeschraubt werden, dass sie interaktive JavaFX-Anwendungen ausführen können. Das wäre sicherlich ein Bereich, in dem JavaFX punkten könnte, aber wohl kaum im Internet, obwohl es durchaus ein paar nennenswerte Beispiele für JavaFX gibt. Das berühmteste stammt von den Olympischen Winterspielen 2010 in Vancouver (<http://www.vancouver2010.com/>). Es bleibt aber bei diesem Beispiel.



Quelle: <http://www.vancouver2010.com/olympic-medals/geo-view/>

Abbildung 1.3: JavaFX-Anwendung

Ist JavaFX damit irrelevant? Nein, denn auch wenn JavaFX im Web wohl kaum eine Rolle spielen wird und wir keine JavaFX-Anwendungen auf mobilen Endgeräten sehen werden, ist doch zu vermuten, dass JavaFX Swing und AWT beerbt. Noch entwickelt sich JavaFX unabhängig und ist kein Teil der Java SE, aber das kann sich ändern – in Java 1.0 war Swing auch noch nicht enthalten. Im Moment ist JavaFX kaum sichtbar, und leistungsfähige kommerzielle oder freie Komponenten existieren praktisch nicht, aber der Weg ist vorgezeichnet, dass JavaFX die Zukunft für clientseitige Oberflächen sein wird. Bei Oracle jedenfalls wird die Energie eindeutig in JavaFX investiert, und Swing/AWT bekommen kaum Aufmerksamkeit. Das lässt sich gut an den Neuerungen der letzten 10 Jahre ablesen. Die Dinge, die in Java 7 integriert wurden, hätte sich Oracle auch spa-

ren können. Im Prinzip gibt es in Java 7 nur vier »größere« Änderungen, darunter eine, die den Farbauswahldialog – der selbst schon sehr selten im Einsatz ist – um eine HSL-Farbselektion²⁶ ergänzte. Toll.

1.2.17 Features, Enhancements (Erweiterungen) und ein JSR

Java wächst von Version zu Version, sodass es regelmäßig größere Zuwächse bei den Bibliotheken gibt sowie wohlproportionierte Änderungen an der Sprache und minimale Änderungen an der JVM. Änderungen an der Java SE-Plattform werden in Features und Enhancements kategorisiert. Ein *Enhancement* ist dabei eine kleine Änderung, die nicht der Rede wert ist – dass etwa eine kleine Funktion wie `isEmpty()` bei der Klasse `String` hinzukommt. Diese Erweiterungen bedürfen keiner großen Abstimmung und Planung und werden von Oracle-Mitarbeitern einfach integriert.

Features dagegen sind in Bezug auf den Aufwand der Implementierung schon etwas Größeres. Oder aber die Community erwartet diese Funktionalität dringend – das macht sie deutlich, indem sie einen Feature-Request auf Oracles Webseite platziert und viele für dieses Feature stimmen. Eine weitere Besonderheit ist, wie viele dieser Features geplant werden. Denn oftmals entsteht ein *JSR* (*Java Specification Request*), der eine bestimmte Reihenfolge bei der Planung vorschreibt. Die meisten Änderungen an den Bibliotheken beschreibt ein JSR, und es gibt mittlerweile Hunderte von JSRs.

In der Anfangszeit war die Implementierung gleichzeitig die Spezifikation, aber mittlerweile gibt es für einen Java-Compiler, die JVM und diverse Bibliotheken eine beschreibende Spezifikation. Das gilt auch für Java als Gesamtes. *Java SE* ist eine Spezifikation, die zum Beispiel das Oracle JDK, OpenJDK, *Apache Harmony*²⁷ oder *GNU Classpath*²⁸ realisieren. Während das Oracle JDK (kurz JDK) auf dem OpenJDK basiert und das OpenJDK unter der GPL-Lizenz steht, ist Apache Harmony ein Projekt unter der Apache-Lizenz. Das OpenJDK für Windows und Linux ist die Referenzimplementierung und definiert somit den Standard. (Das ist dann wichtig, wenn eben gewisse Eigenschaften doch nicht dokumentiert sind.) Java SE 7 wird im JSR 336 beschreiben, was ein sogenanntes *Umbrella-JSR* ist, weil es andere JSRs enthält.

²⁶ Hue-Saturation-Luminance

²⁷ <http://harmony.apache.org/>

²⁸ Zu Hause unter <http://www.gnu.org/s/classpath/>. Allerdings wurde diese Seite schon seit 2009 nicht mehr aktualisiert. Da das OpenJDK wie auch GNU Classpath unter der GPL-Lizenz steht, gibt es keine Notwendigkeit mehr für eine weitere Java SE-Implementierung unter der GPL.

1.2.18 Die Entwicklung von Java und seine Zukunftsaussichten

Ein Buch über Java sollte im Allgemeinen eine positive Grundstimmung haben und die Leser nicht mit apokalyptischen Szenarien verschrecken. Doch gibt es einige Indizien dafür, dass sich Java als Programmiersprache nicht großartig weiterentwickeln wird und eher in einen »Wartungsmodus« übergeht. Zu dem Erfolg der Sprache zählt, dass sie im Vergleich zum Vorgänger C++ deutlich einfacher zu erlernen war und von »gefährlichen« syntaktischen Konstrukten die Finger ließ (obwohl immer noch einige Punkte übrig blieben). So schrieb einer der Sprachväter, James Gosling – der nach der Übernahme von Sun durch Oracle das Unternehmen verlassen hat –, schon 1997:

»Java ist eine Arbeitssprache. Sie ist nicht das Produkt einer Doktorarbeit, sondern eine Sprache für einen Job. Java fühlt sich für viele Programmierer sehr vertraut an, denn ich tendiere stark dazu, Dinge zu bevorzugen, die schon oft verwendet wurden, statt Dingen, die eher wie eine gute Idee klangen.«²⁹

Diese Haltung besteht bis heute. Java soll bloß einfach bleiben, auch wenn die anderen Programmiersprachen um Java herum syntaktische Features haben, die jeden Compilerbauer ins Schwitzen bringen. Bedeutende Sprachänderungen gab es eigentlich nur in Java 5 (also etwa zehn Jahre nach der Einführung), und für Java 7 mussten die Sprachänderungen ausdrücklich klein bleiben. Daher finden sich in Java 7 auch keine großen Kramcher, sondern Features, die bei anderen Sprachen eher ein Achselzucken provozieren würden. Als Beispiel sei genannt, dass in Zahlen Unterstriche erlaubt sind, die die Lesbarkeit erhöhen. Statt 123456 kann es nun 12_34_56 heißen.

Bei der Dreifaltigkeit der Java-Plattform – Java als Programmiersprache, den Standardbibliotheken und der JVM als Laufzeitumgebung – lässt sich erkennen, dass es große Bewegung bei den unterschiedlichen Programmiersprachen auf der Java-Laufzeitumgebung gibt. Es zeichnet sich ab, dass Java-Entwickler weiterhin in Java entwickeln werden, aber eine zweite Programmiersprache auf der JVM zusätzlich nutzen. Das kann Groovy, Scala, Jython, JRuby oder eine andere JVM-Sprache sein. Dadurch, dass die alternativen Programmiersprachen auf der JVM aufsetzen, können sie alle Java-Bibliotheken nutzen und daher Java als Programmiersprache in einigen Bereichen ablösen. Dass die alternativen Sprachen auf die üblichen Standardbibliotheken zurückgreifen, funktioniert reibungslos, allerdings ist der umgekehrte Weg, dass etwa Scala-Bibliotheken aus Jython heraus genutzt werden, (noch) nicht standardisiert. Bei der .NET-Plattform klappt das besser, und hier ist es wirklich egal, ob C# oder VB.NET Klassen deklariert oder nutzt.

²⁹ Im Original: »Java is a blue collar language. It's not PhD thesis material but a language for a job. Java feels very familiar to many different programmers because I had a very strong tendency to prefer things that had been used a lot over things that just sounded like a good idea.« (<http://www.computer.org/portal/web/cSDL/doi/10.1109/2.587548>)

Als die Übernahme von Sun vor der Tür stand, hat Oracle sich sehr engagiert gegenüber den Sun-Technologien gezeigt. Nach der Übernahme wandelt sich das Bild nun etwas, und Oracle hat eher für negative Schlagzeilen gesorgt, etwa als es die Unterstützung für OpenSolaris eingestellt hat. Auch was die Informationspolitik und Unterstützung von Usergroups angeht, verhält sich Oracle ganz anders als Sun. Verlässliche Zeitaussagen zu Java 7 gab es lange Zeit nicht, und durch die Klage gegen Google wegen Urheberrechtsverletzungen in Android machte sich Oracle auch keine Freunde. Android gilt als Beweis, dass Java auf dem Client tatsächlich erfolgreich ist.

1.3 Java-Plattformen: Java SE, Java EE und Java ME

Die Java-Plattform besteht aus Projekten, die es erlauben, Java-Programme auszuführen. Im Moment werden vier Plattformen unterschieden: Java SE, Java ME, Java Card und Java EE.

1.3.1 Die Java SE-Plattform

Die *Java Platform Standard Edition* (Java SE) – früher J2SE genannt – ist eine Systemumgebung zur Entwicklung und Ausführung von Java-Programmen. Java SE enthält alles, was zur Entwicklung von Java-Programmen nötig ist. Obwohl die Begrifflichkeit etwas unscharf ist, lässt sich die Java SE als Spezifikation verstehen und nicht als Implementierung. Damit Java-Programme übersetzt und ausgeführt werden können, müssen aber ein konkreter Compiler, Interpreter und die Java-Bibliotheken auf unserem Rechner installiert sein. Oracle bietet auf der Webseite <http://www.oracle.com/technetwork/java/javase/downloads/index.html> ein JDK an, das die Standardimplementierung der Java SE für Windows, Solaris OS und Linux ist. Das freie und mittlerweile unter der GPL stehende OpenJDK ist die Referenzimplementierung, aber nicht die Version, die der Benutzer direkt von der Oracle-Webseite bekommt. Es spielt in diesem Buch nur eine untergeordnete Rolle, genauso wie die alternativen Implementierungen Apache Harmony und GNU Classpath. Auch gibt es verschiedene Laufzeitumgebungen, doch uns interessiert im Folgenden nur die JVM von Oracle.

Versionen der Java SE

Am 23. Mai 1995 stellte damals noch Sun Java erstmals der breiten Öffentlichkeit vor. Seitdem ist viel passiert, und in jeder Version erweiterte sich die Java-Bibliothek. In den Versionen Java 1.1 und Java 5 gab es größere Änderungen an der Programmiersprache

selbst, in Java 7 ein paar. Java 8 wird größere Änderungen bekommen, die stark die Art und Weise ändern werden, wie Software geschrieben wird. Dennoch gibt es von einer Version zur nächsten kaum Inkompatibilitäten, und fast zu 100 % kann das, was unter Java N übersetzt wurde, auch unter Java N+1 übersetzt werden – nur selten gibt es Abstriche in der Bytecode-Kompatibilität.³⁰

Version	Datum	Einige Neuerungen bzw. Besonderheiten
1.0, Urversion	1995	Die 1.0.x-Versionen lösen diverse Sicherheitsprobleme.
1.1	Februar 1997	Neuerungen bei der Ereignisbehandlung, beim Umgang mit Unicode-Dateien (Reader/Writer statt nur Streams), außerdem Datenbankunterstützung via JDBC sowie innere Klassen und eine standardisierte Unterstützung für Nicht-Java-Code (nativen Code)
1.2	November 1998	Es heißt nun nicht mehr JDK, sondern <i>Java 2 Software Development Kit (SDK)</i> . <i>Swing</i> ist die neue Bibliothek für grafische Oberflächen, und es gibt eine Collection-API für Datenstrukturen und Algorithmen.
1.3	Mai 2000	Namensdienste mit JNDI, verteilte Programmierung mit RMI/IOP, Sound-Unterstützung
1.4	Februar 2002	Schnittstelle für XML-Parser, Logging, neues IO-System (NIO), reguläre Ausdrücke, Assertions
5	September 2004	Das Java-SDK heißt wieder <i>JDK</i> . Neu sind generische Typen, typsichere Aufzählungen, erweitertes for, Autoboxing, Annotationen.
6	Ende 2006	Web-Services, Skript-Unterstützung, Compiler-API, Java-Objekte an XML-Dokumente binden, System Tray
7	Juli 2011	Kleine Sprachänderungen, NIO2, erste pure GPL-Version
8	Geplant für Mitte 2013	Weitere Sprachänderungen wie Closures, Plattform-Modularisierung

Tabelle 1.1: Neuerungen und Besonderheiten der verschiedenen Java-Versionen

³⁰ Die Seite <http://tutego.de/go/migratingtojava5> zeigt auf, wie Walmart der Umstieg auf Java 5 gelang – relativ problemlos: »[...] the overall feeling is that a migration to Java 1.5 in a production environment can be a mostly painless exercise.«

Die Produktzyklen waren in der Vergangenheit relativ konstant. Einen Bruch gibt es bei der Version von Java 6 auf Java 7, auf die Entwickler sehr lange warten mussten. Gründe für die Verzögerung waren a) der Aufkauf von Sun durch Oracle, b) die Änderung auf die Open-Source-Lizenz GPL und c) die Entwicklung von JavaFX, in die viel Energie investiert wurde.

Java 7 und ein Bug



Fünf Tage vor dem Release von Java 7 kam ein schwerwiegender Fehler in der JVM ans Tageslicht, der bei ganz speziellen Schleifenkonstruktionen zum Absturz führte. Oracle erschien im schlechten Licht, da das Release nicht verschoben wurde und Oracle den Fehler mehr oder weniger totschwieg. Das erste Update behob den Fehler.

Codenamen und Namensänderungen

Mit Java 5 entfiel das Präfix »1.« in der Versionskennung des Produkts. Es heißt also Java 6 statt Java 1.6 und Java 7 statt Java 1.7; in den Entwickler-Versionen bleibt die Schreibweise mit der »1.« aber weiterhin gültig.³¹ Auch das Anhängsel ».0« für die Hauptversionen und die Unterverisionen ist verschwunden – es bleiben ganze Zahlen mit Updates. Schlussendlich fiel auch die »2« aus den in Java 1.2 eingeführten Begriffen J2SE, J2ME und J2EE heraus; es heißt aktuell *Java SE*, *Java ME* und *Java EE*. Früher vergab Sun auch Codenamen, wie *Tiger* für Java 5, doch das ist Vergangenheit.³²

1.3.2 Java für die Kleinen

Die *Java Platform, Micro Edition* (Java ME) ist eine Umgebung für kompakte tragbare Computer, also PDAs, Organizer und Telefone. Java ist heutzutage auf fast jedem Handy zu finden, und Sun sprach 2007 von über 2 Milliarden Java-fähigen Telefonen.³³ Die Bedeutung der Java ME schwindet jedoch, da moderne Geräte mittlerweile so viel Power haben, dass sie eine normale JVM laufen lassen können. Die Referenzimplementierung der Java ME heißt *phoneMe*³⁴ und steht unter der GPL; sie implementiert weitere Standards, wie *Java Mobile Media API*, *Scalable 2D Vector Graphics API* und weitere.

³¹ Siehe dazu <http://download.oracle.com/javase/1.5.0/docs/relnotes/version-5.0.html> und http://java.sun.com/j2se/versioning_naming.html.

³² <http://java.sun.com/j2se/codenames.html>

³³ Quelle: http://de.sun.com/sunnews/events/2007/20071203/pdf/TD_FRA_GoslingKeynote.pdf

³⁴ <https://phoneme.dev.java.net/>

Seitenwind bekommt Java ME von *Android*, einem Projekt, das von Google initiiert wurde und nun in den Händen der *Open Handset Alliance* liegt. Android ist nicht nur eine Software-Plattform, sondern auch ein Betriebssystem. Statt einer JVM mit standarisierter Java-Bytecode nutzt Android einen völlig anderen Bytecode und führt ihn auf der *Dalvik Virtual Machine* aus. Die immer noch laufende Klage von Oracle gegen Google gibt einen Eindruck von der Wichtigkeit des mobilen Markts.

1.3.3 Java für die ganz, ganz Kleinen

Mit *Java Card* definiert Oracle einen Standard für Java-ähnliche Programme auf Chipkarten (Smartcards). Der Sprachstandard von Java ist allerdings etwas eingeschränkt. Die Ausgabe des Java-Compilers ist ein Bytecode, der dem Standard-Bytecode ähnlich ist. Dieser Bytecode wird dann auf der *Java Card Virtual Machine* ausgeführt, die auf der Smart-Card (etwa einer SIM-Karte) Platz findet. Da es jedoch ganz andere Speicheranforderungen für so ein winziges System gibt, ist die Laufzeitumgebung nicht mit der Standard-JVM vergleichbar. Es gibt keine Threads und keine Garbage-Collection. Auch bei den Bibliotheken gibt es Unterschiede. Nicht nur, dass viele bekannte Klassen fehlen, umgekehrt gehören starke kryptografische Algorithmen mit zum Paket, und natürlich gehört auch ein Paket dazu, mit dem die Kartenanwendung mit der Außenwelt kommunizieren kann. Seit dem *Java Card 3.0*-Standard gibt es eine *Classic Edition* und eine *Connected Edition*, wobei die Connected Edition viele Einschränkungen nicht mehr hat; so gibt es nun auch bei ihr Threads und Garbage-Collection.

Mit dem *Java Card*-Standard können viel einfacher Programme auf Karten unterschiedlicher Hersteller gebracht werden – sofern die Karte dem Standard entspricht. Vorher war das immer etwas schwierig, da jeder Kartenhersteller unterschiedliche APIs und Tools verwendete und die Karte in der Regel in einem C-Dialekt programmiert wurde.

1.3.4 Java für die Großen

Die *Java Platform, Enterprise Edition (Java EE)* ist ein Aufsatz für die Java SE und integriert Pakete, die zur Entwicklung von Geschäftsanwendungen (Enterprise-Applikationen genannt) nötig sind. Dazu zählen etwa die Komponententechnologie der *Enterprise JavaBeans (EJBs)*, *JSP/JSF/Servlets* für dynamische Webseiten, die *JavaMail-API* und weitere. Die Referenzimplementierung für Java EE 6 und Java EE 5 ist *GlassFish*. »Java 7 – Mehr als eine Insel« geht auf einige Bibliotheken aus der Java EE kurz ein, etwa auf JPA, JSP und Servlets.

Interessant ist zu beobachten, dass im Laufe der letzten Jahre Teile aus der Java EE in die Java SE gewandert sind, etwa JAX-WS (Web-Services), JNDI (Verzeichnisservice), JAXB (Objekt-XML-Mapping). Möglicherweise kommt in Zukunft die Technologie für objekt-relationales Mapping (JPA) aus der Java EE in die Standard Edition.

1.3.5 Echtzeit-Java (Real-time Java)

Zwar laufen bei der Java ME Programme auf Geräten mit reduziertem Speicher und eingeschränkter Prozessor-Leistungsfähigkeit, das sagt aber nichts über die Reaktionsfähigkeit der Laufzeitumgebung auf externe Ereignisse aus. Wenn ein Sensor in der Stoßstange einen Aufprall meldet, darf die Laufzeitumgebung keine 20 ms in einer Speicheraufräumaktion festhängen, bevor das Ereignis verarbeitet wird und der Airbag aufgeht. Um diese Lücke zu schließen, wurde schon früh – im Java 2001 – von der Java-Community der JSR 1, »Real-time Specification for Java« (kurz RTSJ), definiert. (Mittlerweile JSR 282 für den Nachfolger RTJS 1.1.)

Echtzeit-Anwendungen zeichnen sich dadurch aus, dass es eine maximale deterministische Wartezeit gibt, die das System zum Beispiel bei der Garbage-Collection blockiert, um etwa auf Änderungen von Sensoren zu reagieren – ein Echtzeitsystem kann eine Antwortzeit garantieren, etwas, was eine normale virtuelle Maschine nicht kann. Denn nicht nur die Zeit für die Garbage-Collection ist bei normalen Laufzeitumgebungen eher unbestimmt, auch andere Aktionen unbestimmter Dauer kommen dazu: Lädt Java eine Klasse, dann zur Laufzeit. Das kann zu beliebig vielen weiteren Abhängigkeiten und Laudezyklen führen. Bis also eine Methode ausgeführt werden kann, können Hunderte von Klassendateien nötig sein, und das Laden kann unbestimmt lange dauern.

Mit Echtzeitfähigkeiten lassen sich auch Industrieanlagen mit Java steuern und lässt sich Software aus dem Bereich Luft- und Raumfahrt, Medizin, Telekommunikation und Unterhaltungselektronik mit Java realisieren. Dieser Bereich blieb Java lange Zeit verschlossen und bildete eine Domäne von C++. Damit dies in Java möglich ist, müssen JVM und Betriebssystem zusammenpassen. Während eine herkömmliche JVM auf mehr oder weniger jedem beliebigen Betriebssystem läuft, sind die Anforderungen für Echtzeit-Java strenger. Das Fundament bildet immer ein Betriebssystem mit Echtzeitfähigkeiten (*Real-Time Operating System (RTOS)*), etwa *Solaris 10*, *Realtime Linux*, *QNX*, *OS-9* oder *VxWorks*. Darauf setzt eine Echtzeit-JVM auf, eine Implementierung der Realtime-Spezifikation, wie die *JamaicaVM* (<http://aicas.com/jamaica.html>) oder *Aonix Perc Pico* von Atego (<http://www.atego.com/capabilities/real-time>). Real-time Java (RT-Java) unterscheidet sich so auch in Details, etwa dass Speicherbereiche direkt belegt und freigegeben werden können (*scoped memory*), dass mehr Thread-Prioritäten zur Verfügung

stehen oder dass das Scheduling deutlich mehr in der Hand der Entwickler liegt. Die Entwicklung ist anders, findet aber unter den bekannten Werkzeugen wie IDEs, Test-tools und Bibliotheken statt.

1.4 Die Installation der Java Platform Standard Edition (Java SE)

Die folgende Anleitung beschreibt, woher wir Oracles Java SE-Implementierung beziehen können und wie die Installation verläuft.

1.4.1 Die Java SE von Oracle

Es gibt unterschiedliche Möglichkeiten, in den Besitz der Java SE zu kommen. Wer einen schnellen Zugang zum Internet hat, kann es sich von den Oracle-Seiten herunterladen. Nicht-Internet-Nutzer oder Anwender ohne schnelle Verbindungen finden Entwicklungsversionen häufig auch auf DVDs, wie etwa der DVD in diesem Buch.

JDK und JRE

In der Java SE-Familie gibt es verschiedene Ausprägungen: das JDK und das JRE. Da diejenigen, die Java-Programme nur laufen lassen möchten, nicht unbedingt alle Entwicklungstools benötigen, hat Oracle zwei Pakete geschnürt:

- Das *Java SE Runtime Environment (JRE)* enthält genau das, was zur Ausführung von Java-Programmen nötig ist. Die Distribution umfasst nur die JVM und Java-Bibliotheken, aber weder den Quellcode der Java-Bibliotheken noch Tools.
- Mit dem *Java Development Kit (JDK)* lassen sich Java SE-Applikationen entwickeln. Dem JDK sind Hilfsprogramme beigelegt, die für die Java-Entwicklung nötig sind. Dazu zählen der essenzielle Compiler, aber auch andere Hilfsprogramme, etwa zur Signierung von Java-Archiven oder zum Start einer Management-Konsole. In den Versionen Java 1.2, 1.3 und 1.4 heißt das JDK *Java 2 Software Development Kit (J2SDK)*, kurz *SDK*, ab Java 5 heißt es wieder *JDK*.

Das JRE und JDK von Oracle sind beide gratis erhältlich. Das Projekt ist selbst extrem komplex und umfasst (Stand 2009) etwa 900.000 Zeilen C++-Code.³⁵

³⁵ http://llvm.org/devmtg/2009-10/Cifuentes_ParfaitBugChecker.pdf

1.4.2 Download des JDK

Oracle bietet auf der Webseite <http://www.oracle.com/technetwork/java/javase/downloads/> das Java SE und andere Dinge zum Download an. Oracle bündelt die Implementierung in unterschiedliche Pakete:

- JDK
- JRE
- JDK mit Java EE
- JDK mit der NetBeans IDE

The screenshot shows the Oracle Java SE Development Kit 7 Downloads page. At the top, there's a navigation bar with links for Products and Services, Downloads, Store, Support, Training, Partners, About, and Oracle Technology Network. Below the navigation bar, the URL is shown as Oracle Technology Network > Java > Java SE > Downloads.

Java SE Development Kit 7 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java™ platform.

Java SE Development Kit 7

You must accept the [Oracle Binary Code License Agreement for Java SE](#) to download this software.

Accept License Agreement Decline License Agreement

Product / File Description	File Size	Download
Linux x86 - RPM Installer	77.28 MB	jdk-7-linux-i586.rpm
Linux x86 - Compressed Binary	92.17 MB	jdk-7-linux-i586.tar.gz
Linux x64 - RPM Installer	77.91 MB	jdk-7-linux-x64.rpm
Linux x64 - Compressed Binary	90.57 MB	jdk-7-linux-x64.tar.gz
Solaris x86 - Compressed Packages	154.74 MB	jdk-7-solaris-i586.tar.Z
Solaris x86 - Compressed Binary	94.75 MB	jdk-7-solaris-i586.tar.gz
Solaris SPARC - Compressed Packages	157.81 MB	jdk-7-solaris-sparc.tar.Z
Solaris SPARC - Compressed Binary	99.48 MB	jdk-7-solaris-sparc.tar.gz
Solaris SPARC 64-bit - Compressed Packages	16.28 MB	jdk-7-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit - Compressed Binary	12.38 MB	jdk-7-solaris-sparcv9.tar.gz
Solaris x64 - Compressed Packages	14.66 MB	jdk-7-solaris-x64.tar.Z
Solaris x64 - Compressed Binary	9.39 MB	jdk-7-solaris-x64.tar.gz
Windows x86	79.48 MB	jdk-7-windows-i586.exe
Windows x64	80.25 MB	jdk-7-windows-x64.exe

Java SDKs and Tools

- [Java SE](#)
- [Java EE and Glassfish](#)
- [Java ME](#)
- [JavaFX](#)
- [Java Card](#)
- [NetBeans IDE](#)

Java Resources

- [New to Java?](#)
- [APIs](#)
- [Code Samples & Apps](#)
- [Developer Training](#)
- [Documentation](#)
- [Java BluePrints](#)
- [Java.com](#)
- [Java.net](#)
- [Student Developers](#)
- [Tutorials](#)

Java magazine NEW! Get it now for FREE! Subscribe Today

Hardware and Software
Engineered to Work Together

About Oracle | Oracle and Sun | | Subscribe | Careers | Contact Us | Site Maps | Legal Notices | Terms of Use | Your Privacy Rights

Abbildung 1.4: Die Java SE-Download-Seite von Oracle

Wir entscheiden uns für das pure JDK, denn es enthält einige Entwicklungstools, die wir später benötigen (auch wenn für Eclipse und einfache Programme prinzipiell das JRE ausreicht)³⁶.

Ein Klick auf JDK DOWNLOAD führt zur Seite <http://www.oracle.com/technetwork/java/javase/downloads/java-se-jdk-7-download-432154.html>, die direkt zu Installationspaketen für folgende Systeme führt:

- Microsoft Windows für jeweils 32- und 64-Bit-Systeme
- Solaris SPARC 32 und 64 Bit, Solaris x86, Solaris x64
- Linux x86 und Linux x64

Für Windows x86 ist es der Link <http://download.oracle.com/otn-pub/java/jdk/7/jdk-7-windows-i586.exe>. Ein Download ist jedoch erst dann möglich, wenn die Lizenzbestimmungen (*Oracle Binary Code License Agreement for Java SE*) bestätigt wurden.

Download der Dokumentation

Die API-Dokumentation der Standardbibliothek und Tools ist kein Teil des JDK (bei einer Größe des JDK von fast 76 MiB ist eine Trennung sinnvoll, denn die Dokumentation selbst umfasst etwa die gleiche Größe). Die Hilfe kann online unter <http://www.tutego.de/go/javaapi/> eingesehen oder als Zip-Datei extra bezogen und lokal ausgepackt werden. Das komprimierte Archiv ist auf der DVD oder unter <http://www.oracle.com/technetwork/java/index-jsp-142903.html#documentation> erhältlich. Ausgepackt ist die API-Dokumentation eine Sammlung von HTML-Dateien. Unter <http://www.allmant.org/javadoc/index.php> findet sich die Hilfe auch im HTMLHelp- und WinHelp-Format. Diese Microsoft-Formate erleichtern die Suche in der Dokumentation.

1.4.3 Java SE unter Windows installieren

Die ausführbare Datei *jdk-7-windows-i586.exe* ist das Installationsprogramm. Es installiert die ausführbaren Programme wie Compiler und Interpreter sowie die Bibliotheken, Quellcodes und auch Beispielprogramme. Voraussetzung für die Installation sind genügend Rechte, ein paar MiB Plattspeicher und die Windows-Service-Packs. Quellcodes und Demos müssen nicht unbedingt installiert sein. Die Datei *jdk-7-windows-i586.exe*

³⁶ Eclipse bringt einen eigenen Java-Compiler mit, daher ist der Compiler des JDK nicht nötig. Doch das JDK bringt auch die Quellen der Java-Bibliotheken mit, was sehr nützlich ist, denn aus den Quellen wird direkt die API-Dokumentation generiert. Sie muss daher nicht extra bezogen werden.

ist die einzige, die für Java zwingend installiert werden muss; alles andere sind Extras, wie grafische Entwicklungsumgebungen.

Schritt-für-Schritt-Installation

Gehen wir nun Schritt für Schritt durch die Installation.



Abbildung 1.5: Der Startbildschirm

Nach dem Klick auf NEXT fragt der Installer nach den zu installierenden Komponenten. Zuerst wird das JDK, dann das JRE installiert.



Abbildung 1.6: Auswahl der Komponenten

Ein Klick auf CHANGE, und wir können das Installationsverzeichnis des JDK ändern. Mit NEXT beginnt die Installation des JDK.

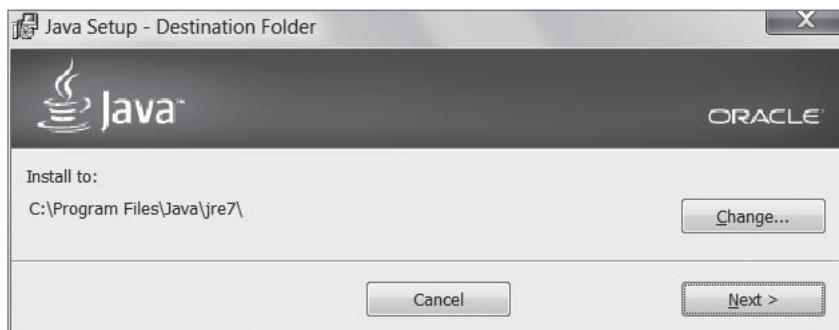


Abbildung 1.7: Installationsverzeichnis für das JRE bestimmen.

Nach der Installation des JDK folgt das JRE. Auch hier lässt sich das Verzeichnis ändern. Wir aktivieren NEXT. Anschließend folgt ein Dialog, mit dem wir uns für Updates registrieren lassen können. FINISH beendet die Installation.



Abbildung 1.8: Die Installation wurde erfolgreich beendet.

Nach der abgeschlossenen Installation können wir unter Windows – ohne Änderung des Installationsverzeichnisses – im Dateibaum unter *C:\Program Files\Java* bzw. *C:\Programme\Java* (oder *C:\Program Files (x86)\Java* bzw. *C:\Program Files (x64)\Java* je nach Windows-Version) die beiden Ordner *jdk1.7.0* und *jre* ausmachen (für die folgenden Beispiele geben wir immer den Pfad *C:\Program Files\Java\jdk1.7.0* an).

Registry-Einträge unter Windows *

Unter Windows wird in der Registry ein Zweig `HKEY_CURRENT_USER\Software\JavaSoft` angelegt, in dem unter anderem Preferences-Einstellungen gespeichert werden. Weiterhin gibt es einen Zweig `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\javaws.exe` mit dem Wert `C:\Program Files\Java\jre7\bin\javaws.exe` und dem zusätzlichen Schlüssel Path, der mit `C:\Program Files\Java\jre7\bin` belegt wird.

Ungünstig ist, dass der Installer nicht überprüft, ob eine alte Version schon vorhanden ist – er überschreibt die Registry-Einträge einfach. Die Lösungen sind nicht wirklich befriedigend: Entweder wird die Registry später restauriert oder eine alte Java-Version wird noch einmal darüber installiert.

Java im *C:\Windows\System32*-Pfad *

Der Installer setzt unter Windows die ausführbare Datei *java.exe* in das System-Verzeichnis *C:\Windows\System32*. Das Programm liest die Registry aus (wie gerade beschrieben) und startet die dort eingetragene JVM. Wer also den Standard ändern möchte, der muss die Registry-Schlüssel umbiegen.

Der Grund für die Installation an dieser zentralen Systemstelle ist, dass das Verzeichnis Teil des Windows-Standard-Suchpfades ist und dass wir nach der Installation von der Kommandozeile aus über *java* sofort Programme starten können. Der Installer könnte grundsätzlich auch die PATH-Variable erweitern, doch Oracle wählte diesen Weg nicht. Im System-Ordner ist auch nur *java* eingetragen, nicht aber *javac*, weshalb Entwickler doch oft die PATH-Variable auf das *bin*-Verzeichnis des JDK setzen.

Programme im *bin*-Verzeichnis

Im *bin*-Verzeichnis des JDK (*C:\Programme\Java\jdk1.7.0\bin*) sind zusätzliche Entwicklungswerzeuge und Java-Quellen untergebracht, die das JRE nicht enthält. Leider wird aber auch vieles doppelt installiert;³⁷ Oracle betrachtet das JRE nicht als Teilmenge des JDK.

Der JDK-Ordner hat nicht viele Verzeichnisse und Dateien. Die wichtigsten sind:

- *bin*: Hier befinden sich unter anderem der Compiler *javac* und der Interpreter *java*.
- *jre*: die eingebettete Laufzeitumgebung
- *demo* (falls installiert): Diverse Unterverzeichnisse enthalten Beispiel-Programme. Ein interessantes Demo finden wir unter *jfc/Java2D* (ein Doppelklick auf die *.jar*-Datei startet es) und online unter <http://download.java.net/javadesktop/swingset3/SwingSet3.jnlp>.
- *src.zip* (falls installiert): Das Archiv enthält den Quellcode der öffentlichen Bibliotheken. Entwicklungsumgebungen wie Eclipse und NetBeans binden die Quellen automatisch mit ein, sodass sie leicht über einen Tastendruck zugänglich sind.

1.5 Das erste Programm compilieren und testen

Nachdem wir die grundlegenden Konzepte von Java besprochen haben, wollen wir ganz dem Zitat von Dennis M. Ritchie folgen, der sagt: »Eine neue Programmiersprache lernt man nur, wenn man in ihr Programme schreibt.« In diesem Abschnitt nutzen wir den Java-Compiler und Interpreter von der Kommandozeile. Wer gleich eine ordentliche Entwicklungsumgebung wünscht, der kann problemlos diesen Teil überspringen und bei den IDEs fortfahren.

1.5.1 Ein Quadratzahlen-Programm

Das erste Programm zeigt einen Algorithmus, der die Quadrate der Zahlen von 1 bis 4 ausgibt. Die ganze Programmlogik sitzt in einer Klasse *Quadrat*, die drei Methoden enthält. Alle Methoden in einer objektorientierten Programmiersprache wie Java müssen in Klassen platziert werden. Die erste Methode, *quadrat()*, bekommt als Übergabeparameter eine ganze Zahl und berechnet daraus die Quadratzahl, die sie anschließend zurückgibt. Eine weitere Methode übernimmt die Ausgabe der Quadratzahlen bis zu einer vorgegebenen Grenze. Die Methode bedient sich dabei der Methode *quadrat()*. Zum Schluss muss es noch ein besonderes Unterprogramm *main()* geben, das für den Java-Interpreter den Einstiegspunkt bietet. Die Methode *main()* ruft dann die Methode *ausgabe()* auf.

³⁷ Das JRE gibt es so zum Beispiel unter *C:\Program Files\Java\jdk1.7.0\jre* und auch unter *C:\Program Files\Java\jre7*. Daraus folgt aber auch, dass zusätzliche Bibliotheken auch an zwei Stellen installiert werden müssen.

Listing 1.1: Quadrat.java

```
/**  
 * @version 1.01    6 Dez 1998  
 * @author Christian Ullenboom  
 */  
  
public class Quadrat  
{  
    static int quadrat( int n )  
    {  
        return n * n;  
    }  
    static void ausgabe( int n )  
    {  
        String s;  
        int i;  
        for ( i = 1; i <= n; i=i+1 )  
        {  
            s = "Quadrat(" + i + ")" + " = " + quadrat(i);  
            System.out.println( s );  
        }  
    }  
    public static void main( String[] args )  
    {  
        ausgabe( 4 );  
    }  
}
```

Hinweis

Der Java-Compiler unterscheidet sehr penibel zwischen Groß- und Kleinschreibung.

Der Quellcode (engl. *source code*) für *Quadrat.java* soll exemplarisch im Verzeichnis *C:\projekte* gespeichert werden. Dazu kann ein einfacher Editor wie *Notepad* (START •

PROGRAMME • ZUBEHÖR • EDITOR) unter Windows verwendet werden. Beim Abspeichern mit Notepad unter DATEI • SPEICHERN UNTER... muss bei DATEINAME *Quadrat.java* stehen und beim Dateityp ALLE DATEIEN ausgewählt sein, damit der Editor nicht automatisch die Dateiendung .txt vergibt.

1.5.2 Der Compilerlauf

Der Quellcode eines Java-Programms lässt sich so allein nicht ausführen. Ein spezielles Programm, der *Compiler* (auch *Übersetzer* genannt), transformiert das geschriebene Programm in eine andere Repräsentation. Im Fall von Java erzeugt der Compiler die DNA jedes Programms, den Bytecode.

Wir wechseln zur Eingabeaufforderung (Konsole) und in das Verzeichnis mit dem Quellcode. Damit sich Programme übersetzen und ausführen lassen, müssen wir die Programme *javac* und *java* aus dem *bin*-Verzeichnis der JDK-Installation aufrufen.

Liegt die Quellcodedatei vor, übersetzt der Compiler sie in Bytecode.

```
C:\projekte>javac Quadrat.java
```

Alle Java-Klassen übersetzt *javac *.java*. Wenn die Dienstprogramme *javac* und *java* nicht im Suchpfad stehen, müssen wir einen kompletten Pfadnamen angeben – wie *C:\Program Files\Java\jdk1.7.0\bin\javac *.java*.

Die zu übersetzende Datei muss – ohne Dateiendung – so heißen wie die in ihr definierte öffentliche Klasse. Die Beachtung der Groß- und Kleinschreibung ist wichtig. Eine andere Endung, wie etwa *.txt* oder *.jav*, ist nicht erlaubt und führt zu einer Fehlermeldung:

```
C:\projekte>javac Quadrat.txt
Quadrat.txt is an invalid option or argument.
Usage: javac <options> <source files>
```

Der Compiler legt – vorausgesetzt, das Programm war fehlerfrei – die Datei *Quadrat.class* an. Diese enthält den Bytecode.

Findet der Compiler in einer Zeile einen Fehler, so meldet er diesen unter der Angabe der Datei und der Zeilennummer. Nehmen wir noch einmal unser Quadratzahlen-Programm, und bauen wir in der *quadrat()*-Methode einen Fehler ein (das Semikolon fällt der Löschtaste zum Opfer). Der Compilerdurchlauf meldet:

```
Quadrat.java:10: ';' expected.  
    return n * n  
           ^  
1 error
```

Den Suchpfad für Windows setzen

Da es unpraktisch ist, bei jedem Aufruf immer den kompletten Pfad zur JDK-Installation anzugeben, lässt sich der Suchpfad erweitern, in dem die Shell nach ausführbaren Programmen sucht. Um die Pfade dauerhaft zu setzen, müssen wir die Umgebungsvariable PATH modifizieren. Für eine Sitzung reicht es, den *bin*-Pfad des JDK hinzuzunehmen. Wir setzen dazu in der Kommandozeile von Windows den Pfad *jdk1.7.0\bin* an den Anfang der Suchliste, damit im Fall von Altinstallationen immer das neue JDK verwendet wird:

```
set PATH=C:\Program Files\Java\jdk1.7.0\bin;%PATH%
```

Die Anweisung modifiziert die Pfad-Variable und legt einen zusätzlichen Verweis auf das *bin*-Verzeichnis von Java an.

Damit die Pfadangabe auch nach einem Neustart des Rechners noch verfügbar ist, müssen wir abhängig vom System unterschiedliche Einstellungen vornehmen.

- Ab Windows Vista aktiviere im Startmenü **SYSTEMSTEUERUNG**, und klicke dann auf **SYSTEM UND SICHERHEIT**. Aktiviere **SYSTEM** und dann **ERWEITERTE SYSTEMEIGENSCHAFTEN**. Es öffnet sich ein neuer Dialog mit einer Schaltfläche **UMGEBUNGSVARIABLEN...** Im unteren Teil, **SYSTEMVARIABLEN**, scrolle nach **PATH**. Mit **BEARBEITEN...** verändere den Eintrag, und füge dem Pfad hinter einem Semikolon das JDK-*bin*-Verzeichnis hinzu. Bestätige die Änderung mit **OK, OK, OK**.
- Unter Windows XP aktivieren wir den Dialog **SYSTEMEIGENSCHAFTEN** unter **START • EINSTELLUNGEN • SYSTEMSTEUERUNG • SYSTEM**. Unter dem Reiter **ERWEITERT** wählen wir die Schaltfläche **UMGEBUNGSVARIABLEN**, wo wir anschließend bei **SYSTEMVARIABLEN** die Variable **PATH** auswählen und mit **BEARBEITEN** verändern – natürlich können statt der Systemvariablen auch die lokalen Benutzereinstellungen modifiziert werden; da gibt es **PATH** noch einmal. Hinter einem Semikolon tragen wir den Pfad zum *bin*-Verzeichnis ein. Dann können wir den Dialog mit **OK, OK, OK** verlassen. War eine Eingabeaufforderung offen, bekommt sie von der Änderung nichts mit; ein neues Eingabeaufforderungsfenster muss geöffnet werden.

Weitere Hilfen gibt die Datei <http://tutego.de/go/installwindows>.

1.5.3 Die Laufzeitumgebung

Der vom Compiler erzeugte Bytecode ist kein üblicher Maschinencode für einen speziellen Prozessor, da Java als plattformunabhängige Programmiersprache entworfen wurde, die sich also nicht an einen physikalischen Prozessor klammert – Prozessoren wie Intel-, AMD- oder PowerPC-CPUs können mit diesem Bytecode nichts anfangen. Hier hilft eine Laufzeitumgebung weiter. Diese liest die Bytecode-Datei Anweisung für Anweisung aus und führt sie auf dem konkreten Mikroprozessor aus.

Der Interpreter *java* bringt das Programm zur Ausführung:

```
C:\projekte>java Quadrat  
Quadrat(1) = 1  
Quadrat(2) = 4  
Quadrat(3) = 9  
Quadrat(4) = 16
```

Als Argument bekommt die Laufzeitumgebung *java* den Namen der Klasse, die eine *main()*-Methode enthält und somit als ausführbar gilt. Die Angabe ist nicht mit der Endung *.class* zu versehen, da hier kein Dateiname, sondern ein Klassenname gefordert ist.

1.5.4 Häufige Compiler- und Interpreterprobleme

Arbeiten wir auf der Kommandozeilenebene (Shell) ohne integrierte Entwicklungsumgebung, können verschiedene Probleme auftreten. Ist der Pfad zum Compiler nicht richtig gesetzt, gibt der Kommandozeileninterpreter eine Fehlermeldung der Form

```
$ javac Quadrat.java
```

Der Befehl ist entweder falsch geschrieben oder konnte nicht gefunden werden.

Bitte überprüfen Sie die Schreibweise und die Umgebungsvariable 'PATH'.

aus. Unter Unix lautet die Meldung gewohnt kurz:

```
javac: Command not found
```

War der Compilerdurchlauf erfolgreich, können wir den Interpreter mit dem Programm *java* aufrufen. Verschreiben wir uns bei dem Namen der Klasse oder fügen wir unserem Klassennamen das Suffix *.class* hinzu, so meckert der Interpreter. Beim Versuch, die nicht existente Klasse Q zum Leben zu erwecken, schreibt der Interpreter auf den Fehlerkanal:

```
$ java Q.class  
Exception in thread "main"  java.lang.NoClassDefFoundError:Q/class
```

Ist der Name der Klassendatei korrekt, hat aber die Hauptmethode keine Signatur `public static void main(String[])`, so kann der Java-Interpreter keine Methode finden, bei der er mit der Ausführung beginnen soll. Verschreiben wir uns bei der `main()`-Methode in Quadrat, folgt die Fehlermeldung:

In class Quadrat: void main(String argv[]) is not defined

Hinweis

Der Java-Compiler und die Java-Laufzeitumgebung haben einen Schalter `-help`, der weitere Informationen zeigt. Amüsanterweise steht dort noch »Weitere Einzelheiten finden Sie unter <http://java.sun.com/javase/reference>«, wobei Oracle sonst gründlich vorging, um die Marke »Sun« aus den Quellen zu entfernen. Die URL wurde erst nach der ersten Version von Java 7 auf <http://www.oracle.com/technetwork/java/javase/documentation/index.html> korrigiert.

1.6 Entwicklungsumgebungen im Allgemeinen

Als Laufzeitumgebung ist das JRE geeignet, und mit dem JDK können auf der Kommandozeile Java-Programme übersetzt und ausgeführt werden – angenehm ist das allerdings nicht. Daher haben unterschiedliche Hersteller in den letzten Jahren einigen Aufwand betrieben, um die Java-Entwicklung zu vereinfachen. Moderne Entwicklungsumgebungen bieten gegenüber einfachen Texteditoren den Vorteil, dass sie besonders Spracheinsteigern helfen, sich mit der Syntax anzufreunden. Eclipse beispielsweise unterkriegt ähnlich wie moderne Textverarbeitungssysteme fehlerhafte Stellen. Zusätzlich bieten die IDEs die notwendigen Hilfen beim Entwickeln, wie etwa farbige Hervorhebung, automatische Codevollständigung und Zugriff auf Versionsverwaltungen oder auch Wizards, die mit ein paar Eintragungen Quellcode etwa für grafische Oberflächen oder Web-Service-Zugriffe generieren.

1.6.1 Die Entwicklungsumgebung Eclipse

Seit Ende 2001 arbeitet IBM an der Java-basierten Open-Source-Software *Eclipse* (<http://www.eclipse.org/>) und löste damit die alte *WebSphere*-Reihe und die Umgebung *Visual Age for Java* ab. 2003/2004 gründete IBM mit der *Eclipse Foundation* ein Konsortium,

das die Weiterentwicklung bestimmt. Diesem Konsortium gehören unter anderem die Mitglieder BEA, Borland, Computer Associates, Intel, HP, SAP und Sybase an. Eclipse steht heute unter der Common Public License und ist als quelloffene Software für jeden kostenlos zugänglich.

Eclipse macht es möglich, Tools als sogenannte Plugins zu integrieren. Viele Anbieter haben ihre Produkte schon für Eclipse angepasst, und die Entwicklung läuft weltweit in einem raschen Tempo.

Da Oracles IDE *NetBeans* ebenfalls frei ist und um andere Fremdkomponenten bereichert werden kann, zog sich IBM damals den Groll von Sun zu. Sun warf IBM vor, die Entwicklergemeinde zu spalten und noch eine unnötige Entwicklungsumgebung auf den Markt zu werfen, wo doch NetBeans schon so toll sei. Nun ja, die Entwickler haben entschieden: Statistiken sehen Eclipse deutlich vorne, wobei in den letzten Jahren NetBeans etwas Boden gutmachen konnte.

Eclipse ist ein Java-Produkt mit einer nativen grafischen Oberfläche, das flüssig seine Arbeit verrichtet – genügend Speicher vorausgesetzt (> 512 MiB). Die Arbeitszeiten sind auch deswegen so schnell, weil Eclipse mit einem sogenannten *inkrementellen Compiler* arbeitet. Speichert der Anwender eine Java-Quellcodedatei, übersetzt der Compiler diese Datei automatisch. Dieses Feature nennt sich *autobuild*.

1.6.2 NetBeans von Oracle

In den Anfängen der Java-Bewegung brachte Sun mit der Software *Java-Workshop* eine eigene Entwicklungsumgebung auf den Markt. Die Produktivitätsmöglichkeiten waren jedoch gering. Das änderte sich mit zwei strategischen Einkäufen, um wieder eine bedeutendere Rolle bei den Java-Entwicklungsumgebungen zu spielen:

- Im August 1999 übernahm Sun das kalifornische Softwarehaus Forte. Sun interessierte sich besonders für *SynerJ*, eine Produktsuite für Java SE- und Java EE-Entwicklungen.
- Etwa zwei Monate später erwarb Sun vom tschechischen Unternehmen NetBeans die gleichnamige Entwicklungsumgebung *NetBeans*. Nach kurzer Umbenennung in *Forté for Java* wurde es im Jahr 2000 als quelloffene Lösung wieder zu *NetBeans*.

NetBeans (<http://www.netbeans.org/>) bietet komfortable Möglichkeiten zur Java SE-, Java ME- und Java EE-Entwicklung mit Editoren und Wizards für die Erstellung grafischer Oberflächen von Swing- und Webanwendungen. Oracle ist sehr experimentierfreudig und unterstützt eine Reihe von Bibliotheken und Frameworks, deren Entwicklung noch nicht abgeschlossen ist.

Seit dem Wechsel von Sun zu Oracle befürchtet die Community, dass Oracle der IDE NetBeans in Zukunft keine hohe Priorität mehr einräumt. Zum einen hat das Unternehmen mit dem *Oracle JDeveloper* schon eine IDE im Programm, und zu anderen unterstützt Oracle auch sehr aktiv Eclipse und bietet spezielle Java EE-Plugins. Doch bisher ist keine Abkehr von NetBeans zu spüren, und von offizieller Stelle gibt es eher Unterstützung. Die Angst ist wohl unbegründet, und Oracle möchte mit den drei Entwicklungsumgebungen verschiedene Zielgruppen ansprechen.

1.6.3 IntelliJ IDEA

Dass Unternehmen mit einer Java-IDE noch Geld verdienen können, zeigt JetBrains, ein aus Tschechien stammendes Softwarehaus. Die Entwicklungsumgebung *IntelliJ IDEA* gibt es in einer freien, quelloffenen Grundversion *Community Edition*, die alles abdeckt, was zur Java SE-Entwicklung nötig ist, und in einer kommerziellen *Ultimate Edition* für 249 US-Dollar (Einzelentwickler) bzw. 599 US-Dollar (Unternehmen), die sich an die Java EE-Entwickler richtet. Die Basisversion enthält auch schon einen GUI-Builder, Unterstützung für Test-Frameworks und Versionsverwaltungssysteme und ist etwa mit *Eclipse IDE for Java Developers* vergleichbar. Die freie Community-Version ist beliebt, da die Unterstützung der alternativen JVM-Sprache *Groovy* sehr gut ist und ein tolles Scala-Plugin existiert.

1.6.4 Ein Wort zu Microsoft, Java und zu J++, J#

In der Anfangszeit verursachte Microsoft einigen Wirbel um Java. Mit Visual J++ (gesprochen »Jay Plus Plus«) bot Microsoft schon früh einen eigenen Java-Compiler (als Teil des *Microsoft Development Kit*) und mit der *Microsoft Java Virtual Machine* (MS-JVM) eine eigene schnelle Laufzeitumgebung. Das Problem war nur, dass Dinge wie RMI und JNI absichtlich fehlten³⁸ – JNI wurde 1998 nachgereicht. Entgegen allen Standards führte der J++-Compiler neue Schlüsselwörter wie `multicast` und `delegate` ein. Weiterhin fügte Microsoft einige neue Methoden und Eigenschaften hinzu, zum Beispiel *J/Direct*, um der plattformunabhängigen Programmiersprache den Windows-Stempel zu verpassen. Mit *J/Direct* konnten Programmierer aus Java heraus direkt auf Funktionen der Win32-API zugreifen und damit reine Windows-Programme in Java programmieren. Durch Integration von *DirectX* sollte die Internet-Programmiersprache Java multimediafähig gemacht werden. Das führte natürlich zu dem Problem, dass Applikationen, die

38 <http://www.microsoft.com/presspass/legal/charles.mspx>

mit J++ erstellt wurden, nicht zwangsläufig auf anderen Plattformen liefen. Sun klagte gegen Microsoft.

Da es Sun in der Vergangenheit finanziell nicht besonders gut ging, pumpte Microsoft im April 2004 satte 1,6 Milliarden US-Dollar in die Firma. Microsoft erkaufte sich damit das Ende der Kartellprobleme und Patentstreitigkeiten. Dass es bis zu dieser Einigung nicht einfach gewesen war, zeigen Aussagen von Microsoft-Projektleiter Ben Slivka über das JDK beziehungsweise die *Java Foundation Classes*, man müsse sie »bei jeder sich bietenden Gelegenheit anpissen« (»pissing on at every opportunity«).³⁹

Im Januar 2004 beendete Microsoft die Arbeit an J++, denn die Energie floss in das .NET-Framework und die .NET-Sprachen. Am Anfang gab es mit J# eine Java-Version, die Java-Programme auf der Microsoft .NET-Laufzeitumgebung CLR ausführt, doch Anfang 2007 wurde auch J# eingestellt. Das freie *IKVM.NET* (<http://www.ikvm.net/>) ist eine JVM für .NET und verfügt über einen Übersetzer von Java-Bytecode nach .NET-Bytecode, was es möglich macht, Java-Programme unter .NET zu nutzen. Das ist praktisch, denn für Java gibt es eine riesige Anzahl von Programmen, die somit auch für .NET-Entwickler zugänglich sind.

Microsoft hat sich aus der Java-Entwicklung nahezu vollständig zurückgezogen. Es gibt zum Beispiel noch den *Microsoft JDBC Driver for SQL Server*, und Microsoft unterstützt eine API für Office-Dokumente. Das Verhältnis ist heute auch deutlich entspannter, und vielleicht gratuliert Microsoft irgendwann einmal Oracle, wie es auch Linux zum 20. Geburtstag gratuliert hat.⁴⁰

1.7 Eclipse im Speziellen

Die Entwicklungsumgebung Eclipse ist selbst in Java programmiert und benötigt zur Ausführung mindestens eine Java 5 JRE.⁴¹ Da allerdings Teile, wie die grafische Oberfläche, in C implementiert sind, ist Eclipse nicht 100 % pures Java, und beim Download unter <http://www.eclipse.org/downloads/> ist auf das passende System zu achten.

³⁹ Würden wir nicht gerade im westlichen Kulturkreis leben, wäre diese Geste auch nicht zwangsläufig unappetitlich. Im alten Mesopotamien steht »pissing on« für »anbeten«. Da jedoch die E-Mail nicht aus dem Zweistromland kam, bleibt die wahre Bedeutung wohl unserer Fantasie überlassen.

⁴⁰ <http://www.youtube.com/watch?v=ZA2kqAIoZM>

⁴¹ Natürlich kann Eclipse auch Klassendateien für Java 1.0 erzeugen, nur die IDE selbst benötigt mindestens Java 5.

The screenshot shows the Eclipse Downloads page. At the top, there's a banner for 'EclipseCon Europe 2011' with a call to 'Early Registration ends Sept 30th Register Now!'. Below the banner, the main navigation menu includes Home, Downloads, Users, Members, Committers, Resources, Projects, and About Us. There's also a search bar with a 'Google Custom Search' button and a 'Search' button.

The main content area is titled 'Eclipse Downloads' and features a navigation bar with links for Packages, Developer Builds, Projects, and Indigo Torrents. A 'Hint' section on the right provides information about Java runtime requirements. The 'Installing Eclipse' section contains links for an Install Guide, Known Issues, and Updating Eclipse. The 'Related Links' section lists Source Code, Documentation, Make a Donation, Forums, Eclipse Indigo (3.7), Eclipse Helios (3.6), and Eclipse Galileo (3.5).

The central part of the page displays a list of software packages with their details and download links:

- Eclipse IDE for Java EE Developers, 212 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse IDE for Java Developers, 128 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse IDE for C/C++ Developers (includes Incubating components), 107 MB: Windows 32 Bit, Windows 64 Bit
- Actuate BIRT Designer Pro** (Promoted Download): Windows 32 Bit, Windows 64 Bit
- Eclipse IDE for Java and Report Developers, 242 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse IDE for JavaScript Web Developers, 110 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse for RCP and RAP Developers, 181 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse Modeling Tools, 271 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse for Testers, 90 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse Classic 3.7.1, 174 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse IDE for Parallel Application Developers (includes Incubating components), 186 MB: Windows 32 Bit, Windows 64 Bit
- Eclipse for Scout Developers, 175 MB: Windows 32 Bit, Windows 64 Bit

At the bottom of the list, there's a link for 'Looking for the Eclipse for PHP Developers Package? More Info'.

At the very bottom of the page, there are links for Home, Privacy Policy, Terms of Use, Copyright Agent, Legal, and Contact Us. The copyright notice at the bottom right states 'Copyright © 2011 The Eclipse Foundation. All Rights Reserved.'

Abbildung 1.9: <http://www.eclipse.org/downloads>

Hinweis

Erst Eclipse 3.7.1, Eclipse 3.8 und Eclipse 4.2 unterstützen die neuen Spracheigenschaften von Java 7, und der Editor verarbeitet Java-Quellcode korrekt.

Eclipse gliedert sich in unterschiedliche *Pakete*, die bei NetBeans *Bundles* heißen:

- **Eclipse IDE for Java Developers:** Die kleinste Version zum Entwickeln von Java SE-Anwendungen
- **Eclipse IDE for Java EE Developers:** Diese Version enthält einen XML-Editor und Erweiterungen für die Entwicklung von Webanwendungen und Java EE-Applikationen. Die eingebundenen Unterprojekte heißen *Web Standard Tools* (WST) und *J2EE Standard Tools Project* (JST).
- **Eclipse Classic 3.7:** Wie die *Eclipse IDE for Java Developers*, nur mit Quellen und Dokumentation.
- **Eclipse IDE for C/C++ Developers:** Eclipse als Entwicklungsumgebung für C(++)-Programmierer. Das kleinste Paket, da es ausschließlich für die Programmiersprache C(++) und nicht für Java ist.

Auf der Download-Seite sind neben der aktuellen Version auch die letzten Releases zu finden. Die Hauptversionen heißen *Maintenance Packages*. Neben ihnen gibt es *Stable Builds* und für Mutige die *Integration Builds* und *Nightly Builds*, die einen Blick auf kommende Versionen werfen. Standardmäßig sind Beschriftungen der Entwicklungsumgebung in englischer Sprache, doch gibt es mit den *Eclipse Language Packs* Übersetzungen etwa für Deutsch, Spanisch, Italienisch, Japanisch, Chinesisch und weitere Sprachen. Für die Unterprojekte (WST, JST) gibt es individuelle Updates.



Hinweis

Im Buch setzen wir für die Webentwicklung die *Eclipse IDE for Java EE Developers* voraus. Für alle anderen Kapitel ist das Paket *Eclipse IDE for Java Developers* ausreichend.

1.7.1 Eclipse starten

Eine Installation von Eclipse im typischen Sinne mit einem Installer ist nicht erforderlich, dennoch beschreiben die folgenden Schritte die Benutzung unter Windows.

Nach dem Download und Auspacken des Zip-Archivs gibt es einen Ordner *eclipse* mit der ausführbaren Datei *eclipse.exe* und Unterverzeichnisse für nachinstallierbare Plugins. Das Eclipse-Verzeichnis lässt sich frei wählen.

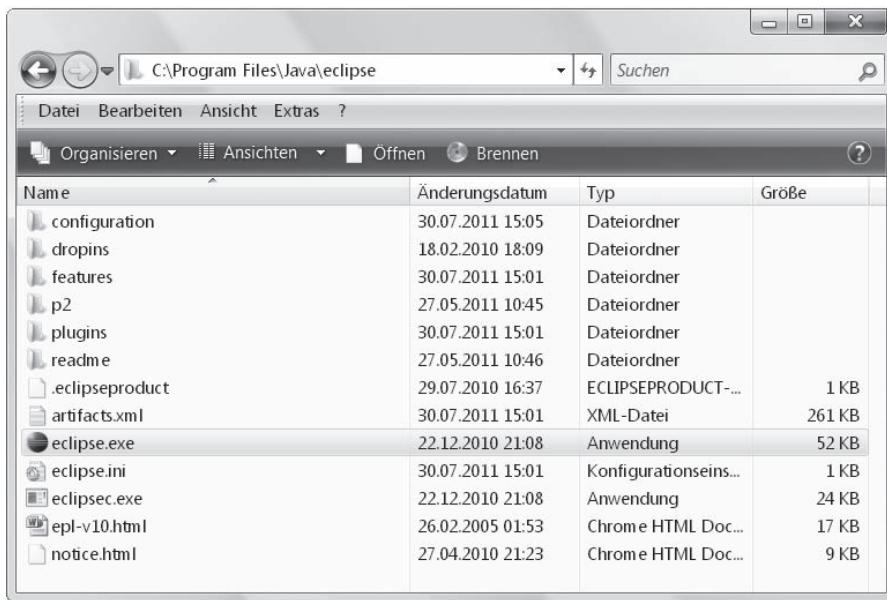


Abbildung 1.10: Eclipse-Ordner

Nach dem Start von *eclipse.exe* folgen ein Willkommensbildschirm und ein Dialog wie dieser:

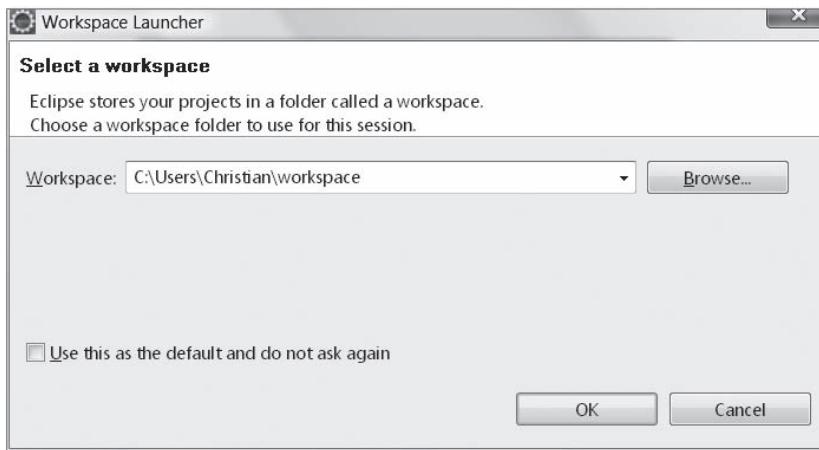


Abbildung 1.11: Workspace auswählen

Mit einer Eclipse-Instanz ist ein *Workspace* verbunden. Das ist ein Verzeichnis, in dem Eclipse-Konfigurationsdaten, Dateien zur Änderungsverfolgung und standardmäßig

Quellcode-Dateien sowie Binärdateien gespeichert sind. Der Workspace kann später gewechselt werden, doch ist nur ein Workspace zur gleichen Zeit aktiv; er muss zu Beginn der Eclipse-Sitzung festgelegt werden. Wir belassen es bei dem Home-Verzeichnis des Benutzers und können den Haken aktivieren, um beim nächsten Start nicht noch einmal gefragt zu werden.

Es folgt das Hauptfenster von Eclipse mit einem Hilfsangebot inklusive Tutorials für Einsteiger und mit Erklärungen, was in der Version neu ist, für Fortgeschrittene. Ein Klick auf das × rechts vom abgerundeten Reiter WELCOME schließt die Ansicht.



Abbildung 1.12: Eclipse-Startbildschirm

1.7.2 Das erste Projekt anlegen

Nach dem Start von Eclipse muss ein Projekt angelegt (oder eingebunden) werden – ohne dieses lässt sich kein Java-Programm ausführen. Im Menü ist dazu FILE • NEW • PROJECT... auszuwählen. Alternativ führt auch die erste Schaltfläche in der Symbolleiste zu diesem Dialog. Es öffnet sich ein Wizard, mit dessen Hilfe wir ein NEW PROJECT erzeugen können.

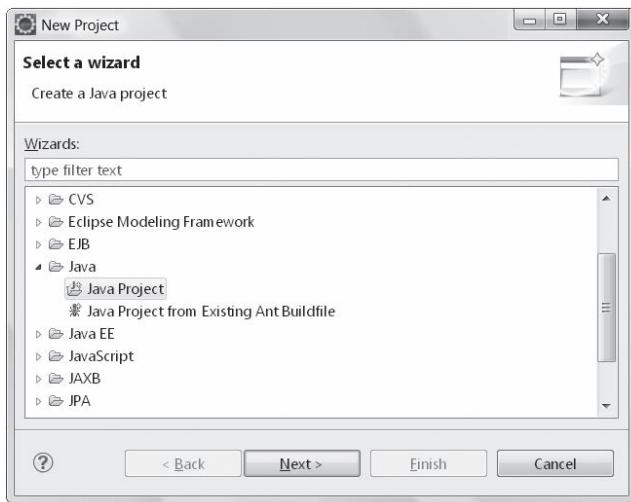


Abbildung 1.13: Dialog für neues Projekt

Der Klick auf NEXT > blendet einen neuen Dialog für weitere Einstellungen ein.

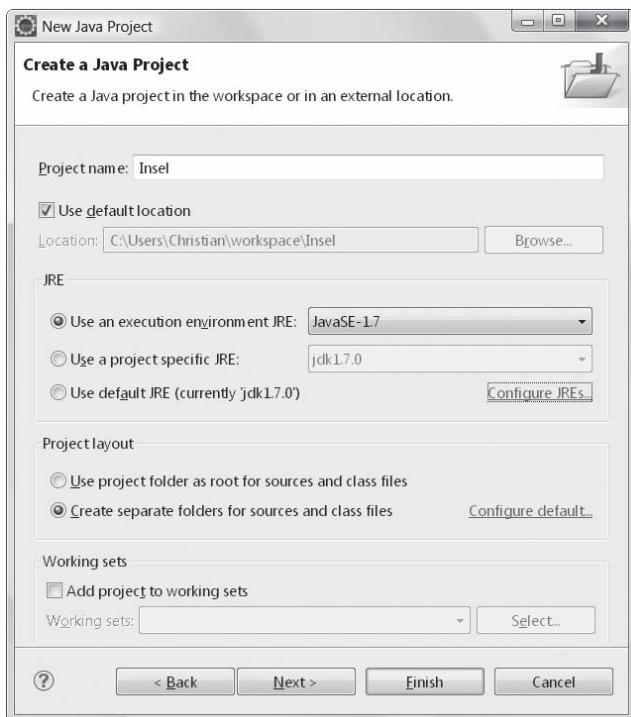


Abbildung 1.14: Dialog für neues Java-Projekt

Unter PROJECT NAME geben wir einen Namen für unser Projekt ein: »Insel«. Mit dem Projekt ist ein Pfad verbunden, in dem die Quellcodes und übersetzten Klassen gespeichert sind. Standardmäßig speichert Eclipse die Projekte im Workspace ab, wir können aber einen anderen Ordner wählen; belassen wir es hier bei einem Unterverzeichnis im Workspace. Im Rahmen JRE steht bei USE DEFAULT JRE nicht unbedingt das gewünschte JDK (etwa für Java 7). Das lässt sich global für alle folgenden Projekte einstellen oder lokal nur für dieses neue Projekt. Wir wollen global die Einstellung ändern und gehen daher auf CONFIGURE JREs..., was uns zu einem zentralen Konfigurationsdialog führt.

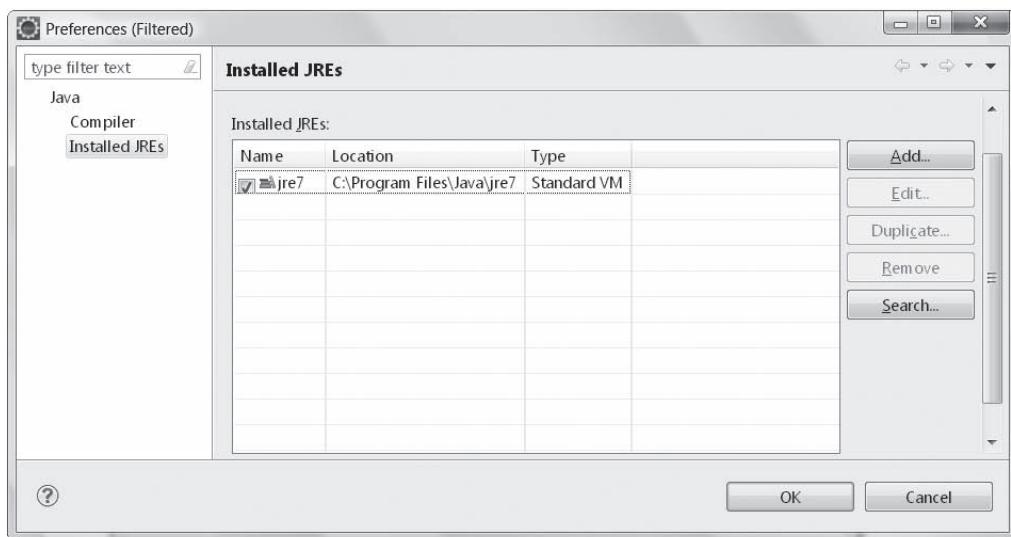


Abbildung 1.15: JRE-Einstellungen nach der Installation

Dort nehmen wir zwei Einstellungen vor: Als Erstes setzen wir die passende Laufzeitumgebung, und dann setzen wir den Compiler auf die gewünschte Version. Sind wir im linken Baum unter INSTALLED JREs, suchen wir mit SEARCH... die Platte nach JVMs ab. Die Liste kann je nach Installation kürzer sein und andere Einträge enthalten. Dann setzen wir ein Häkchen beim gewünschten JDK, in unserem Fall bei JDK1.7.0 (siehe Abbildung 1.16).

Die zweite Einstellung betrifft im Baum den Eintrag COMPILER (siehe Abbildung 1.17). Dort wählen wir unter COMPILER COMPLIANCE LEVEL das Gewünschte, also etwa die Version 1.7. Nach der Dialogbestätigung mit OK folgt ein kleiner Hinweis, den wir mit YES bestätigen.

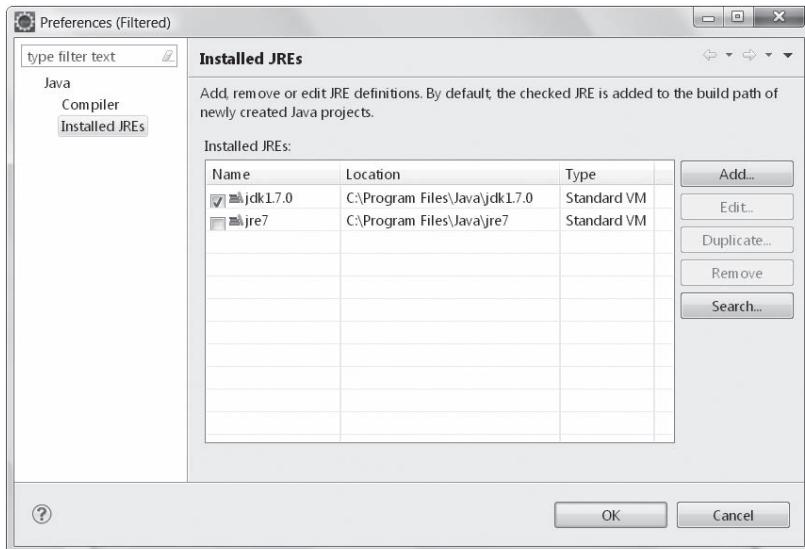


Abbildung 1.16: JRE-Einstellungen nach dem Hinzufügen einer neuen Laufzeitumgebung

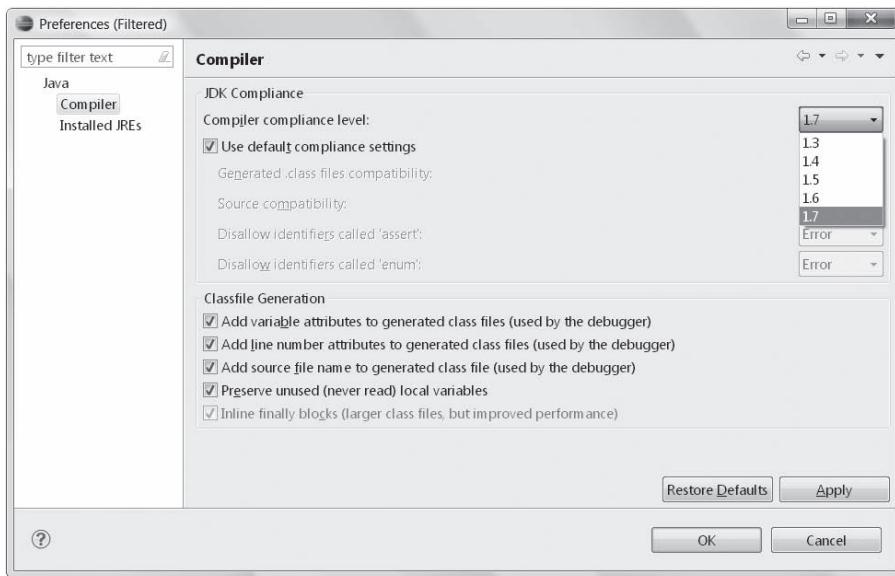


Abbildung 1.17: Java 7 Compiler einstellen

Das bringt uns zum Dialog NEW JAVA PROJECT zurück. Die Schaltfläche FINISH schließt das Anlegen ab. Da es sich um ein Java-Projekt handelt, möchte Eclipse auch in eine Java-Ansicht gehen, und den folgenden Dialog sollten wir mit YES bestätigen.

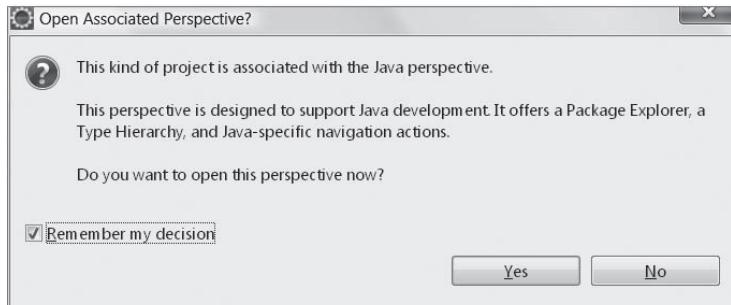


Abbildung 1.18: Perspektivenwechsel für die Java-Umgebung

Jetzt arbeiten wir mit einem Teil von Eclipse, der sich *Workbench* nennt. Welche Ansichten Eclipse platziert, bestimmt die Perspektive (engl. *perspective*). Zu einer Perspektive gehören Ansichten (engl. *views*) und Editoren. Im Menüpunkt **WINDOW • OPEN PERSPECTIVE** lässt sich diese Perspektive ändern, doch um in Java zu entwickeln, ist die Java-Perspektive im Allgemeinen die beste. Das ist die, die Eclipse auch automatisch gewählt hat, nachdem wir das Java-Projekt angelegt haben.

Jede Ansicht lässt sich per Drag & Drop beliebig umsetzen. Die Ansicht **OUTLINE** oder **TASK LIST** auf der rechten Seite lässt sich auf diese Weise einfach an eine andere Stelle schieben – unter dem **PACKAGE EXPLORER** ist sie meistens gut aufgehoben.

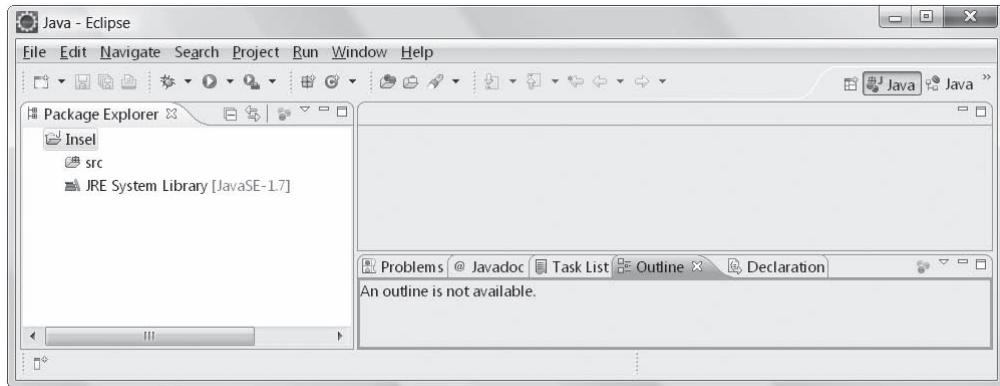


Abbildung 1.19: Eclipse mit dem »Insel«-Java-Projekt und verschobenen Views

1.7.3 Eine Klasse hinzufügen

Dem Projekt können nun Dateien – das heißt Klassen, Java-Archive, Grafiken oder andere Inhalte – hinzugefügt werden. Auch lassen sich in das Verzeichnis nachträglich Da-

teien einfügen, die Eclipse dann direkt anzeigt. Doch beginnen wir mit dem Hinzufügen einer Klasse aus Eclipse. Dazu aktivieren wir über FILE • NEW • CLASS ein neues Fenster. Das Fenster öffnet sich ebenfalls nach der Aktivierung der Schaltfläche mit dem grünen C in der Symbolleiste bzw. im Kontextmenü unter src.

Notwendig ist der Name der Klasse; hier SQUARED. Wir wollen auch einen Schalter für PUBLIC STATIC VOID MAIN(STRING[] ARGS) setzen, damit wir gleich eine Einstiegsmethode haben, in der sich unser erster Quellcode platzieren lässt.

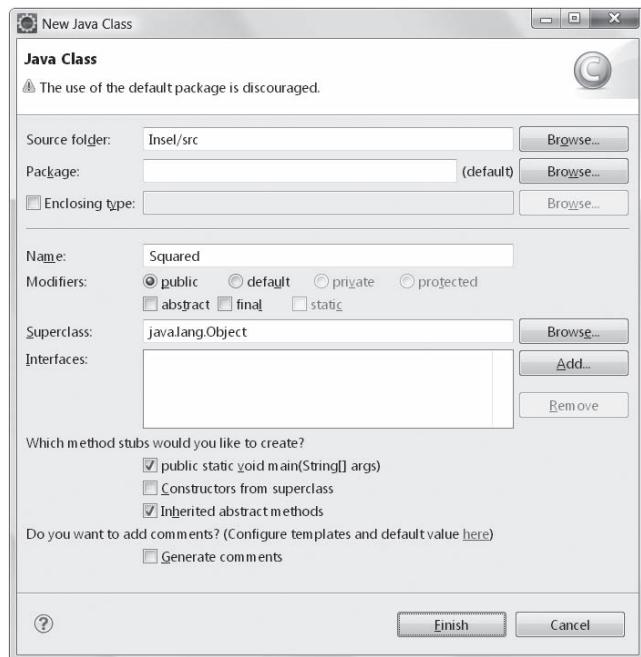


Abbildung 1.20: Neue Klasse in Eclipse anlegen

Nach dem FINISH fügt Eclipse diese Klasse unserem Projektbaum hinzu, erstellt also eine Java-Datei im Dateisystem und öffnet sie gleichzeitig im Editor. In die main()-Methode schreiben wir zum Testen:

```
int n = 2;
System.out.println( "Quadrat: " + n * n );
```

Eclipse besitzt keine Schaltfläche zum Übersetzen. Zum einen lässt Eclipse automatisch einen Compiler im Hintergrund laufen – sonst könnten wir die Fehlermeldungen zur Tippzeit nicht sehen –, und zum anderen nimmt Eclipse das Speichern zum Anlass, einen Übersetzungsvorgang zu starten.

1.7.4 Übersetzen und ausführen

Damit Eclipse eine bestimmte Klasse mit einer `main()`-Methode ausführt, können wir mehrere Wege gehen. Wird zum ersten Mal Programmcode einer Klasse ausgeführt, können wir rechts neben dem grünen Kreis mit dem Play-Symbol auf den Pfeil klicken und im Popup-Menü RUN AS und anschließend JAVA APPLICATION auswählen. Ein anderer Weg: **Alt** + **Shift** + **X**, dann **J**.

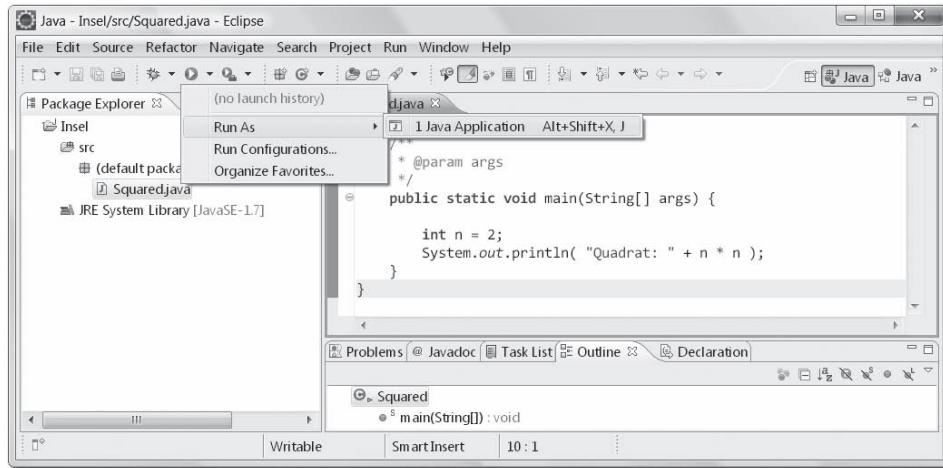


Abbildung 1.21: Java-Applikationen ausführen

Anschließend startet die JVM die Applikation. Assoziiert Eclipse einmal mit einem Start eine Klasse, reicht in Zukunft ein Aufruf mit **Strg** + **F11**. Unten in der Ansicht mit der Aufschrift CONSOLE ist die Ausgabe zu sehen.

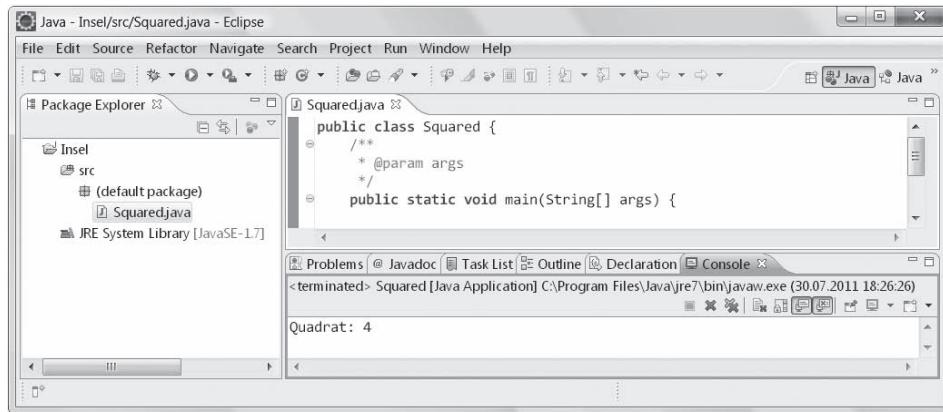


Abbildung 1.22: Ausgaben der Java-Programme in der Console-View

1.7.5 JDK statt JRE *

Beim ersten Start sucht Eclipse eine installierte Java-Version. Das ist im Allgemeinen das JRE, das ohne Dokumentation daherkommt, was beim Programmieren sehr unpraktisch ist. Erst das Java JDK enthält die Dokumentation, die Eclipse aus dem Quellcode extrahiert. Im ersten Schritt hatten wir schon das JDK eingebunden. Wer das übersprungen hat, der kann es nachholen: Es beginnt mit dem globalen Konfigurationsdialog unter **WINDOW • PREFERENCES...** Hier sollte jeder Entwickler einmal die Konfigurationsmöglichkeiten durchgehen. Für unser JRE/JDK-Problem lässt sich links im Baum **JAVA** ausfalten und anschließend im Zweig **INSTALLED JREs** nach einem JDK suchen. Die Schaltfläche **SEARCH...** aktiviert einen Dateiauswahl dialog, der zum Beispiel auf *C:\Program Files\Java* steht – nicht auf dem Verzeichnis mit der Installation selbst! Nach der Suche befindet sich in der Liste ein JDK, wo wir das Häkchen setzen. Schließlich bestätigen wir den Dialog mit **OK**.

1.7.6 Start eines Programms ohne Speicheraufforderung

In der Standardeinstellung fragt Eclipse vor der Übersetzung und Ausführung mit einem Dialog nach, ob noch nicht gesicherte Dateien gespeichert werden sollen.



Abbildung 1.23: Vor dem Ausführen muss die Datei gespeichert werden.

Dort kann das Häkchen gesetzt werden, das die Quellen immer speichert.

In der Regel soll die Entwicklungsumgebung selbst die veränderten Dateien vor dem Übersetzen speichern. Es gibt noch einen anderen Weg, um dies einzustellen. Dazu muss eine Einstellung in der Konfiguration vorgenommen werden: Unter **WINDOW • PREFERENCES** öffnen wir wieder das Konfigurationsfenster und wählen den Zweig **RUN/DEBUG** und dort den Unterzweig **LAUNCHING**. Rechts unter **SAVE REQUIRED DIRTY EDITORS BEFORE LAUNCHING** aktivieren wir dann den Schalter **ALWAYS**.

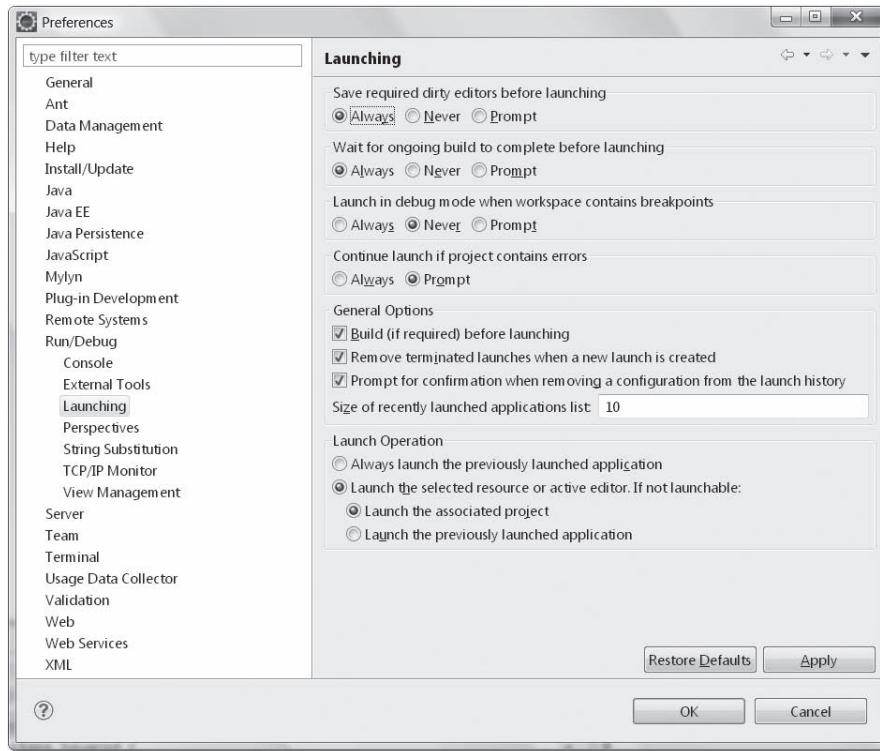


Abbildung 1.24: Editoren immer vor dem Ausführen speichern

1.7.7 Projekt einfügen, Workspace für die Programme wechseln

Alle Beispielprogramme im Buch gibt es auf der DVD oder im Netz. Die Beispielprogramme eines Kapitels befinden sich in einem eigenen Verzeichnis, sodass etwa für Kapitel 4, »Der Umgang mit Zeichenketten«, der Name *1_04_Chars.Strings* vergeben ist. Um das Ausprobieren noch einfacher zu machen, ist jedes Beispiel-Verzeichnis ein eigenständiges Eclipse-Projekt. Um zum aktuellen Workspace die Beispiele aus Kapitel 4 hinzuzunehmen, wählen wir im Menü FILE • IMPORT..., dann im Dialog der Sektion GENERAL die Optionen EXISTING PROJECTS INTO WORKSPACE und anschließend NEXT. Bei SELECT ROOT DIRECTORY wählen wir unter BROWSE das Verzeichnis *1_04_Chars.Strings* des Dateisystems. Zurück im IMPORT-Dialog, erscheint nun unter PROJECTS der Projektname *04_Chars.Strings*. Nach einem FINISH ist das Projekt mit den Beispielen für Zeichenketten Teil des eigenen Workspace.

Das Projekt wird nicht physisch in den Workspace kopiert (dazu muss der Schalter COPY PROJECTS INTO WORKSPACE gewählt sein), sondern nur referenziert.

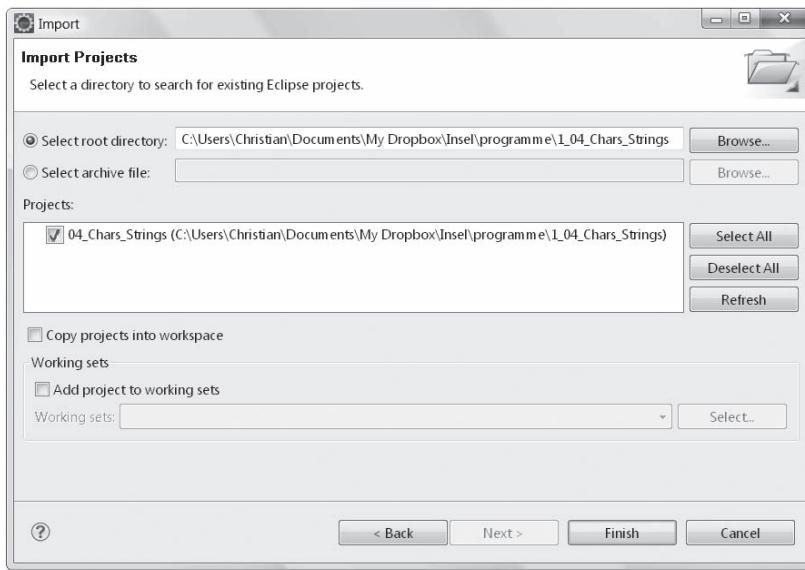


Abbildung 1.25: Importieren von Projekten in den Eclipse-Workspace

Ein nicht mehr benötigtes Projekt im Workspace schließen wir mit der Operation CLOSE PROJECT im Kontextmenü des Projekts. Ebenfalls im Kontextmenü unter DELETE (oder über `Entf` auf der Tastatur) lässt sich ein Projekt löschen. Beim Löschen folgt noch eine Sicherheitsabfrage.

Da alle Beispielprogramme des Buchs auch als Eclipse-Workspace mit je einem Eclipse-Projekt pro Kapitel organisiert sind, lassen sich auch alle Buchbeispiele gleichzeitig einbinden. Dazu ist FILE • SWITCH WORKSPACE... zu wählen und im folgenden Dialog der Pfad zu den Programmen anzulegen.

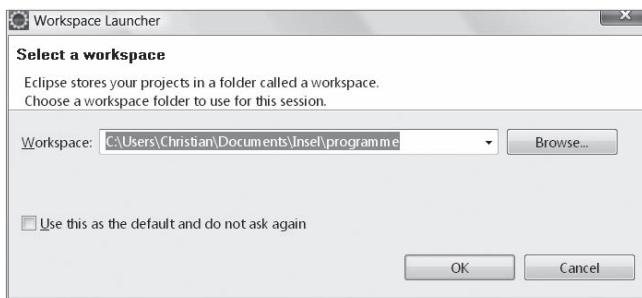


Abbildung 1.26: Zu einem neuen Workspace wechseln

Eclipse beendet sich jetzt und startet anschließend mit dem neuen Workspace.

1.7.8 Plugins für Eclipse

Zusätzliche Anwendungen, die in Eclipse integriert werden können, werden *Plugins* genannt. Ein Plugin besteht aus einer Sammlung von Dateien in einem Verzeichnis oder Java-Archiv. Für die Installation gibt es mehrere Möglichkeiten: Eine davon besteht darin, den Update-Manager zu bemühen, der automatisch im Internet das Plugin lädt; die andere besteht darin, ein Archiv zu laden, das in das *plugin*-Verzeichnis von Eclipse entpackt wird. Beim nächsten Start erkennt Eclipse automatisch das Plugin und integriert es (ein Neustart von Eclipse bei hinzugenommenen Plugins war bislang immer nötig).

Hunderte von Plugins sind verfügbar, einige auserwählte stellt <http://www.tutego.de/java/eclipse/plugin/eclipse-plugins.html> zusammen.

1.8 NetBeans im Speziellen

Die folgenden Abschnitte beschreiben, wie mit NetBeans ein Projekt aufgebaut wird. Bei der Installation sollte die neueste Version *NetBeans 7* für Java 7 installiert werden.

1.8.1 NetBeans-Bundles

Je nach Anwendungsgebiet gibt es von NetBeans (<http://netbeans.org/downloads/>) unterschiedliche Bundles.

Die wichtigsten Bundles (mit Größenangaben für die Version 7.0) sind:

- **Java SE:** Enthält mit 66 MiB alles Nötige zur Entwicklung von Java SE-Anwendungen.
- **Java EE:** Bietet neben der Kern-IDE Tools zur Entwicklung von Web- und Java-Enterprise-Anwendungen. Integriert in den 157 MiB auch den Servlet-Container *Tomcat* und den Java EE-Application-Server *GlassFish*.
- **All:** Enthält in 244 MiB alles, auch Werkzeuge für Ruby, PHP und C++, aber (bisher) nicht JavaFX.

Weiterhin liefert Oracle zwei spezielle Bundles für C(++) und PHP aus. Im Folgenden werden wir uns mit der einfachen Java SE-Version von NetBeans begnügen.

NetBeans IDE 7.0.1-Download

6.9.1 | 7.0.1 | Entwicklung | Archiv

Emailadresse (optional): IDE-Sprache: English | Plattform: Windows

Newsletter abbonieren: Monatlich Wöchentlich NetBeans darf mich kontaktieren

Achtung! Technologien in grau nicht für alle Plattformen.

NetBeans IDE Download-Pakete				
Unterstützte Technologien *	Java SE	Java EE	C/C++	PHP
NetBeans Platform SDK	●	●		●
Java SE	●	●		●
Java EE		●		●
Java ME				●
Java Card™ 3 Connected				●
C/C++			●	●
Groovy				●
PHP				●
Mitgelieferte Server				
GlassFish Server Open Source Edition 3.1.1		●		●
Apache Tomcat 7.0.14	●			●
	Download	Download	Download	Download
	Kostenlos, 66 MB	Kostenlos, 158 MB	Kostenlos, 44 MB	Kostenlos, 42 MB
				Kostenlos, 245 MB

* Mit der Pluginverwaltung der IDE können Sie jederzeit Pakete hinzufügen und entfernen (Extras | Plugins).

Java 6 wird zur Installation und Ausführung der PHP und C/C++ NetBeans-Pakete benötigt. Das [neueste Java](#) bekommen Sie auf [java.com](#).

JDK 6 wird zur Installation und Ausführung der Java SE, Java EE und 'All'-NetBeans-Pakete benötigt. Sie können das [eigenständige JDK](#) herunterladen, oder das [neueste Paket mit JDK und NetBeans IDE für Java SE](#).

Wichtige rechtliche Information:

NetBeans Community Distributions are available under a Dual License consisting of the Common Development and Distribution License (CDDL) v1.0 and GNU General Public License (GPL) v2. See the [third-party readme](#) for external components included in NetBeans and their associated licenses.

[Shop](#) [SiteMap](#) [About Us](#) [Contact](#) [Legal](#) By use of this website, you agree to the [NetBeans Policies and Terms of Use](#)

Abbildung 1.27: Die NetBeans-Download-Seite mit unterschiedlichen Bundles

1.8.2 NetBeans installieren

Nach dem Start der unter Windows ausführbaren Datei *netbeans-7.0.xyz-javase-windows.exe* erscheint das Installationsfenster. Hier sind zwei Lizenzbedingungen zu akzeptieren, und dann ist ein Installationsverzeichnis zu wählen.

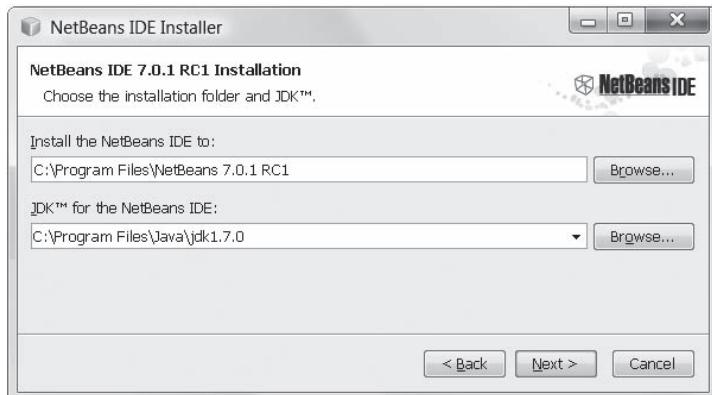


Abbildung 1.28: Installation von NetBeans

Nach dem NEXT und INSTALL kommt NetBeans auf die Platte. Es schließt mit einem Dialog, der dem Benutzer die Wahl lässt, an Oracle Statistikdaten zu übermitteln. FINISH schließt die Installation endgültig ab.

1.8.3 NetBeans starten

Anders als Eclipse bindet sich NetBeans unter Windows stärker in das System ein und ist auch im Startmenü präsent. So lässt es sich auch starten. Das erste Fenster bietet ähnlich wie Eclipse einen Reiter mit START PAGE, doch der Reiter lässt sich schließen.

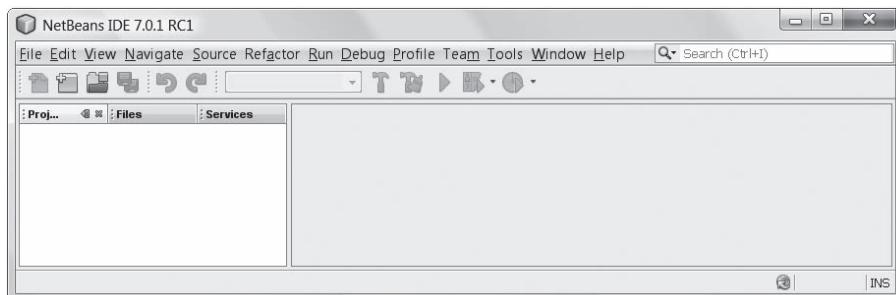


Abbildung 1.29: Die Arbeitsoberfläche von NetBeans ohne Startseite

1.8.4 Ein neues NetBeans-Projekt anlegen

Der erste Menüpunkt unter FILE ist NEW PROJECT... und ermöglicht es uns, ein neues Java-Projekt anzulegen:

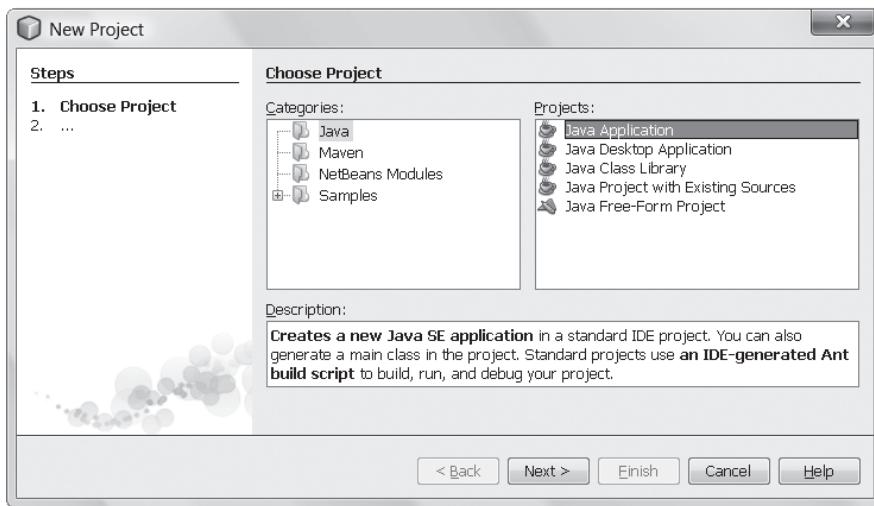


Abbildung 1.30: Neues Java-Projekt anlegen

Wähle unter JAVA den Punkt JAVA APPLICATION, dann NEXT. Als Projektname trage »Insel« ein und bei CREATE MAIN CLASS den Klassennamen SQUARED.

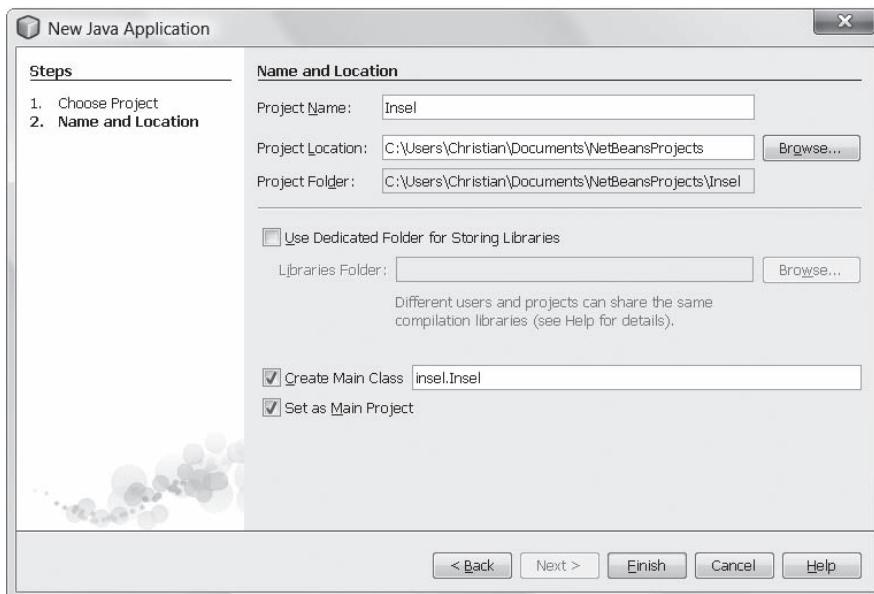


Abbildung 1.31: Den Projektnamen und den Klassennamen eintragen

Ein Klick auf FINISH schließt den Prozess ab.

1.8.5 Ein Java-Programm starten

In der `main()`-Methode tragen wir Folgendes ein:

```
int n = 2;  
System.out.println( "Quadrat: " + n * n );
```

In der Symbolleiste gibt es ein grünes Dreieck, das das Programm startet; alternativ können wir auch **F6** drücken.

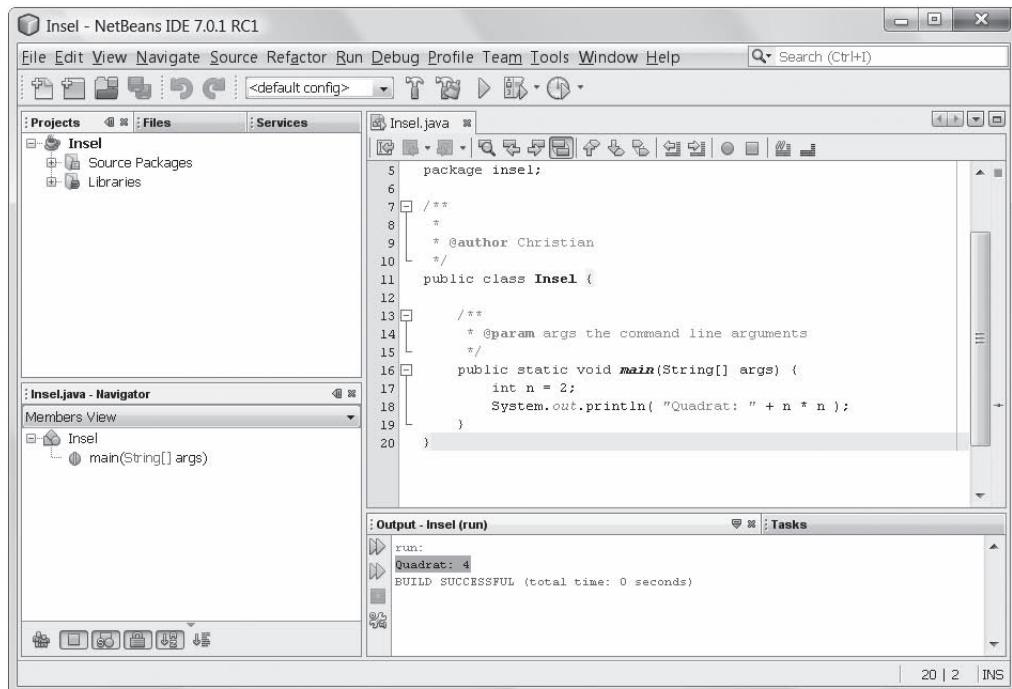


Abbildung 1.32: Ausgabe von NetBeans

1.8.6 Einstellungen

Es gibt zwei Arten von Einstellungen: globale wie neue Plugins und projektbezogene, lokale. Eine Einstellung, die wir ändern wollen, betrifft den Compiler, der standardmäßig auf Java 6 steht. Um dies zu ändern, wählen wir im Projekt-Kontextmenü (in unserem Fall INSEL) den Eintrag PROPERTIES. Es öffnet sich ein Dialog, und wenn links im Baum SOURCE aktiviert ist, lässt sich unten bei SOURCE/BINARY FORMAT JDK 7 einstellen.

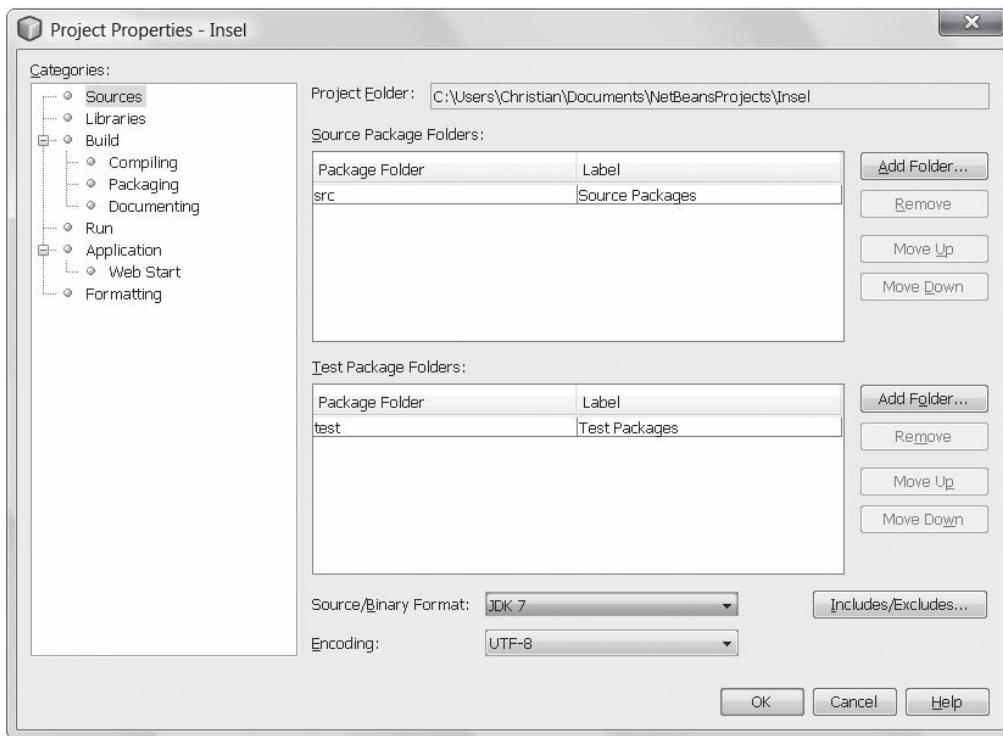


Abbildung 1.33: Java 7 aktivieren

1.9 Zum Weiterlesen

Sun gab ein kleines Büchlein namens »Hello World(s) – From Code to Culture: A 10 Year Celebration of Java Technology« heraus (ISBN 0131888676), das Informationen zur Entstehung von Java bietet. Weitere Online-Informationen zur Entwicklermannschaft und zum *7-Projekt liefern <http://tutego.de/go/star7> sowie <http://tutego.de/go/javasaga>. Die virtuelle Maschine selbst gibt es für Geschichtsliebhaber in allen Versionen unter <http://tutego.de/go/javaarchive>. Dass Java eine robuste Sprache ist, haben auch Google und eBay erkannt. Google nutzt Java für viele Lösungen, etwa *Google Mail* oder auch das neue *Google+*. Das Auktionshaus eBay nutzt Oracle-Hardware und realisiert seine Geschäftslogik in Java. Die Seite <http://tutego.de/go/sunebay> berichtet darüber, und auch der FAZ war diese Tatsache einen Artikel wert: <http://tutego.de/go/faz>. Auch das Community-Netzwerk LinkedIn (<http://www.linkedin.com/>) nutzt Oracle-x86-Hardware,

Java als Programmiersprache und verwaltetet damit über 25 Millionen Mitglieder, die im Schnitt 2 Millionen Nachrichten am Tag schicken.⁴²

Eclipse ist das Standardwerkzeug der Softwareentwickler.⁴³ Entwickler, die Eclipse einsetzen, können in der Hilfe unter <http://tutego.de/go/eclipseshelp> viel Interessantes erfahren und sollten diverse Plugins als Ergänzung evaluieren. Leser, die Webapplikationen oder Web-Services entwickeln, können einen Blick auf die Webseite <http://tutego.de/go/webtoolscommunity> werfen. Durch Plugins kann die IDE erweitert werden, und so kann ein Entwickler auch in Bereiche wie Webentwicklung mit PHP oder Mainframe-Anwendungen mit COBOL vordringen, die nichts mit Java zu tun haben. Durch die riesige Anzahl an Plugins ist aber nicht immer offensichtlich, welches Plugin gut ist – das Problem ist vergleichbar mit iPhone-Anwendungen im Apple App Store – und wie es mit anderen Plugins zusammenspielt. Aufeinander abgestimmte Plugins sind da Gold wert, und das Unternehmen Genuitec verdient mit *MyEclipse* (<http://www.myeclipseide.com/>) sein Geld, indem es gut aufeinander abgestimmte hochwertige Plugins bündelt. Einen Schritt weiter gehen *JBuilder* von *Embarcadero* (früher *Borland*) und das *SAP NetWeaver Developer Studio*, die Eclipse gleich zur Basis ihrer eigenen Entwicklungsumgebungen gemacht haben.

Die IDE *NetBeans* von Oracle kann mit gelungenen Innovationen, etwa dem GUI-Builder *Matisse* (<http://tutego.de/go/nbform>), Unterstützung für Java EE 6 (<http://tutego.de/go/nbglassfish>), der neuen Skript-Sprache *JavaFX* (<http://tutego.de/go/nbfx>) und *Java ME* (<http://tutego.de/go/nbmobility>) mit einem super visuellen Editor dagegenhalten. Ich selbst verwende gern Matisse als GUI-Builder, aber entwickle üblicherweise in Eclipse.

⁴² <http://developers.sun.com/learning/javaoneonline/2008/pdf/TS-5234.pdf>

⁴³ Manche sagen, nur weil es frei ist – sonst würden sie viel lieber IntelliJ nutzen ...



Kapitel 2

Imperative Sprachkonzepte

2

»Wenn ich eine Oper hundertmal dirigiert habe, dann ist es Zeit,

sie wieder zu lernen.«

– Arturo Toscanini (1867–1957)

Ein Programm in Java wird nicht umgangssprachlich beschrieben, sondern ein Regelwerk und eine Grammatik definieren die Syntax und die Semantik. In den nächsten Abschnitten werden wir kleinere Beispiele für Java-Programme kennenlernen, und dann ist der Weg frei für größere Programme.

2.1 Elemente der Programmiersprache Java

Wir wollen im Folgenden über das Regelwerk, die Grammatik und die Syntax der Programmiersprache Java sprechen und uns unter anderem über die Unicode-Kodierung, Tokens sowie Bezeichner Gedanken machen. Bei der Benennung einer Methode zum Beispiel dürfen wir aus einer großen Anzahl Zeichen wählen; der Zeichenvorrat nennt sich *Lexikalik*.

Die Syntax eines Java-Programms definiert die Tokens und bildet so das Vokabular. Richtig geschriebene Programme müssen aber dennoch nicht korrekt sein. Unter dem Begriff *Semantik* fassen wir daher die Bedeutung eines syntaktisch korrekten Programms zusammen. Die Semantik bestimmt, was das Programm macht. Die Abstraktionsreihenfolge ist also Lexikalik, Syntax und Semantik. Der Compiler durchläuft diese Schritte, bevor er den Bytecode erzeugen kann.

2.1.1 Token

Ein *Token* ist eine lexikalische Einheit, die dem Compiler die Bausteine des Programms liefert. Der Compiler erkennt an der Grammatik einer Sprache, welche Folgen von Zei-

chen ein Token bilden. Für Bezeichner heißt dies beispielsweise: »Nimm die nächsten Zeichen, solange auf einen Buchstaben nur Buchstaben oder Ziffern folgen.« Eine Zahl wie 1982 bildet zum Beispiel ein Token durch folgende Regel: »Lies so lange Ziffern, bis keine Ziffer mehr folgt.« Bei Kommentaren bilden die Kombinationen /* und */ ein Token.¹

Whitespace

Problematisch wird es in einer Sprache immer dann, wenn der Compiler die Tokens nicht voneinander unterscheiden kann. Daher fügen wir *Trennzeichen* (engl. *white-space*) ein, die auch *Wortzwischenräume* genannt werden. Zu den Trennern zählen Leerzeichen, Tabulatoren, Zeilenvorschub- und Seitenvorschubzeichen. Außer als Trennzeichen haben diese Zeichen keine Bedeutung. Daher können sie in beliebiger Anzahl zwischen die Tokens gesetzt werden. Das heißt auch, beliebig viele Leerzeichen sind zwischen Tokens gültig. Und da wir damit nicht geizen müssen, können sie einen Programmabschnitt enorm verdeutlichen. Programme sind besser lesbar, wenn sie luftig formatiert sind.

Folgendes ist alles andere als gut zu lesen, obwohl der Compiler es akzeptiert:

```
class _{static long _
(long _,long __) {
return __==0 ? __+ 1:
__==0?__(__-1,1):__(_
-1,__,-1)) ; }
static {int _=2 ,__
= 2;System.out.print(
"a("+_+', '+__+ ")="+
__(_,_)) ;System
.exit(1);} // (C) Ulli
```

Neben den Trennern gibt es noch 9 Zeichen, die als *Separator* definiert werden:

; , . () { } []

¹ Das ist in C(++) unglücklich, denn so wird ein Ausdruck *s/*t nicht wie erwartet geparsrt. Erst ein Leerzeichen zwischen dem Geteiltzeichen und dem Stern »hilft« dem Parser, die gewünschte Division zu erkennen.

2.1.2 Textkodierung durch Unicode-Zeichen

Java kodiert Texte durch *Unicode-Zeichen*. Jedem Zeichen ist ein eindeutiger Zahlenwert (engl. *code point*) zugewiesen, sodass zum Beispiel das große A an Position 65 liegt. Der Unicode-Zeichensatz beinhaltet die ISO-US-ASCII-Zeichen² von 0 bis 127 (hexadezimal 0x00 bis 0x7f, also 7 Bit) und die erweiterte Kodierung nach ISO 8859-1 (Latin-1), die Zeichen von 128 bis 255 hinzunimmt. Mehr Details zu Unicode liefert Kapitel 4, »Der Umgang mit Zeichenketten«.

2.1.3 Bezeichner

Für Variablen (und damit Konstanten), Methoden, Klassen und Schnittstellen werden *Bezeichner* vergeben – auch *Identifizierer* (von engl. *identifier*) genannt –, die die entsprechenden Bausteine anschließend im Programm identifizieren. Unter Variablen sind dann Daten verfügbar. Methoden sind die Unterprogramme in objektorientierten Programmiersprachen, und Klassen sind die Bausteine objektorientierter Programme.

Ein Bezeichner ist eine Folge von Zeichen, die fast beliebig lang sein kann (die Länge ist nur theoretisch festgelegt). Die Zeichen sind Elemente aus dem Unicode-Zeichensatz, und jedes Zeichen ist für die Identifikation wichtig.³ Das heißt, ein Bezeichner, der 100 Zeichen lang ist, muss auch immer mit allen 100 Zeichen korrekt angegeben werden. Manche C- und FORTRAN-Compiler sind in dieser Hinsicht etwas großzügiger und bewerten nur die ersten Stellen.

Beispiel

zB

Im folgenden Java-Programm sind die Bezeichner fett und unterstrichen gesetzt.

```
class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
    }
}
```

² <http://en.wikipedia.org/wiki/ASCII>

³ Die Java-Methoden `Character.isJavaIdentifierStart()`/`isJavaIdentifierPart()` stellen auch fest, ob Zeichen Java-Identifier sind.

zB Beispiel (Forts.)

Dass String fett und unterstrichen ist, hat seinen Grund, denn String ist eine Klasse und kein eingebauter Datentyp wie int. Zwar wird die Klasse String in Java bevorzugt behandelt – das Plus kann Zeichenketten zusammenhängen –, aber es ist immer noch ein Klassentyp.

Aufbau der Bezeichner

Jeder Java-Bezeichner ist eine Folge aus *Java-Buchstaben* und *Java-Ziffern*,⁴ wobei der Bezeichner mit einem Java-Buchstaben beginnen muss. Ein Java-Buchstabe umfasst nicht nur unsere lateinischen Buchstaben aus dem Bereich »A« bis »Z« (auch »a« bis »z«), sondern auch viele weitere Zeichen aus dem Unicode-Alphabet, etwa den Unterstrich, Währungszeichen – wie die Zeichen für Dollar (\$), Euro (€), Yen (¥) – oder griechische Buchstaben. Auch wenn damit viele wilde Zeichen als Bezeichner-Buchstaben grundsätzlich möglich sind, sollte doch die Programmierung mit englischen Bezeichnernamen erfolgen. Es ist noch einmal zu betonen, dass Java streng zwischen Groß- und Kleinschreibung unterscheidet.

Die folgende Tabelle listet einige gültige Bezeichner auf:

Gültige Bezeichner	Grund
Mami	Mami besteht nur aus Alphazeichen und ist daher korrekt.
RAPHAEL IST LIEB	Unterstriche sind erlaubt.
bóöléáñ	Ist korrekt, auch wenn es Akzente enthält.
α	Das griechische Alpha ist ein gültiger Java-Buchstabe.
REZE\$\$SION	Das Dollar-Zeichen ist ein gültiger Java-Buchstabe.
¥€\$	Tatsächlich auch gültige Java-Buchstaben

Tabelle 2.1: Beispiele für gültige Bezeichner in Java

⁴ Ob ein Zeichen ein Buchstabe ist, stellt die statische Methode Character.isLetter() fest; ob er ein gültiger Bezeichner-Buchstabe ist, sagen die Funktionen isJavaIdentifierStart() für den Startbuchstaben und isJavaIdentifierPart() für den Rest.

Ungültige Bezeichner dagegen sind:

Ungültige Bezeichner	Grund
2und2macht4	Das erste Symbol muss ein Java-Buchstabe sein und keine Ziffer.
hose gewaschen	Leerzeichen sind in Bezeichnern nicht erlaubt.
Faster!	Das Ausrufezeichen ist, wie viele Sonderzeichen, ungültig.
null, class	Der Name ist schon von Java belegt. Null – Groß-/Kleinschreibung ist relevant – oder cláss wären möglich.

Tabelle 2.2: Beispiele für ungültige Bezeichner in Java

Hinweis

In Java-Programmen bilden sich Bezeichnernamen oft aus zusammengesetzten Wörtern einer Beschreibung. Dies bedeutet, dass in einem Satz wie »open file read only« die Leerzeichen entfernt werden und die nach dem ersten Wort folgenden Wörter mit Großbuchstaben beginnen. Damit wird aus dem Beispielsatz anschließend »openFileRead-Only«. Sprachwissenschaftler nennen einen Großbuchstaben inmitten von Wörtern *Binnenmajuskel*.



2.1.4 Literale

Ein *Literal* ist ein konstanter Ausdruck. Es gibt verschiedene Typen von Literalen:

- die Wahrheitswerte `true` und `false`
- integrale Literale für Zahlen, etwa `122`
- Zeichenliterale, etwa `'X'` oder `'\n'`
- Fließkommaliterale, etwa `12.567` oder `9.999E-2`
- Stringliterale für Zeichenketten, wie `"Paolo Pinkas"`
- `null` steht für einen besonderen Referenztyp.

Beispiel



Im folgenden Java-Programm sind die beiden Literale **fett** und **unterstrichen** gesetzt.

zB Beispiel (Forts.)

```
class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
        System.out.println( 1 + 2 );
    }
}
```

2.1.5 Reservierte Schlüsselwörter

Bestimmte Wörter sind als Bezeichner nicht zulässig, da sie als *Schlüsselwörter* vom Compiler besonders behandelt werden. Schlüsselwörter bestimmen die »Sprache« eines Compilers.

zB Beispiel

Reservierte Schlüsselwörter sind im Folgenden fett und unterstrichen gesetzt.

```
class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
    }
}
```

Schlüsselwörter und Literale in Java

Nachfolgende Zeichenfolgen sind Schlüsselwörter (beziehungsweise Literale im Fall von `true`, `false` und `null`)⁵ und sind in Java daher nicht als Bezeichnernamen möglich.

⁵ Siehe dazu Abschnitt 3.9, »Keywords«, der Sprachdefinition unter http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.9.

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

Tabelle 2.3: Reservierte Schlüsselwörter in Java

Obwohl die mit † gekennzeichneten Wörter zurzeit nicht von Java benutzt werden, können doch keine Variablen dieses Namens deklariert werden.

2.1.6 Zusammenfassung der lexikalischen Analyse

Übersetzt der Compiler Java-Programme, so beginnt er mit der lexikalischen Untersuchung des Quellcodes. Wir haben dabei die zentralen Elemente schon kennengelernt, und diese sollen hier noch einmal zusammengefasst werden. Nehmen wir dazu das folgende einfache Programm:

```
class Application
{
    public static void main( String[] args )
    {
        String text = "Hallo Welt " + 21;
        System.out.println( text );
    }
}
```

Der Compiler überliest alle Kommentare, und die Trennzeichen bringen den Compiler von Token zu Token. Folgende Tokens lassen sich im Programm ausmachen:

Token-Typ	Beispiel	Erklärung
Bezeichner	Application, main, args, text, System, out, println	Namen für Klasse, Variable, Methode, ...
Schlüsselwort	class, public, static, void	Reservierte Wörter
Literal	"Hallo Welt", 21	Konstante Werte, wie Strings, Zahlen, ...
Operator	=, +	Operator für Zuweisungen, Berechnungen, ...
Trennzeichen	(,), {, }, ;	Symbole, die neben dem Trennzeichen die Tokens trennen

Tabelle 2.4: Token des Beispielprogramms

2.1.7 Kommentare

Programmieren heißt nicht nur, einen korrekten Algorithmus in einer Sprache auszudrücken, sondern auch, unsere Gedanken verständlich zu formulieren. Dies geschieht beispielsweise durch eine sinnvolle Namensgebung für Programmobjekte wie Klassen, Methoden und Variablen. Ein selbsterklärender Klassename hilft den Entwicklern erheblich. Doch die Lösungsidee und der Algorithmus werden auch durch die schönsten Variablennamen nicht zwingend klarer. Damit Außenstehende (und nach Monaten wir selbst) unsere Lösungsidee schnell nachvollziehen und später das Programm erweitern oder abändern können, werden *Kommentare* in den Quelltext geschrieben. Sie dienen nur den Lesern der Programme, haben aber auf die Abarbeitung keine Auswirkungen.

Unterschiedliche Kommentartypen

In Java gibt es zum Formulieren von Kommentaren drei Möglichkeiten:

- **Zeilenkommentare:** Sie beginnen mit zwei Schrägstrichen⁶ // und kommentieren den Rest einer Zeile aus. Der Kommentar gilt von diesen Zeichen an bis zum Ende der Zeile, also bis zum Zeilenumbruchzeichen.
- **Blockkommentare:** Sie kommentieren in /* */ Abschnitte aus. Der Text im Blockkommentar darf selbst kein */ enthalten, denn Blockkommentare dürfen nicht verschachtelt sein.

⁶ In C++ haben die Entwickler übrigens das Zeilenkommentarzeichen // aus der Vor-Vorgängersprache BCPL wieder eingeführt, das in C entfernt wurde.

- **JavaDoc-Kommentare:** Das sind besondere Blockkommentare, die JavaDoc-Kommentare mit `/** */` enthalten. Ein JavaDoc-Kommentar beschreibt etwa die Methode oder die Parameter, aus denen sich später die API-Dokumentation generieren lässt.

Schauen wir uns ein Beispiel an, in dem alle drei Kommentartypen vorkommen:

```
/*
 * Der Quellcode ist public domain.
 */
// Magic. Do not touch.
/**
 * @author Christian Ullenboom
 */
class DoYouHaveAnyCommentsToMake      // TODO: Umbenennen
{
    // When I wrote this, only God and I understood what I was doing
    // Now, God only knows
    public static void main( String[] args /* Kommandozeilenargument */ )
    {
    }
}
```

Für den Compiler sieht die Klasse mit den Kommentaren genauso aus wie ohne, also wie `class DoYouHaveAnyCommentsToMake { }.` Im Bytecode steht exakt das Gleiche – alle Kommentare werden vom Compiler verworfen.

Kommentare mit Stil

Alle Kommentare und Bemerkungen sollten in Englisch verfasst werden, um Projektmitgliedern aus anderen Ländern das Lesen zu erleichtern. Für allgemeine Kommentare sollten wir die Zeichen `//` benutzen. Sie haben zwei Vorteile:

- Bei Editoren, die Kommentare nicht farbig hervorheben, oder bei einer einfachen Quellcodeausgabe auf der Kommandozeile lässt sich ersehen, dass eine Zeile, die mit `//` beginnt, ein Kommentar ist. Den Überblick über einen Quelltext zu behalten, der für mehrere Seiten mit den Kommentarzeichen `/*` und `*/` unterbrochen wird, ist schwierig. Zeilenkommentare machen deutlich, wo Kommentare beginnen und wo sie enden.

- Der Einsatz der Zeilenkommentare eignet sich besser dazu, während der Entwicklungs- und Debug-Phase Codeblöcke auszukommentieren. Benutzen wir zur Programmdokumentation die Blockkommentare, so sind wir eingeschränkt, denn Kommentare dieser Form können wir nicht verschachteln. Zeilenkommentare können einfacher geschachtelt werden.



Die Tastenkombination `Strg` + `7` – oder `Strg` + `/`, was das Kommentarzeichen »« noch deutlicher macht – kommentiert eine Zeile aus. Eclipse setzt dann vor die Zeile die Kommentarzeichen `//`. Sind mehrere Zeilen selektiert, kommentiert die Tastenkombination alle markierten Zeilen mit Zeilenkommentaren aus. In einer kommentierten Zeile nimmt ein erneutes `Strg` + `7` die Kommentare einer Zeile wieder zurück.



`Strg` + `Shift` + `C` in kommentiert eine Zeile bzw. einen Block in NetBeans ein und aus.

2.2 Von der Klasse zur Anweisung

Programme sind Ablauffolgen, die im Kern aus Anweisungen bestehen. Sie werden zu größeren Bausteinen zusammengesetzt, den Methoden, die wiederum Klassen bilden. Klassen selbst werden in Paketen gesammelt, und eine Sammlung von Paketen wird als Java-Archiv ausgeliefert.

2.2.1 Was sind Anweisungen?

Java zählt zu den imperativen Programmiersprachen, in denen der Programmierer die Abarbeitungsschritte seiner Algorithmen durch *Anweisungen* (engl. *statements*) vor gibt. Anweisungen können unter anderem sein:

- Ausdrucksanweisungen, etwa für Zuweisungen oder Methodenaufrufe
- Fallunterscheidungen, zum Beispiel mit `if`
- Schleifen für Wiederholungen, etwa mit `for` oder `do-while`



Hinweis

Diese Befehlsform ist für Programmiersprachen gar nicht selbstverständlich, da es Sprachen gibt, die zu einer Problembeschreibung selbstständig eine Lösung finden. Ein Vertreter dieser Art von Sprachen ist *Prolog*.

Hinweis (Forts.)

Die Schwierigkeit hierbei besteht darin, die Aufgabe so präzise zu beschreiben, dass das System eine Lösung finden kann. Auch die Datenbanksprache SQL ist keine imperative Programmiersprache, denn wie das Datenbankmanagement-System zu unserer Abfrage die Ergebnisse ermittelt, müssen und können wir weder vorgeben noch sehen.

2.2.2 Klassendeklaration

Programme setzen sich aus Anweisungen zusammen. In Java können jedoch nicht einfach Anweisungen in eine Datei geschrieben und dem Compiler übergeben werden. Sie müssen zunächst in einen Rahmen gepackt werden. Dieser Rahmen heißt *Kompilationseinheit* (engl. *compilation unit*) und deklariert eine Klasse mit ihren Methoden und Variablen.

Die nächsten Programmcodezeilen werden am Anfang etwas befremdlich wirken (wir erklären die Elemente später genauer). Die folgende Datei erhält den (frei wählbaren) Namen *Application.java*:

Listing 2.1: Application.java

```
public class Application
{
    public static void main( String[] args )
    {
        // Hier ist der Anfang unserer Programme
        // Jetzt ist hier Platz für unsere eigenen Anweisungen
        // Hier enden unsere Programme
    }
}
```

Hinter den beiden Schrägstrichen // befindet sich ein Kommentar. Er gilt bis zum Ende der Zeile und dient dazu, Erläuterungen zu den Quellcodezeilen hinzuzufügen, die den Code verständlicher machen.

Eclipse zeigt Schlüsselwörter, Literale und Kommentare farbig an. Diese Farbgebung lässt sich unter WINDOW • PREFERENCES ändern.



Unter TOOLS • OPTIONS und dann im Bereich FONTS & COLOR lassen sich bei NetBeans der Zeichensatz und die Farbbelegung ändern.



Java ist eine objektorientierte Programmiersprache, die Programmlogik außerhalb von Klassen nicht erlaubt. Aus diesem Grund deklariert die Datei *Application.java* mit dem Schlüsselwort `class` eine Klasse `Application`, um später eine Methode mit der Programmlogik anzugeben. Der Klassenname darf grundsätzlich beliebig sein, doch besteht die Einschränkung, dass in einer mit `public` deklarierten Klasse der Klassenname so lauten muss wie der Dateiname. Alle Schlüsselwörter in Java beginnen mit Kleinbuchstaben, und Klassennamen beginnen üblicherweise mit Großbuchstaben.

In den geschweiften Klammern der Klasse folgen Deklarationen von Methoden, also Unterprogrammen, die eine Klasse anbietet. Eine Methode ist eine Sammlung von Anweisungen unter einem Namen.

2.2.3 Die Reise beginnt am `main()`

Wir programmieren hier eine besondere Methode, die sich `main()` nennt. Die Schlüsselwörter davor und die Angabe in dem Paar runder Klammern hinter dem Namen müssen wir einhalten. Die Methode `main()` ist für die Laufzeitumgebung etwas ganz Besonderes, denn beim Aufruf des Java-Interpreters mit einem Klassennamen wird unsere Methode als Erstes ausgeführt.⁷ Demnach werden genau die Anweisungen ausgeführt, die innerhalb der geschweiften Klammern stehen. Halten wir uns fälschlicherweise nicht an die Syntax für den Startpunkt, so kann der Interpreter die Ausführung nicht beginnen, und wir haben einen semantischen Fehler produziert, obwohl die Methode selbst korrekt gebildet ist. Innerhalb von `main()` befindet sich ein Parameter mit dem Namen `args`. Der Name ist willkürlich gewählt, wir werden allerdings immer `args` verwenden.



Dass Fehler unterkriegt werden, hat sich als Visualisierung durchgesetzt. Eclipse gibt im Falle eines Fehlers sehr viele Hinweise. Ein Fehler im Quellcode wird von Eclipse mit einer gekringelten roten Linie angezeigt. Als weiterer Indikator wird (unter Umständen erst beim Speichern) ein kleines rundes Kreuz an der Fehlerzeile angezeigt. Gleichzeitig findet sich im Schieberegler ein kleiner roter Block. Im PACKAGE EXPLORER findet sich ebenfalls ein Hinweis auf Fehler. Zum nächsten Fehler bringt die Tastenkombination `Strg` + `.` (Punkt) zurück.

⁷ Na ja, so ganz präzise ist das auch nicht. In einem static-Block könnten wir auch einen Funktionsaufruf setzen, doch das wollen wir hier einmal nicht annehmen. static-Blöcke werden beim Laden der Klassen in die virtuelle Maschine ausgeführt. Andere Initialisierungen sind dann auch schon gemacht.

[Strg] + [.] führt bei NetBeans nur in der Fehleransicht zum Fehler selbst, nicht aber aus dem Java-Editor heraus. Die Tastenkombination kann im Editor einfach über **TOOLS • OPTION** gesetzt werden. Dann wählen wir **KEYMAP**, geben unter **SEARCH** den Suchbegriff »**ERROR**« ein und selektieren dann in der **ACTION-Spalte** **NEXT ERROR IN EDITOR**. In der zweiten Spalte **SHORTCUT** setzen wir den Fokus und drücken **[Strg] + [.]**. Dann beenden wir den Dialog mit **OK**.

2.2.4 Der erste Methodenaufruf: `println()`

In Java gibt es eine große Klassenbibliothek, die es Entwicklern erlaubt, Dateien anzulegen, Fenster zu öffnen, auf Datenbanken zuzugreifen, Web-Services aufzurufen und vieles mehr. Am untersten Ende der Klassenbibliothek stehen Methoden, die eine gewünschte Operation ausführen.

Eine einfache Methode ist `println()`. Sie gibt Meldungen auf dem Bildschirm (der Konsole) aus. Innerhalb der Klammern von `println()` können wir *Argumente* angeben. Die `println()`-Methode erlaubt zum Beispiel *Zeichenketten* (ein anderes Wort ist *Strings*) als Argumente, die dann auf der Konsole erscheinen. Ein String ist eine Folge von Buchstaben, Ziffern oder Sonderzeichen in doppelten Anführungszeichen.

Implementieren⁸ wir damit eine vollständige Java-Klasse mit einem *Methodenaufruf*, die über `println()` etwas auf dem Bildschirm ausgibt:

Listing 2.2: Application.java

```
class Application
{
    public static void main( String[] args )
    {
        // Start des Programms

        System.out.println( "Hallo Javanesen" );

        // Ende des Programms
    }
}
```

⁸ »Implementieren« stammt vom lateinischen Wort »implere« ab, was für »erfüllen« und »ergänzen« steht.



Hinweis

Der Begriff *Methode* ist die korrekte Bezeichnung für ein Unterprogramm in Java – die *Java Language Specification* (JLS) verwendet den Begriff *Funktion* nicht.

2.2.5 Atomare Anweisungen und Anweisungssequenzen

Methodenaufrufe wie `System.out.println()`, die *leere Anweisung*, die nur aus einem Semikolon besteht, oder auch Variablendeklarationen (die später vorgestellt werden) nennen sich *atomare* (auch *elementare*) *Anweisungen*. Diese unteilbaren Anweisungen werden zu *Anweisungssequenzen* zusammengesetzt, die Programme bilden.



zB Beispiel

Eine Anweisungssequenz:

```
System.out.println( "Wer morgens total zerknittert aufsteht, " );
System.out.println( "hat am Tag die besten Entfaltungsmöglichkeiten." );
;
System.out.println();
;
```

Leere Anweisungen (also die Zeilen mit dem Semikolon) gibt es im Allgemeinen nur bei Endloswiederholungen.

Die Laufzeitumgebung von Java führt jede einzelne Anweisung der Sequenz in der angegebenen Reihenfolge hintereinander aus. Anweisungen und Anweisungssequenzen dürfen nicht irgendwo stehen, sondern nur an bestimmten Stellen, etwa innerhalb eines Methodenkörpers.

2.2.6 Mehr zu `print()`, `println()` und `printf()` für Bildschirmausgaben

Die meisten Methoden verraten durch ihren Namen, was sie leisten, und für eigene Programme ist es sinnvoll, aussagekräftige Namen zu verwenden. Wenn die Java-Entwickler die Ausgabemethode statt `println()` einfach `glubschi()` genannt hätten, bliebe uns der Sinn der Methode verborgen. `println()` zeigt jedoch durch den Wortstamm »print« an, dass etwas geschrieben wird. Die Endung `ln` (kurz für *line*) bedeutet, dass noch ein Zeilenvorschubzeichen ausgegeben wird. Umgangssprachlich heißt das: Eine neue Aus-

gabe beginnt in der nächsten Zeile. Neben `println()` existiert die Bibliotheksmethode `print()`, die keinen Zeilenvorschub anhängt.

Die `printXXX()`-Methoden⁹ können in Klammern unterschiedliche Argumente bekommen. Ein Argument ist ein Wert, den wir der Methode beim Aufruf mitgeben. Auch wenn wir einer Methode keine Argumente übergeben, muss beim Aufruf hinter dem Methodennamen ein Klammernpaar folgen. Dies ist konsequent, da wir so wissen, dass es sich um einen Methodenaufruf handelt und um nichts anderes. Andernfalls führt es zu Verwechslungen mit Variablen.

Überladene Methoden

Java erlaubt Methoden, die gleich heißen, denen aber unterschiedliche Dinge übergeben werden können; diese Methoden nennen wir *überladen*. Die `printXXX()`-Methoden sind zum Beispiel überladen und akzeptieren neben dem Argumenttyp `String` auch Typen wie einzelne Zeichen, Wahrheitswerte oder Zahlen – oder auch gar nichts:

Listing 2.3: OverloadedPrintln.java

```
public class OverloadedPrintln
{
    public static void main( String[] args )
    {
        System.out.println( "Verhaften Sie die üblichen Verdächtigen!" );
        System.out.println( true );
        System.out.println( -273 );
        System.out.println();           // Gibt eine Leerzeile aus
        System.out.println( 1.6180339887498948 );
    }
}
```

Die Ausgabe ist:

Verhaften Sie die üblichen Verdächtigen!

true

-273

1.618033988749895

⁹ Abkürzung für Methoden, die mit `print` beginnen, also `print()` und `println()`.

In der letzten Zeile ist gut zu sehen, dass es Probleme mit der Genauigkeit gibt – dieses Phänomen werden wir uns noch genauer anschauen.



Ist in Eclipse eine andere Ansicht aktiviert, etwa weil wir auf das Konsolenfenster geklickt haben, bringt die Taste **F12** uns wieder in den Editor zurück.

Variable Argumentlisten

Java unterstützt seit der Version 5 variable Argumentlisten, was bedeutet, dass es möglich ist, bestimmten Methoden beliebig viele Argumente (oder auch kein Argument) zu übergeben. Die Methode `printf()` erlaubt zum Beispiel variable Argumentlisten, um gemäß einer Formatierungsanweisung – einem String, der immer als erstes Argument übergeben werden muss – die nachfolgenden Methodenargumente aufzubereiten und auszugeben:

Listing 2.4: VarArgs.java

```
public class VarArgs
{
    public static void main( String[] args )
    {
        System.out.printf( "Was sagst du?%n" );
        System.out.printf( "%d Kanäle und überall nur %s.%n", 220, "Katzen" );
    }
}
```

Die Ausgabe der Anweisung ist:

```
Was sagst du?
220 Kanäle und überall nur Katzen.
```

Die Formatierungsanweisung `%n` setzt einen Zeilenumbruch, `%d` ist ein Platzhalter für eine Dezimalzahl und `%s` ein Platzhalter für eine Zeichenkette oder etwas, das in einen String konvertiert werden soll. Weitere Platzhalter werden in Abschnitt 4.11, »Ausgaben formatieren«, vorgestellt.

2.2.7 Die API-Dokumentation

Die wichtigste Informationsquelle für Programmierer ist die offizielle API-Dokumentation von Oracle. Zu der Methode `println()` können wir bei der Klasse `PrintStream` zum Beispiel erfahren, dass diese eine Ganzzahl, eine Fließkommazahl, einen Wahrheitswert,

ein Zeichen oder aber eine Zeichenkette akzeptiert. Die Dokumentation ist weder Teil vom JRE noch vom JDK – dafür ist die Hilfe zu groß. Wer über eine permanente Internetverbindung verfügt, kann die Dokumentation online unter <http://tutego.de/go/javaapi> lesen oder sie von der Oracle-Seite <http://www.oracle.com/technetwork/java/javase/downloads/> herunterladen und als Sammlung von HTML-Dokumenten auspacken.

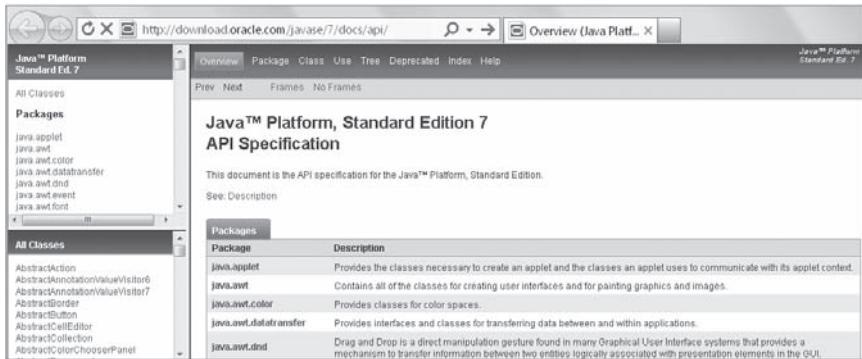


Abbildung 2.1: Online-Dokumentation bei Oracle

Aus Entwicklungsumgebungen ist die API-Dokumentation auch zugänglich, sodass eine Suche auf der Webseite nicht nötig ist.

Eclipse zeigt mithilfe der Tasten $\text{Shift} + \text{F2}$ in einem eingebetteten Browser-Fenster die API-Dokumentation an, wobei die JavaDoc von den Oracle-Seiten kommt. Mithilfe der F2 -Taste bekommen wir ein kleines gelbes Vorschaufenster, das ebenfalls die API-Dokumentation zeigt.

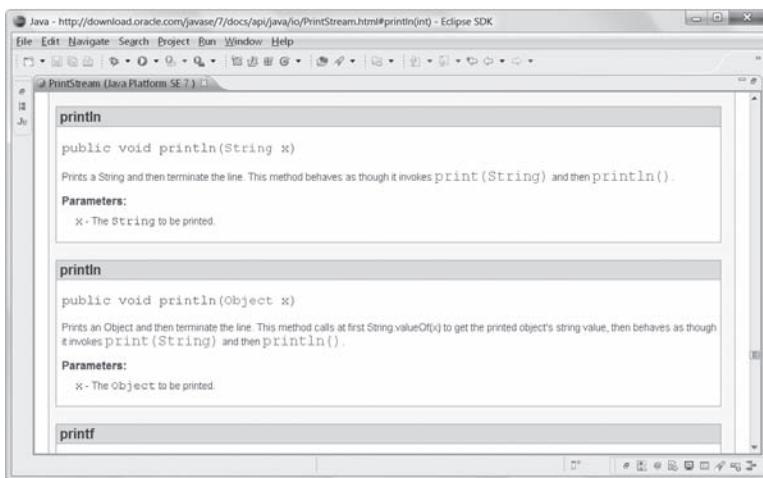
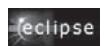


Abbildung 2.2: API-Dokumentation in Eclipse

API-Dokumentation im HTML-Help-Format *

Die Oracle-Dokumentation als Loseblattsammlung hat einen Nachteil, der sich im Programmieralltag bemerkbar macht: Sie lässt sich nur ungenügend durchsuchen. Da die Webseiten statisch sind, können wir nicht einfach nach Methoden forschen, die zum Beispiel auf »listener« enden. Franck Allimant (<http://tutego.de/go/allimant>) übersetzt regelmäßig die HTML-Dokumentation von Oracle in das Format *Windows HTML-Help* (CHM-Dateien), das auch unter Unix und Mac OS X mit der Open-Source-Software <http://xchm.sourceforge.net/> gelesen werden kann. Neben den komprimierten Hilfe-Dateien lassen sich auch die Sprach- und JVM-Spezifikation sowie die API-Dokumentation der Enterprise Edition und der Servlets im Speziellen beziehen.

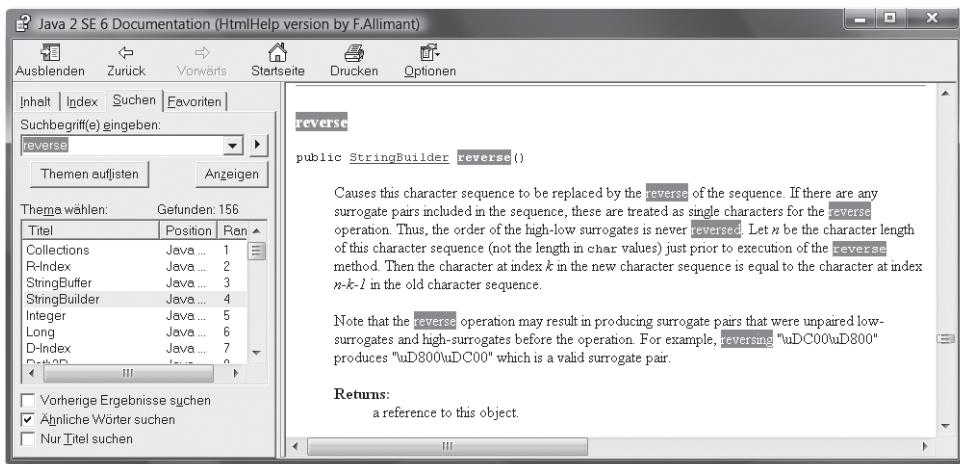


Abbildung 2.3: API-Dokumentation in der Windows-Hilfe

2.2.8 Ausdrücke

Ein *Ausdruck* (engl. *expression*) ergibt bei der Auswertung ein Ergebnis. Im Beispiel *OverloadedPrintln.java* steht in der `main()`-Methode:

```
System.out.println( "Verhaften Sie die üblichen Verdächtigen!" );
System.out.println( true );
System.out.println( -273 );
System.out.println( 1.6180339887498948 );
```

Die Argumente für `println()`, wie der String, der Wahrheitswert oder die Zahlen, sind Ausdrücke. Im dem Beispiel kommt der Ausdruck von einem Literal, aber mit Operatoren lassen sich auch komplexere Ausdrücke wie $(1 + 2) * 1.19$ bilden:

```
System.out.println( (1 + 2) * 1.19 );
```

Der Wert eines Ausdrucks wird auch *Resultat* genannt. Ausdrücke haben immer einen Wert, während das für Anweisungen (wie eine Schleife) nicht gilt. Daher kann ein Ausdruck an allen Stellen stehen, an denen ein Wert benötigt wird, etwa als Argument von `println()`. Dieser Wert ist entweder ein numerischer Wert (von arithmetischen Ausdrücken), ein Wahrheitswert (`boolean`) oder eine Referenz (etwa von einer Objekt-Erzeugung).

2.2.9 Ausdrucksanweisung

In einem Programm reiht sich Anweisung an Anweisung. Auch bestimmte Ausdrücke und Methodenaufrufe lassen sich als Anweisungen einsetzen, wenn sie mit einem Semikolon abgeschlossen sind; wir sprechen dann von einer *Ausdrucksanweisung* (engl. *expression statement*). Jeder Methodenaufruf mit Semikolon bildet zum Beispiel eine Ausdrucksanweisung. Dabei ist es egal, ob die Methode selbst eine Rückgabe liefert oder nicht.

```
System.out.println();      // println() besitzt keine Rückgabe (void)  
Math.random();           // random() liefert eine Fließkommazahl
```

Die Methode `Math.random()` liefert als Ergebnis einen Zufallswert zwischen 0 (inklusiv) und 1 (exklusiv). Da mit dem Ergebnis des Ausdrucks nichts gemacht wird, wird der Rückgabewert verworfen. Im Fall der Zufallsmethode ist das nicht sinnvoll, denn sie macht außer der Berechnung nichts anderes.

Neben Methodenaufrufen mit abschließendem Semikolon gibt es andere Formen von Ausdrucksanweisungen, wie etwa Zuweisungen. Doch allen ist das Semikolon gemeinsam.¹⁰

Hinweis

Nicht jeder Ausdruck kann eine Ausdrucksanweisung sein. `1+2` ist etwa ein Ausdruck, aber `1+2;` – also der Ausdruck mit Semikolon abgeschlossen – ist keine gültige Anweisung. In JavaScript ist so etwas erlaubt, in Java nicht.

¹⁰ Das Semikolon dient auch nicht wie in Pascal zur Trennung von Anweisungen, sondern schließt sie immer ab.

2.2.10 Erste Idee der Objektorientierung

In einer objektorientierten Programmiersprache sind alle Methoden an bestimmte Objekte gebunden (daher der Begriff *objektorientiert*). Betrachten wir zum Beispiel das Objekt Radio: Ein Radio spielt Musik ab, wenn der Einschalter betätigt wird und ein Sender und die Lautstärke eingestellt sind. Ein Radio bietet also bestimmte Dienste (Operationen) an, wie Musik an/aus, lauter/leiser. Zusätzlich hat ein Objekt auch noch einen Zustand, zum Beispiel die Lautstärke oder das Baujahr. Wichtig in objektorientierten Sprachen ist, dass die Operationen und Zustände immer (und da gibt es keine Ausnahmen) an Objekte beziehungsweise Klassen gebunden sind (mehr zu dieser Unterscheidung folgt später). Der Aufruf einer Methode auf einem Objekt richtet die Anfrage genau an ein bestimmtes Objekt. Steht in einem Java-Programm nur die Anweisung lauter, so weiß der Compiler nicht, wen er fragen soll, wenn es etwa drei Radio-Objekte gibt. Was ist, wenn es auch einen Fernseher mit der gleichen Operation gibt? Aus diesem Grund verbinden wir das Objekt, das etwas kann, mit der Operation. Ein Punkt trennt das Objekt von der Operation oder dem Zustand. So gehört `println()` zu einem Objekt `out`, das die Bildschirmausgabe übernimmt. Dieses Objekt `out` wiederum gehört zu der Klasse `System`.

System.out und System.err

Das Laufzeitsystem bietet uns zwei Ausgabekanäle: einen für normale Ausgaben und einen, in den wir Fehler leiten können. Der Vorteil ist, dass über diese Unterteilung die Fehler von der herkömmlichen Ausgabe getrennt werden können. Standardausgaben wandern in `System.out`, und Fehlerausgaben werden in `System.err` weitergeleitet. `out` und `err` sind vom gleichen Typ, sodass die `printXXX()`-Methoden bei beiden gleich sind:

```
System.out.println( "Das ist eine normale Ausgabe" );
System.err.println( "Das ist eine Fehlerausgabe" );
```

Die Objektorientierung wird hierbei noch einmal besonders deutlich. Das `out`- und das `err`-Objekt sind zwei Objekte, die das Gleiche können, nämlich mit `println()` etwas ausgeben. Doch ist es nicht möglich, ohne explizite Objektangabe die Methode `println()` in den Raum zu rufen und von der Laufzeitumgebung zu erwarten, dass diese weiß, ob die Anfrage an `System.out` oder an `System.err` geht.



Abbildung 2.4: Eclipse stellt normale Ausgaben schwarz und Fehlerausgaben rot dar. Damit ist es leicht, zu erkennen, welche Ausgabe in welchen Kanal geschickt wurde.

2.2.11 Modifizierer

Die Deklaration einer Klasse oder Methode kann einen oder mehrere *Modifizierer* (engl. *modifier*) enthalten, die zum Beispiel die Nutzung einschränken oder parallelen Zugriff synchronisieren.

Beispiel

zB

Im folgenden Programm kommen drei Modifizierer vor, die fett und unterstrichen sind:

```
public class Application
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo Welt" );
    }
}
```

Der Modifizierer `public` ist ein *Sichtbarkeitsmodifizierer*. Er bestimmt, ob die Klasse beziehungsweise die Methode für Programmcode anderer Klassen sichtbar ist oder nicht. Der Modifizierer `static` zwingt den Programmierer nicht dazu, vor dem Methodenaufruf ein Objekt der Klasse zu bilden. Anders gesagt: Dieser Modifizierer bestimmt die Eigenschaft, ob sich eine Methode nur über ein konkretes Objekt aufrufen lässt oder eine Eigenschaft der Klasse ist, sodass für den Aufruf kein Objekt der Klasse nötig wird. Wir arbeiten in den ersten beiden Kapiteln nur mit statischen Methoden und werden ab Kapitel 3, »Klassen und Objekte«, nicht-statische Methoden einführen.

2.2.12 Gruppieren von Anweisungen mit Blöcken

Ein *Block* fasst eine Gruppe von Anweisungen zusammen, die hintereinander ausgeführt werden. Anders gesagt: Ein Block ist *eine Anweisung*, die in geschweiften Klammern { } eine Folge von Anweisungen zu einer neuen Anweisung zusammenfasst:

```
{
    Anweisung1;
    Anweisung2;
    ...
}
```

Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann. Der neue Block hat jedoch eine Besonderheit in Bezug auf Variablen, da er einen lokalen Bereich für die darin befindlichen Anweisungen inklusive der Variablen bildet.



Codestyle

Die Zeilen, die in geschweiften Klammern stehen, werden in der Regel mit Leerraum eingekückt. Üblicherweise sind es zwei (wie in diesem Buch) oder vier Leerzeichen. Viele Autoren setzen die geschweiften Klammern in eine eigene Zeile. Diesem Stil folgt auch dieses Buch in der Regel, es sei denn, der Programmcode soll weniger »vertikal wachsen«.

Leerer Block

Ein Block {} ohne Anweisung nennt sich *leerer Block*. Er verhält sich wie eine leere Anweisung, also wie ein Semikolon. In einigen Fällen ist der leere Block mit dem Semikolon wirklich austauschbar, in anderen Fällen erzwingt die Java-Sprache einen Block, der, falls es keine Anweisungen gibt, leer ist, anstatt hier auch ein Semikolon zu erlauben.

Geschachtelte Blöcke

Blöcke können beliebig geschachtelt werden. So ergeben sich innere Blöcke und äußere Blöcke:

```
{
    // Beginn äußerer Block
    {
        // Beginn innerer Block
    }
    // Ende innerer Block
}
// Ende äußerer Block
```

Mit leeren Blöcken ist Folgendes in der statischen Methode `main()` in Ordnung:

```
public static void main( String[] args )
{
    System.out.println( "Hallo Computer" ); {{}}{{}{}}}
}
```

Blöcke spielen eine wichtige Rolle beim Zusammenfassen von Anweisungen, die in Abhängigkeit von Bedingungen einmal oder mehrmals ausgeführt werden. Im Abschnitt 2.5 und 2.6 kommen wir darauf noch einmal praktisch zurück.

2.3 Datentypen, Typisierung, Variablen und Zuweisungen

Java nutzt, wie es für imperative Programmiersprachen typisch ist, Variablen zum Ablegen von Daten. Eine Variable ist ein reservierter Speicherbereich und belegt – abhängig vom Inhalt – eine feste Anzahl von Bytes. Alle Variablen (und auch Ausdrücke) haben einen *Typ*, der zur Übersetzungszeit bekannt ist. Der Typ wird auch *Datentyp* genannt, da eine Variable einen Datenwert, auch *Datum* genannt, enthält. Beispiele für einfache Datentypen sind: Ganzzahlen, Fließkommazahlen, Wahrheitswerte und Zeichen. Der Typ bestimmt auch die zulässigen Operationen, denn Wahrheitswerte lassen sich nicht addieren, Ganzzahlen schon. Dagegen lassen sich Fließkommazahlen addieren, aber nicht Xor-verknüpfen. Da jede Variable einen vom Programmierer vorgegebenen festen Datentyp hat, der zur Übersetzungszeit bekannt ist und sich später nicht mehr ändern lässt, und Java stark darauf achtet, welche Operationen erlaubt sind, und auch von jedem Ausdruck spätestens zur Laufzeit den Typ kennt, ist Java eine *statisch typisierte* und *streueng (stark) typisierte* Programmiersprache.¹¹

Hinweis

In Java muss der Datentyp einer Variablen zur Übersetzungszeit bekannt sein. Das nennt sich dann *statisch typisiert*. Das Gegenteil ist eine *dynamische Typisierung*, wie sie etwa JavaScript verwendet. Hier kann sich der Typ einer Variablen zur Laufzeit ändern, je nachdem, was die Variable enthält.

¹¹ Während in der Literatur bei den Begriffen *statisch getypt* und *dynamisch getypt* mehr oder weniger Einigkeit herrscht, haben verschiedene Autoren unterschiedliche Vorstellungen von den Begriffen *streueng (stark) typisiert* und *schwach typisiert*.

Primitiv- oder Verweis-Typ

Die Datentypen in Java zerfallen in zwei Kategorien:

- *Primitive Typen*: Die primitiven (einfachen) Typen sind die eingebauten Datentypen für Zahlen, Unicode-Zeichen und Wahrheitswerte.
- *Referenztypen*: Mit diesem Datentyp lassen sich Objektverweise etwa auf Zeichenketten, Dialoge oder Datenstrukturen verwalten.

Warum sich damals Sun für diese Teilung entschieden hat, lässt sich mit zwei Gründen erklären:

- Zu der Zeit, als Java eingeführt wurde, kannten viele Programmierer die Syntax und Semantik von C(++) und ähnlichen imperativen Programmiersprachen. Zur neuen Sprache Java zu wechseln, fiel dadurch leichter, und es half, sich sofort auf der Insel zurechtzufinden. Es gibt aber auch Programmiersprachen wie Smalltalk, die keine primitiven Datentypen besitzen.
- Der andere Grund ist die Tatsache, dass häufig vorkommende elementare Rechenoperationen schnell durchgeführt werden müssen und bei einem einfachen Typ leicht Optimierungen durchzuführen sind.

Wir werden uns im Folgenden erst mit primitiven Datentypen beschäftigen. Referenzen werden nur dann eingesetzt, wenn Objekte ins Spiel kommen. Die nehmen wir uns in Kapitel 3, »Klassen und Objekte«, vor.

2.3.1 Primitive Datentypen im Überblick

In Java gibt es zwei Arten eingebauter Datentypen:

- *arithmetische Typen* (ganze Zahlen – auch integrale Typen genannt –, Fließkomma-zahlen, Unicode-Zeichen)
- *Wahrheitswerte* für die Zustände wahr und falsch

Die folgende Tabelle vermittelt dazu einen Überblick. Anschließend betrachten wir jeden Datentyp präziser.

Typ	Belegung (Wertebereich)
boolean	true oder false
char	16-Bit-Unicode-Zeichen (0x0000 ... 0xFFFF)

Tabelle 2.5: Java-Datentypen und ihre Wertebereiche

Typ	Belegung (Wertebereich)
byte	-2^7 bis 2^7 - 1 (-128 ... 127)
short	-2^15 bis 2^15 - 1 (-32.768 ... 32.767)
int	-2^31 bis 2^31 - 1 (-2.147.483.648 ... 2.147.483.647)
long	-2^63 bis 2^63 - 1 (-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807)
float	1,40239846E-45f ... 3,40282347E+38f
double	4,94065645841246544E-324 ... 1,79769131486231570E+308

Tabelle 2.5: Java-Datentypen und ihre Wertebereiche (Forts.)

Bei den Ganzzahlen fällt auf, dass es eine positive Zahl »weniger« gibt als negative.

Für float und double ist das Vorzeichen nicht angegeben, da die kleinsten und größten darstellbaren Zahlen sowohl positiv wie auch negativ sein können. Mit anderen Worten: Die Wertebereiche unterscheiden sich nicht – anders als etwa bei int – in Abhängigkeit vom Vorzeichen. Wer eine »klassische« Darstellung wünscht, der kann sich das so vorstellen: Der Wertebereich (vom double) ist 4,94065645841246544E-324 bis 1,79769131486231570E+308 bzw. mit dem Vorzeichen von etwa -1.8E308 (über -4,9E-324 und +4,9E-324) bis +1.8E308.¹²

Detailwissen



Genau genommen sieht die Sprachgrammatik von Java keine negativen Zahlenliterale vor. Bei einer Zahl wie -1.2 oder -1 ist das Minus der unäre Operator und gehört nicht zur Zahl. Im Bytecode selbst sind die negativen Zahlen natürlich wieder abgebildet.



¹² Es gibt bei Fließkommazahlen noch »Sonderzahlen«, wie plus oder minus Unendlich, aber dazu später mehr.

Die folgende Tabelle zeigt eine etwas andere Darstellung:

Typ	Größe	Format
Ganzzahlen		
byte	8 Bit	Zweierkomplement
short	16 Bit	Zweierkomplement
int	32 Bit	Zweierkomplement
long	64 Bit	Zweierkomplement
Fließkommazahlen		
float	32 Bit	IEEE 754
double	64 Bit	IEEE 754
Weitere Datentypen		
boolean	1 Bit	true, false
char	16 Bit	16-Bit-Unicode

Tabelle 2.6: Java-Datentypen und ihre Größen und Formate



Hinweis

Strings werden bevorzugt behandelt, sind aber lediglich Verweise auf Objekte und kein primitiver Datentyp.

Zwei wesentliche Punkte zeichnen die primitiven Datentypen aus:

- Alle Datentypen haben eine festgesetzte Länge, die sich unter keinen Umständen ändert. Der Nachteil, dass sich bei einigen Hochsprachen die Länge eines Datentyps ändern kann, besteht in Java nicht. In den Sprachen C(++) bleibt dies immer unsicher, und die Umstellung auf 64-Bit-Maschinen bringt viele Probleme mit sich. Der Datentyp `char` ist 16 Bit lang.
- Die numerischen Datentypen `byte`, `short`, `int` und `long` sind vorzeichenbehaftet, Fließkommazahlen sowieso. Dies ist leider nicht immer praktisch, aber wir müssen stets daran denken. Probleme gibt es, wenn wir einem Byte zum Beispiel den Wert 240 zuweisen wollen, denn 240 liegt außerhalb des Wertebereichs, der von -128 bis 127 reicht. Ein `char` ist im Prinzip ein vorzeichenloser Ganzzahltyp.

Wenn wir also die numerischen Datentypen (lassen wir hier `char` außen vor) nach ihrer Größe sortieren wollten, könnten wir zwei Linien für Ganzzahlen und Fließkommazahlen aufbauen:

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

Hinweis

Die Klassen `Byte`, `Integer`, `Long`, `Short`, `Character`, `Double` und `Float` deklarieren die Konstanten `MAX_VALUE` und `MIN_VALUE`, die den größten und kleinsten zulässigen Wert des jeweiligen Wertebereichs bzw. die Grenzen der Wertebereiche der jeweiligen Datentypen angeben.

```
System.out.println( Byte.MIN_VALUE );           // -128
System.out.println( Byte.MAX_VALUE );           // 127
System.out.println( Character.MIN_VALUE );       // '\u0000'
System.out.println( Character.MAX_VALUE );       // '\uFFFF'
System.out.println( Double.MIN_VALUE );          // 4.9E-324
System.out.println( Double.MAX_VALUE );          // 1.7976931348623157E308
```



2.3.2 Variablen-deklarationen

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Um Variablen zu nutzen, müssen sie deklariert (definiert¹³) werden. Die Schreibweise einer Variablen-deklaration ist immer die gleiche: Hinter dem Typnamen folgt der Name der Variablen. Sie ist eine Anweisung und wird daher mit einem Semikolon abgeschlossen. In Java kennt der Compiler von jeder Variablen und jedem Ausdruck genau den Typ.

Deklarieren wir ein paar (lokale) Variablen in der `main()`-Methode:

Listing 2.5: FirstVariable.java

```
public class FirstVariable
{
```

¹³ In C(++) bedeuten Definition und Deklaration etwas Verschiedenes. In Java kennen wir diesen Unterschied nicht und betrachten daher beide Begriffe als gleichwertig. Die Spezifikation spricht nur von *Deklarationen*.

```

public static void main( String[] args )
{
    String name;           // Name
    int age;              // Alter
    double income;         // Einkommen
    char gender;           // Geschlecht ('f' oder 'm')
    boolean isPresident;   // Ist Präsident (true oder false)
    boolean isVegetarian;  // Ist die Person Vegetarier?
}
}

```

Der Typname ist entweder ein einfacher Typ (wie `int`) oder ein Referenztyp. Viel schwieriger ist eine Deklaration nicht – kryptische Angaben wie in C gibt es in Java nicht.¹⁴ Ein Variablenname (der dann Bezeichner ist) kann alle Buchstaben und Ziffern des Unicode-Zeichensatzes beinhalten, mit der Ausnahme, dass am Anfang des Bezeichners keine Ziffer stehen darf. Auch darf der Bezeichnername mit keinem reservierten Schlüsselwort identisch sein.

Mehrere Variablen kompakt deklarieren

Im oberen Beispiel sind zwei Variablen vom gleichen Typ: `isPresident` und `isVegetarian`.

```

boolean isPresident;
boolean isVegetarian;

```

Immer dann, wenn der Variablentyp der gleiche ist, lässt sich die Deklaration verkürzen: Variablen werden mit Komma getrennt.

```
boolean isPresident, isVegetarian;
```

Variablen-deklaration mit Wertinitialisierung

Gleich bei der Deklaration lassen sich Variablen mit einem Anfangswert initialisieren. Hinter einem Gleichheitszeichen steht der Wert, der oft ein Literal ist. Ein Beispielprogramm:

¹⁴ Das ist natürlich eine Anspielung auf C, in dem Deklarationen wie `char (*(*a[2]))()[2]` möglich sind. Gut, dass es mit `cdecl` ein Programm zum »Vorlesen« solcher Definitionen gibt.

Listing 2.6: Obama.java

```
public class Obama
{
    public static void main( String[] args )
    {
        String name    = "Barack Hussein Obama II";
        int    age     = 48;
        double income = 400000;
        char   gender  = 'm';
        boolean isPresident = true;
    }
}
```

Wir haben gesehen, dass bei der Deklaration mehrerer Variablen gleichen Typs ein Komma die Bezeichner trennt. Das überträgt sich auch auf die Initialisierung. Ein Beispiel:

```
boolean sendSms = true,
       bungaBungaParty = true;
String person1 = "Silvio", person2 = "Ruby the Heart Stealer";
double x, y, bodyHeight = 183;
```

Die Zeilen deklarieren mehrere Variablen auf einen Schlag. x und y am Schluss bleiben uninitialisiert.

Zinsen berechnen als Beispiel zur Variablen-deklaration, -initialisierung & -ausgabe

Zusammen mit der Konsoleneingabe können wir schon einen einfachen Zinsrechner programmieren. Er soll uns ausgeben, wie hoch die Zinsen für ein gegebenes Kapital bei einem gegebenen Zinssatz (engl. *interest rate*) nach einer gewissen Zeit sind.

Listing 2.7: InterestRates.java

```
public class InterestRates
{
    public static void main( String[] args )
    {
        double capital      = 20000 /* Euro */;
        double interestRate = 3.6 /* Prozent */;
        double years        = 2;
```

```

        double interestRates = capital * interestRate * years / 100;
        System.out.printf( "Zinsen: " + interestRates );    // 1440.0
    }
}

```

Das obige Beispiel macht ebenfalls deutlich, dass Strings mit dem Plus aneinandergehangt werden koennen; ist ein Teil kein String, so wird er in einen String konvertiert.

2.3.3 Konsoleneingaben

Bisher haben wir zwei Methoden kennengelernt: `println()` und `random()`. Die Methode `println()` »hangt« am `System.out` bzw. `System.err`-Objekt und `random()` »hangt« am `Math`-Objekt.

Der Gegenpol zu `println()` ist eine Konsoleneingabe. Hier gibt es unterschiedliche Varianten. Die einfachste ist mit der Klasse `java.util.Scanner`. In Kapitel 4, »Der Umgang mit Zeichenketten«, wird die Klasse noch viel genauer untersucht. Es reicht aber an dieser Stelle zu wissen, wie Strings, Ganzzahlen und Fließkommazahlen eingelesen werden.

Eingabe lesen vom Typ	Anweisung
String	<code>String s = new java.util.Scanner(System.in).nextLine();</code>
int	<code>int i = new java.util.Scanner(System.in).nextInt();</code>
double	<code>double d = new java.util.Scanner(System.in).nextDouble();</code>

Tabelle 2.7: Einlesen einer Zeichenkette, Ganz- und Fließkommazahl von der Konsole

Verbinden wir die drei Moglichkeiten zu einem Beispiel. Zunachst soll der Name eingelesen werden, dann das Alter und anschliegend eine Fließkommazahl.

Listing 2.8: SmallConversation.java

```

public class SmallConversation
{
    public static void main( String[] args )
    {
        System.out.println( "Moin! Wie heit denn du?" );
        String name = new java.util.Scanner( System.in ).nextLine();
        System.out.printf( "Hallo %s. Wie alt bist du?%n", name );
    }
}

```

```

int age = new java.util.Scanner( System.in ).nextInt();
System.out.printf( "Aha, %s Jahre, das ist ja die Hälfte von %s.%n",
                   age, age * 2 );
System.out.println( "Sag mal, was ist deine Lieblingsfließkommazahl?" );
double value = new java.util.Scanner( System.in ).nextDouble();
System.out.printf( "%s? Aha, meine ist %s.%n",
                   value, Math.random() * 100000 );
}
}

```

Eine Konversation sieht somit etwa so aus:

Moin! Wie heißt denn du?
Christian
 Hallo Christian. Wie alt bist du?
37
 Aha, 37 Jahre, das ist ja die Hälfte von 74.
 Sag mal, was ist deine Lieblingsfließkommazahl?
9,7
 9.7? Aha, meine ist 60769.81705995359.

Die Eingabe der Fließkommazahl muss mit Komma erfolgen, wenn die JVM auf einem deutschsprachigen Betriebssystem läuft. Die Ausgabe über printf() kann ebenfalls lokale Fließkommazahlen schreiben, dann muss jedoch statt dem Platzhalter %s die Kennung %f oder %g verwendet werden. Das wollen wir in einem zweiten Beispiel nutzen.

Zinsberechnung mit der Benutzereingabe

Die Zinsberechnung, die vorher feste Werte im Programm hatte, soll eine Benutzereingabe bekommen. Des Weiteren erwarten wir die Dauer in Monaten statt Jahren.

Listing 2.9: MyInterestRates.java

```

public class MyInterestRates
{
    public static void main( String[] args )
    {
        System.out.println( "Kapital?" );
        double capital = new java.util.Scanner( System.in ).nextDouble();

```

```

System.out.println( "Zinssatz?" );
double interestRate = new java.util.Scanner( System.in ).nextDouble();

System.out.println( "Anlagedauer in Monaten?" );
int month = new java.util.Scanner( System.in ).nextInt();

double interestRates = capital * interestRate * month / (12*100);
System.out.printf( "Zinsen: %g%n", interestRates );
}
}

```

Die vorher fest verdrahteten Werte sind nun alle dynamisch, und wir kommen mit den Eingaben zum gleichen Ergebnis wie vorher:

```

Kapital?
20000
Zinssatz?
3,6
Anlagedauer in Monaten?
24
Zinsen: 1440,00

```

2.3.4 Fließkommazahlen mit den Datentypen float und double

Für Fließkommazahlen (auch *Gleitkommazahlen* genannt) einfacher und erhöhter Genauigkeit bietet Java die Datentypen `float` und `double`. Die Datentypen sind im *IEEE 754-Standard* beschrieben und haben eine Länge von 4 Byte für `float` und 8 Byte für `double`. Fließkommaliterale können einen Vorkomma teil und einen Nachkomma teil besitzen, die durch einen Dezimalpunkt (kein Komma) getrennt sind. Ein Fließkommaliteral muss keine Vor- oder Nachkommastellen besitzen, sodass auch Folgendes gültig ist:

```
double d = 10.0 + 20. + .11;
```



Hinweis

Der Datentyp `float` ist mit 4 Byte, also 32 Bit, ein schlechter Scherz. Der Datentyp `double` geht mit 64 Bit ja gerade noch, wobei in Hardware eigentlich 80 Bit üblich sind.

Der Datentyp float *

Standardmäßig sind die Fließkomma-Literale vom Typ `double`. Ein nachgestelltes »`f`« (oder »`F`«) zeigt an, dass es sich um ein `float` handelt.

Beispiel

zB

Gültige Zuweisungen für Fließkommazahlen vom Typ `double` und `float`:

```
double pi = 3.1415, delta = .001;
float ratio = 4.33F;
```

Auch für den Datentyp `double` lässt sich ein »`d`« (oder »`D`«) nachstellen, was allerdings nicht nötig ist, wenn Literale für Kommazahlen im Quellcode stehen; Zahlen wie `3.1415` sind automatisch vom Typ `double`. Während jedoch bei `1 + 2 + 4.0` erst `1` und `2` als `int` addiert werden, dann in `double` und anschließend auf `4.0` addiert werden, würde `1D + 2 + 4.0` gleich mit der Fließkommazahl `1` beginnen. So ist auch `1D` gleich `1.` bzw. `1.0`.

Frage

!

Was ist das Ergebnis der Ausgabe?

```
System.out.println( 20000000000F == 20000000000F+1 );
System.out.println( 20000000000D == 20000000000D+1 );
```

Tipp: Was sind die Wertebereiche von `float` und `double`?

Noch genauere Auflösung bei Fließkommazahlen *

Einen höher auflösenden beziehungsweise präziseren Datentyp für Fließkommazahlen als `double` gibt es nicht. Die Standardbibliothek bietet für diese Aufgabe in `java.math` die Klasse `BigDecimal` an, die in Kapitel 18, »Bits und Bytes und Mathematisches«, näher beschrieben ist. Das ist sinnvoll für Daten, die eine sehr gute Genauigkeit aufweisen sollen, wie zum Beispiel Währungen.¹⁵

¹⁵ Einige Programmiersprachen besitzen für Währungen eingebaute Datentypen, wie LotusScript mit `Currency`, das mit 8 Byte einen sehr großen und genauen Wertebereich abdeckt. Erstaunlicherweise gab es einmal in C# den Datentyp `currency` für ganzzahlige Währungen.



Sprachvergleich

In C# gibt es den Datentyp `decimal`, der mit 128 Bit (also 16 Byte) auch genügend Präzision bietet, um eine Zahl wie 0,00000000000000000000000000000001 auszudrücken.

2.3.5 Ganzzahlige Datentypen

Java stellt fünf ganzzahlige Datentypen zur Verfügung: `byte`, `short`, `char`, `int` und `long`. Die feste Länge von jeweils 1, 2, 2, 4 und 8 Byte ist eine wesentliche Eigenschaft von Java. Ganzzahlige Typen sind in Java immer vorzeichenbehaftet (mit der Ausnahme von `char`); einen Modifizierer `unsigned` wie in C(++) gibt es nicht.¹⁶ Negative Zahlen werden durch Voranstellen eines Minuszeichens gebildet. Ein Pluszeichen für positive Zeichen ist möglich. `int` und `long` sind die bevorzugten Typen. `byte` kommt selten vor und `short` nur in wirklich sehr seltenen Fällen, etwa bei Feldern mit Bilddaten.

Ganzzahlen sind standardmäßig vom Typ int

Betrachten wir folgende Zeile, so ist auf den ersten Blick kein Fehler zu erkennen:

```
System.out.println( 123456789012345 ); // ☹
```

Dennoch übersetzt der Compiler die Zeile nicht, da er ein Ganzzahlliteral ohne explizite Größenangabe als 32 Bit langes `int` annimmt. Die obige Zeile führt daher zu einem Compilerfehler, da unsere Zahl nicht im Wertebereich von $-2.147.483.648 \dots +2.147.483.647$ liegt, sondern weit außerhalb: $2147483647 < 123456789012345$. Java reserviert also *nicht* so viele Bits wie benötigt und wählt nicht automatisch den passenden Wertebereich.

Wer wird mehrfacher Milliardär? Der Datentyp long

Der Compiler betrachtet jede Ganzzahl automatisch als `int`. Sollte der Wertebereich von etwa plus/minus zwei Milliarden nicht reichen, greifen Entwickler zum nächsthöheren Datentyp. Dass eine Zahl `long` ist, muss ausdrücklich angegeben werden. Dazu wird an das Ende von Ganzzahlliteralen vom Typ `long` ein »`l`« oder »`L`« gesetzt. Um die Zahl `123456789012345` gültig ausgeben zu lassen, ist Folgendes zu schreiben:

```
System.out.println( 123456789012345L );
```

¹⁶ In Java bilden `long` und `short` einen eigenen Datentyp. Sie dienen nicht wie in C(++) als Modifizierer. Eine Deklaration wie `long int i` ist also genauso falsch wie `long long time_ago`.

Tipp

Das kleine »l« hat sehr viel Ähnlichkeit mit der Ziffer Eins. Daher sollte bei Längenangaben immer ein großes »L« eingefügt werden.

**Frage**

Was gibt die folgende Anweisung aus?

```
System.out.println( 123456789 + 54321 );
```

Der Datentyp byte

Ein byte ist ein Datentyp mit einem Wertebereich von -128 bis +127. Eine Initialisierung wie

```
byte b = 200; // ☹
```

ist also nicht erlaubt, da $200 > 127$ ist. Somit fallen alle Zahlen von 128 bis 255 (hexadezimal $80_{16} - FF_{16}$) raus. In der Datenverarbeitung ist das Java-byte, weil es ein Vorzeichen trägt, nur mittelprächtig brauchbar, da insbesondere in der Dateiverarbeitung Wertebereiche von 0 bis 255 gewünscht sind.

Java erlaubt zwar keine vorzeichenlosen Ganzzahlen, aber mit zwei Schreibweisen lassen sich doch Zahlen wie 200 in einem byte speichern.

```
byte b = (byte) 200;
```

Der Java-Compiler nimmt dazu einfach die Bitbelegung von 200 und interpretiert das oberste dann gesetzte Bit als Vorzeichenbit. Bei der Ausgabe fällt das auf:

```
byte b = (byte) 200;
System.out.println( b ); // -56
```

Der Datentyp short *

Der Datentyp short ist selten anzutreffen. Mit seinen 2 Byte kann er einen Wertebereich von -32.768 bis +32.767 darstellen. Das Vorzeichen »kostet« wie bei den anderen Ganzzahlen 1 Bit, sodass nicht 16 Bit, sondern nur 15 Bit für Zahlen zu Verfügung stehen. Allerdings gilt wie beim byte, dass auch ein short ohne Vorzeichen auf zwei Arten initialisiert werden kann:

```
short s = (short) 33000;
System.out.println( s );      // -32536
```

2.3.6 Wahrheitswerte

Der Datentyp `boolean` beschreibt einen Wahrheitswert, der entweder `true` oder `false` ist. Die Zeichenketten `true` und `false` sind reservierte Wörter und bilden neben konstanten Strings und primitiven Datentypen Literale. Kein anderer Wert ist für Wahrheitswerte möglich, insbesondere werden numerische Werte nicht als Wahrheitswerte interpretiert.

Der boolesche Typ wird beispielsweise bei Bedingungen, Verzweigungen oder Schleifen benötigt. In der Regel ergibt sich ein Wahrheitswert aus Vergleichen.

2.3.7 Unterstriche in Zahlen *

Um eine Anzahl von Millisekunden in Tage zu konvertieren, muss einfach eine Division vorgenommen werden. Um Millisekunden in Sekunden umzurechnen, brauchen wir eine Division durch 1000, von Sekunden auf Minuten eine Division durch 60, von Minuten auf Stunden eine Division durch 60, und die Stunden auf Tage bringt die letzte Division durch 24. Schreiben wir das auf:

```
long millis = 10 * 24*60*60*1000L;
long days = millis / 86400000L;
System.out.println( days );    // 10
```

Eine Sache fällt bei der Zahl 86400000 auf: Besonders gut lesbar ist sie nicht. Die eine Lösung ist, es erst gar nicht zu so einer Zahl kommen zu lassen und sie wie in der ersten Zeile durch eine Reihe von Multiplikationen aufzubauen – mehr Laufzeit kostet das nicht, da dieser konstante Ausdruck zur Übersetzungszeit feststeht.

Die zweite Variante ist eine neue Schreibweise, die Java 7 einführt: *Unterstriche in Zahlen*. Anstatt ein numerisches Literal als 86400000 zu schreiben, ist in Java 7 auch Folgendes erlaubt:

```
long millis = 10 * 86_400_000L;
long days = millis / 86_400_000L;
System.out.println( days );    // 10
```

Die Unterstriche machen die 1000er-Blöcke gut sichtbar. Hilfreich ist die Schreibweise auch bei Literalen in Binär- und Hexdarstellung, da die Unterstriche hier ebenfalls Blöcke absetzen können.¹⁷

Beispiel

zB

Unterstriche verdeutlichen Blöcke bei Binär- und Hexadezimalzahlen.

```
int i = 0b01101001_01001101_11100101_01011110;
long l = 0x7fff_ffff_ffff_ffffL;
```

Der Unterstrich darf in jedem Literal stehen, zwei aufeinanderfolgende Unterstriche sind aber nicht erlaubt.

2.3.8 Alphanumerische Zeichen

Der alphanumerische Datentyp `char` (von engl. *character*, Zeichen) ist 2 Byte groß und nimmt ein Unicode-Zeichen auf. Ein `char` ist nicht vorzeichenbehaftet. Die Literale für Zeichen werden in einfache Hochkommata gesetzt. Sprachneulinge verwechseln häufig die einfachen Hochkommata mit den Anführungszeichen der Zeichenketten (Strings). Die einfache Merkregel lautet: ein Zeichen – ein Hochkomma, mehrere Zeichen – zwei Hochkommata (Gänsefüßchen).

Beispiel

zB

Korrekte Hochkommata für Zeichen und Zeichenketten:

```
char c = 'a';
String s = "Heut' schon gebeckert?";
```

Da der Compiler ein `char` automatisch in ein `int` konvertieren kann, ist auch `int c = 'a';` gültig.

¹⁷ Bei Umrechnungen zwischen Stunden, Minuten und so weiter hilft auch die Klasse `TimeUnit` mit einigen statischen `toXXX()`-Methoden.

2.3.9 Gute Namen, schlechte Namen

Für die optimale Lesbarkeit und Verständlichkeit eines Programmcodes sollten Entwickler beim Schreiben einige Punkte berücksichtigen:

- **Ein konsistentes Namensschema ist wichtig.** Heißt ein Zähler no, nr, cnr oder counter? Auch sollten wir korrekt schreiben und auf Rechtschreibfehler achten, denn leicht wird aus necessaryConnection dann nesesarryConnection. Variablen ähnlicher Schreibweise, etwa counter und counters, sind zu vermeiden.
- **Abstrakte Bezeichner sind ebenfalls zu vermeiden.** Die Deklaration int TEN = 10; ist absurd. Eine unsinnige Idee ist auch die folgende: boolean FALSE = true, TRUE = false;. Im Programmcode würde dann mit FALSE und TRUE gearbeitet. Einer der obersten Plätze bei einem Wettbewerb für die verpfuschesten Java-Programme wäre uns gewiss.
- **Unicode-Sequenzen können zwar in Bezeichnern aufgenommen werden, doch sollten sie vermieden werden.** In double übelkübel, \u00FCbelk\u00FCbel; sind beide Bezeichnernamen gleich, und der Compiler meldet einen Fehler.
- **0 und O und 1 und l sind leicht zu verwechseln.** Die Kombination »rn« ist schwer zu lesen und je nach Zeichensatz leicht mit »m« zu verwechseln.¹⁸ Gültig – aber böse – ist auch: int ínt, ìnt, ínt; boolean bôoleañ;



Bemerkung

In China gibt es 90 Millionen Familien mit dem Nachnamen Li. Das wäre so, als ob wir jede Variable temp1, temp2 ... nennen würden.



Ist ein Bezeichnername unglücklich gewählt (pneumonoultramicroscopicsilicovolcanoconiosis ist schon etwas lang), so lässt er sich problemlos konsistent umbenennen. Dazu wählen wir im Menü REFACTOR • RENAME – oder auch kurz **Alt + ⌘ + R**; der Cursor muss auf dem Bezeichner stehen. Eine optionale Vorschau (engl. *preview*) zeigt an, welche Änderungen die Umbenennung nach sich ziehen wird. Neben RENAME gibt es auch noch eine andere Möglichkeit. Dazu lässt sich auf der Variablen mit **Strg + 1** ein Popup-Fenster mit LOCAL RENAME öffnen. Der Bezeichner wird selektiert und lässt sich ändern. Gleichzeitig ändern sich alle Bezüge auf die Variable mit.

¹⁸ Eine Software wie Mathematica warnt vor Variablen mit fast identischem Namen.

2.3.10 Initialisierung von lokalen Variablen

Die Laufzeitumgebung – beziehungsweise der Compiler – initialisiert lokale Variablen *nicht* automatisch mit einem Nullwert bzw. Wahrheitsvarianten nicht mit false. Vor dem Lesen müssen lokale Variablen von Hand initialisiert werden, andernfalls gibt der Compiler eine Fehlermeldung aus.¹⁹

Im folgenden Beispiel seien die beiden lokalen Variablen age und adult nicht automatisch initialisiert, und so kommt es bei der versuchten Ausgabe von age zu einem Compilerfehler. Der Grund ist, dass ein Lesezugriff nötig ist, aber vorher noch kein Schreibzugriff stattfand.

```
int      age;
boolean adult;
System.out.println( age );    // ☹ Local variable age may not
                             // have been initialized.

age = 18;
if ( age >= 18 )           // Fallunterscheidung: wenn-dann
    adult = true;
System.out.println( adult ); // ☹ Local variable adult may not
                           // have been initialized.
```

Weil Zuweisungen in bedingten Anweisungen vielleicht nicht ausgeführt werden, meldet der Compiler auch bei System.out.println(adult) einen Fehler, da er analysiert, dass es einen Programmfluss ohne die Zuweisung gibt. Da adult nur nach der if-Abfrage auf den Wert true gesetzt wird, wäre nur unter der Bedingung, dass age größer gleich 18 ist, ein Schreibzugriff auf adult erfolgt und ein folgender Lesezugriff möglich. Doch da der Compiler annimmt, dass es andere Fälle geben kann, wäre ein Zugriff auf eine nicht initialisierte Variable ein Fehler.

Eclipse zeigt einen Hinweis und einen Verbesserungsvorschlag an, wenn eine lokale Variable nicht initialisiert ist.



¹⁹ Anders ist das bei Objektvariablen (und statischen Variablen sowie Feldern). Sie sind standardmäßig mit null (Referenzen), 0 (bei Zahlen) oder false belegt.

2.4 Ausdrücke, Operanden und Operatoren

Beginnen wir mit mathematischen Ausdrücken, um dann die Schreibweise in Java zu ermitteln. Eine mathematische Formel, etwa der Ausdruck $-27 * 9$, besteht aus *Operanden* (engl. *operands*) und *Operatoren* (engl. *operators*). Ein Operand ist eine Variable oder ein Literal. Im Fall einer Variablen wird der Wert aus der Variablen ausgelesen und mit ihm die Berechnung durchgeführt.

Die Arten von Operatoren

Operatoren verknüpfen die Operanden. Je nach Anzahl der Operanden unterscheiden wir:

- Ist ein Operator auf genau einem Operanden definiert, so nennt er sich *unärer Operator* (oder *einstelliger Operator*). Das Minus (negatives Vorzeichen) vor einem Operand ist ein unärer Operator, da er für genau den folgenden Operanden gilt.
- Die üblichen Operatoren Plus, Minus, Mal und Geteilt sind *binäre (zweistellige) Operatoren*.
- Es gibt auch einen Fragezeichen-Operator für bedingte Ausdrücke, der dreistellig ist.

Operatoren erlauben die Verbindung einzelner Ausdrücke zu neuen Ausdrücken. Einige Operatoren sind aus der Schule bekannt, wie Addition, Vergleich, Zuweisung und weitere. C(++)-Programmierer werden viele Freunde wiedererkennen.

2.4.1 Zuweisungsoperator

In Java dient das Gleichheitszeichen = der *Zuweisung* (engl. *assignment*).²⁰ Der Zuweisungsoperator ist ein binärer Operator, bei dem auf der linken Seite die zu belegende Variable steht und auf der rechten Seite ein Ausdruck.

zB Beispiel

Ein Ausdruck mit Zuweisungen:

```
int i = 12, j;
j = i * 2;
```

²⁰ Die Zuweisungen sehen zwar so aus wie mathematische Gleichungen, doch existiert ein wichtiger Unterschied: Die Formel $a = a + 1$ ist – zumindest im Dezimalsystem ohne zusätzliche Algebra – mathematisch nicht zu erfüllen, da es kein a geben kann, das $a = a + 1$ erfüllt. Aus Programmiersicht ist es in Ordnung, da die Variable a um eins erhöht wird.

zB

2

Beispiel (Forts.)

Die Multiplikation berechnet das Produkt von 12 und 2 und speichert das Ergebnis in j ab. Von allen primitiven Variablen, die in dem Ausdruck vorkommen, wird also der Wert ausgelesen und in den Ausdruck eingesetzt.²¹ Dies nennt sich auch *Wertoperation*, da der Wert der Variablen betrachtet wird und nicht ihr Speicherort oder gar ihr Variablenname.

Erst nach dem Auswerten des Ausdrucks kopiert der Zuweisungsoperator das Ergebnis in die Variable. Gibt es Laufzeitfehler, etwa durch eine Division durch null, gibt es keinen Schreibzugriff auf die Variable.

Zuweisungen sind auch Ausdrücke

Zwar finden sich Zuweisungen oft als Ausdrucksanweisung wieder, doch können sie an jeder Stelle stehen, an der ein Ausdruck erlaubt ist, etwa in einem Methodenaufruf wie `print()`:

```
int a = 1;           // Deklaration mit Initialisierung
a = 2;             // Anweisung mit Zuweisung
System.out.println( a = 3 ); // Ausdruck mit Zuweisung. Liefert 3.
```

!

Sprachenvergleich

Das einfache Gleichheitszeichen = dient in Java nur der Zuweisung. In anderen Programmiersprachen wird die Zuweisung durch ein anderes Symbol deutlich gemacht, etwa wie in Pascal mit :=. Um Zuweisungen von Vergleichen trennen zu können, definiert Java hier der C(++)-Tradition folgend einen binären Vergleichsoperator ==. Der Vergleichsoperator liefert immer den Ergebnistyp boolean:

```
int baba = 1;
System.out.println( baba == 1 ); // »true«: Ausdruck mit Vergleich
System.out.println( baba = 2 ); // »2«: Ausdruck mit Zuweisung
```

²¹ Es gibt Programmiersprachen, in denen Wertoperationen besonders gekennzeichnet werden. So etwa in LOGO. Eine Wertoperation schreibt sich dort mit einem Doppelpunkt vor der Variablen, etwa :X + :Y.

Mehrere Zuweisungen in einem Schritt

Zuweisungen der Form `a = b = c = 0;` sind erlaubt und gleichbedeutend mit den drei Anweisungen `c = 0; b = c; a = b;`. Die explizite Klammerung `a = (b = (c = 0))` macht noch einmal deutlich, dass sich Zuweisungen verschachteln lassen und Zuweisungen wie `c = 0` Ausdrücke sind, die einen Wert liefern. Doch auch dann, wenn wir meinen, dass

```
a = (b = c + d) + e;
```

eine coole Vereinfachung im Vergleich zu

```
b = c + d;
```

```
a = b + e;
```

ist, sollten wir mit einer Zuweisung pro Zeile auskommen.

Die Reihenfolge der Auswertung zeigt anschaulich folgendes Beispiel:

```
int b = 10;
System.out.println( (b = 20) * b );      // 400
System.out.println( b );                  // 20
```

2.4.2 Arithmetische Operatoren

Ein arithmetischer Operator verknüpft die Operanden mit den Operatoren Addition (+), Subtraktion (-), Multiplikation (*) und Division (/). Zusätzlich gibt es den Restwert-Operator (%), der den bei der Division verbleibenden Rest betrachtet. Alle Operatoren sind für ganzzahlige Werte sowie für Fließkommazahlen definiert. Die arithmetischen Operatoren sind binär, und auf der linken und rechten Seite sind die Typen numerisch. Der Ergebnistyp ist ebenfalls numerisch.

Numerische Umwandlung

Bei Ausdrücken mit unterschiedlichen numerischen Datentypen, etwa `int` und `double`, bringt der Compiler vor der Anwendung der Operation alle Operanden auf den umfassenderen Typ. Vor der Auswertung von `1 + 2.0` wird somit die Ganzzahl 1 in ein `double` konvertiert und dann die Addition vorgenommen – das Ergebnis ist auch vom Typ `double`. Das nennt sich *numerische Umwandlung* (engl. *numeric promotion*). Bei `byte` und `short` gilt die Sonderregelung, dass sie vorher in `int` konvertiert werden.²² (Auch im Java-

²² http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#26917

Bytecode gibt es keine arithmetischen Operationen auf byte, short und char.) Anschließend wird die Operation ausgeführt, und der Ergebnistyp entspricht dem umfassenden Typ.

Der Divisionsoperator

Der binäre Operator »/« bildet den Quotienten aus Dividend und Divisor. Auf der linken Seite steht der Dividend und auf der rechten der Divisor. Die Division ist für Ganzzahlen und für Fließkommazahlen definiert. Bei der Ganzzahldivision wird zu null hin gerundet, und das Ergebnis ist keine Fließkommazahl, sodass $1/3$ das Ergebnis 0 ergibt und nicht 0,333... Den Datentyp des Ergebnisses bestimmen die Operanden und nicht der Operator. Soll das Ergebnis vom Typ double sein, muss ein Operand ebenfalls double sein.

```
System.out.println( 1.0 / 3 );           // 0.3333333333333333  
System.out.println( 1 / 3.0 );          // 0.3333333333333333  
System.out.println( 1 / 3 );            // 0
```

Strafe bei Division durch null

Schon die Schulmathematik lehrte uns, dass die Division durch null nicht erlaubt ist. Führen wir in Java eine Ganzzahldivision mit dem Divisor 0 durch, so bestraft uns Java mit einer ArithmeticException, die, wenn sie nicht behandelt würde, zum Ende des Programmablaufs führt. Bei Fließkommazahlen liefert eine Division durch 0 keine Ausnahme, sondern $+-\infty$ und bei $0.0/0.0$ den Sonderwert NaN (mehr dazu folgt in Kapitel 18, »Bits und Bytes und Mathematisches«). Ein NaN steht für *Not a Number* (auch schon manchmal »Unzahl« genannt) und wird vom Prozessor erzeugt, falls er eine mathematische Operation wie die Division durch null nicht durchführen kann. In Kapitel 12 werden wir auf NaN noch einmal zurückkommen.

Anekdoten

Auf dem Lenkraketenkreuzer USS Yorktown gab ein Mannschaftsmitglied aus Versehen die Zahl Null ein. Das führte zu einer Division durch null, und der Fehler pflanzte sich so weit fort, dass die Software abstürzte und das Antriebssystem stoppte. Das Schiff trieb mehrere Stunden antriebslos im Wasser.



Der Restwert-Operator % *

Eine Ganzzahldivision muss nicht unbedingt glatt aufgehen, wie im Fall von 9/2. In diesem Fall gibt es den Rest 1. Diesen Rest liefert der *Restwert-Operator* (engl. *remainder operator*), oft auch *Modulo* genannt. Mathematiker unterscheiden die beiden Begriffe *Rest* und *Modulo*, da ein Modulo nicht negativ ist, der Rest in Java aber schon. Das soll uns aber egal sein.

```
System.out.println( 9% 2 );           // 1
```

Der Restwert-Operator ist auch auf Fließkommazahlen anwendbar, und die Operanden können negativ sein.

```
System.out.println( 12.0% 2.5 );      // 2.0
```

Die Division und der Restwert richten sich in Java nach einer einfachen Formel:

$$(int)(a/b) \cdot b + (a \% b) = a$$

zB Beispiel

Die Gleichung ist erfüllt, wenn wir etwa $a = 10$ und $b = 3$ wählen. Es gilt: $(int)(10/3) = 3$ und $10 \% 3$ ergibt 1. Dann ergeben $3 * 3 + 1 = 10$.

Aus dieser Gleichung folgt, dass beim Restwert das Ergebnis nur dann negativ ist, wenn der Dividend negativ ist; er ist nur dann positiv, wenn der Dividend positiv ist. Es ist leicht einzusehen, dass das Ergebnis der Restwert-Operation immer echt kleiner ist als der Wert des Divisors. Wir haben den gleichen Fall wie bei der Ganzzahldivision, dass ein Divisor mit dem Wert 0 eine `ArithmaticException` auslöst und bei Fließkommazahlen zum Ergebnis `NaN` führt.

Listing 2.10: RemainderAndDivDemo.java, main()

```
System.out.println( "+5% +3 = " + (+5% +3) );    // 2
System.out.println( "+5 / +3 = " + (+5 / +3) );  // 1

System.out.println( "+5% -3 = " + (+5% -3) );    // 2
System.out.println( "+5 / -3 = " + (+5 / -3) );  // -1

System.out.println( "-5% +3 = " + (-5% +3) );   // -2
System.out.println( "-5 / +3 = " + (-5 / +3) ); // -1
```

```
System.out.println( "-5% -3 = " + (-5% -3) ); // -2
System.out.println( "-5 / -3 = " + (-5 / -3) ); // 1
```

Gewöhnungsbedürftig ist die Tatsache, dass der erste Operand (Dividend) das Vorzeichen des Restes definiert und niemals der zweite (Divisor).

Hinweis

Um mit `value % 2 == 1` zu testen, ob `value` eine ungerade Zahl ist, muss `value` positiv sein, denn $-3 \% 2$ wertet Java zu -1 aus. Der Test auf ungerade Zahlen wird erst wieder korrekt mit `value % 2 != 0`.



Restwert für Fließkommazahlen und `Math.IEEEremainder()` *

Über die oben genannte Formel können wir auch bei Fließkommazahlen das Ergebnis einer Restwert-Operation leicht berechnen. Dabei muss beachtet werden, dass sich der Operator nicht so wie unter IEEE 754 verhält. Denn diese Norm schreibt vor, dass die Restwert-Operation den Rest von einer runden Division berechnet und nicht von einer abschneidenden. So wäre das Verhalten nicht analog zum Restwert bei Ganzzahlen. Java definiert den Restwert jedoch bei Fließkommazahlen genauso wie den Restwert bei Ganzzahlen. Wünschen wir ein Restwert-Verhalten, wie IEEE 754 es vorschreibt, so können wir immer noch die statische Bibliotheksmethode `Math.IEEEremainder()`²³ verwenden.

Auch bei der Restwert-Operation bei Fließkommazahlen werden wir niemals eine Exception erwarten. Eventuelle Fehler werden, wie im IEEE-Standard beschrieben, mit NaN angegeben. Ein Überlauf oder Unterlauf kann zwar vorkommen, aber nicht geprüft werden.

Rundungsfehler *

Prinzipiell sollten Anweisungen wie $1.1 - 0.1$ immer 1.0 ergeben, jedoch treten interne Rundungsfehler bei der Darstellung auf und lassen das Ergebnis von Berechnung zu Berechnung immer ungenauer werden. Ein besonders ungünstiger Fehler trat 1994 beim Pentium-Prozessor im Divisionsalgorithmus Radix-4 SRT auf, ohne dass der Programmierer der Schuldige war:

²³ Es gibt auch Methoden, die nicht mit Kleinbuchstaben beginnen, wobei das sehr selten ist und nur in Sonderfällen auftritt. `ieeeRemainder()` sah für die Autoren nicht nett aus.

```

double x, y, z;
x = 4195835.0;
y = 3145727.0;
z = x - (x/y) * y;
System.out.println( z );

```

Ein fehlerhafter Prozessor liefert hier 256, obwohl laut Rechenregel das Ergebnis 0 sein muss. Laut Intel sollte für einen normalen Benutzer (Spieler, Softwareentwickler, Surfer?) der Fehler nur alle 27.000 Jahre auftauchen. Glück für die meisten. Eine Studie von IBM errechnete eine Fehlerhäufigkeit von einmal in 24 Tagen. Alles in allem hat Intel die CPUs zurückgenommen, über 400 Millionen US-Dollar verloren und spät den Kopf gerade noch aus der Schlinge gezogen.

Die meisten Rundungsfehler resultieren aber daher, dass endliche Dezimalbrüche im Rechner als Näherungswerte für periodische Binärbrüche repräsentiert werden müssen. 0.1 entspricht einer periodischen Mantisse im IEEE-Format.

2.4.3 Unäres Minus und Plus

Die binären Operatoren sitzen zwischen zwei Operanden, während sich ein unärer Operator genau einen Operanden vornimmt. Das unäre Minus (Operator zur Vorzeichenumkehr) etwa dreht das Vorzeichen des Operanden um. So wird aus einem positiven Wert ein negativer und aus einem negativen Wert ein positiver.

zB

Beispiel

Drehe das Vorzeichen einer Zahl um:

a = -a;

Alternativ ist:

a = -1 * a;

Das unäre Plus ist eigentlich unnötig; die Entwickler haben es jedoch aus Symmetriegründen mit eingeführt.

zB

2

Beispiel

Minus und Plus sitzen direkt vor dem Operanden, und der Compiler weiß selbstständig, ob dies unär oder binär ist. Der Compiler erkennt auch folgende Konstruktion:

```
int i = - - - 2 + - + 3;
```

Dies ergibt den Wert -5. Einen Ausdruck wie ---2+-+3 erkennt der Compiler dagegen nicht an, da die zusammenhängenden Minuszeichen als Inkrement interpretiert werden und nicht als unärer Operator. Das Leerzeichen ist also bedeutend.

2.4.4 Zuweisung mit Operation

In Java lassen sich Zuweisungen mit numerischen Operatoren kombinieren. Für einen binären Operator (symbolisch # genannt) im Ausdruck $a = a \# (b)$ kürzt der *Verbundoperator* den Ausdruck zu $a \#= b$ ab. Dazu einige Beispiele:

Schreibweise mit Verbundoperator	Ausführliche Schreibweise
<code>a += 2;</code>	<code>a = a + 2;</code>
<code>a *= -1;</code>	<code>a = a * -1;</code>
<code>a /= 10;</code>	<code>a = a / 10;</code>

Tabelle 2.8: Verbundoperator und ausgeschriebene Variante

Dass eine Zuweisung immer auch ein Ausdruck ist, zeigt folgendes Beispiel:

```
int a = 0;
System.out.println( a );      // 0
System.out.println( a += 2 );  // 2
System.out.println( a );      // 2
```

Besondere Obacht sollten wir auf die automatische Klammerung geben. Bei einem Ausdruck wie $a *= 3 + 5$ gilt $a = a * (3 + 5)$ und nicht selbstverständlich die Punkt-vor-Strich-Regelung $a = a * 3 + 5$.

Falls es sich bei der rechten Seite um einen komplexeren Ausdruck handelt, wird dieser nur einmal ausgewertet. Dies ist wichtig bei Methodenaufrufen, die Nebenwirkungen besitzen, also etwa Zustände wie einen Zähler verändern.

zB Beispiel

Wir profitieren auch bei Feldzugriffen (siehe Abschnitt 3.8, »Arrays«) von Verbundoperationen, da die Auswertung des Index nur einmal stattfindet:

```
array[ 2 * i + j ] = array[ 2 * i + j ] + 1;
```

Leichter zu lesen ist die folgende Anweisung:

```
array[ 2 * i + j ] += 1;
```

2.4.5 Präfix- oder Postfix-Inkrement und -Dekrement

Das Herauf- und Heruntersetzen von Variablen ist eine sehr häufige Operation, wofür die Entwickler in der Vorgängersprache C auch einen Operator spendiert hatten. Die praktischen Operatoren ++ und -- kürzen die Programmzeilen zum Inkrement und Dekrement ab:

```
i++;           // Abkürzung für i = i + 1
j--;           //           j = j - 1
```

Eine lokale Variable muss allerdings vorher initialisiert sein, da ein Lesezugriff vor einem Schreibzugriff stattfindet. Der ++/---Operator erfüllt also zwei Aufgaben: Neben der Wertrückgabe gibt es eine Veränderung der Variablen.

Vorher oder nachher?

Die beiden Operatoren liefern einen Ausdruck und geben daher einen Wert zurück. Es macht jedoch einen feinen Unterschied, wo dieser Operator platziert wird. Es gibt ihn nämlich in zwei Varianten: Er kann vor der Variablen stehen, wie in ++i (Präfix-Schreibweise), oder dahinter, wie bei i++ (Postfix-Schreibweise). Der Präfix-Operator verändert die Variable vor der Auswertung des Ausdrucks, und der Postfix-Operator ändert sie nach der Auswertung des Ausdrucks. Mit anderen Worten: Nutzen wir einen Präfix-Operator, so wird die Variable erst herauf- beziehungsweise heruntergesetzt und dann der Wert geliefert.

zB Beispiel

Präfix/Postfix in einer Ausgabeanweisung:

zB

2

Beispiel (Forts.)**Präfix-Inkrement und -Dekrement**

```
int i = 10, j = 20;
System.out.println( ++i ); // 11
System.out.println( --j ); // 19
System.out.println( i ); // 11
System.out.println( j ); // 19
```

Postfix-Inkrement und -Dekrement

```
int i = 10, j = 20;
System.out.println( i++ ); // 10
System.out.println( j-- ); // 20
System.out.println( i ); // 11
System.out.println( j ); // 19
```

Mit der Möglichkeit, Variablen zu erhöhen und zu vermindern, ergeben sich vier Varianten:

	Präfix	Postfix
Inkrement	Prä-Inkrement, <code>++i</code>	Post-Inkrement, <code>i++</code>
Dekrement	Prä-Dekrement, <code>--i</code>	Post-Dekrement, <code>i--</code>

Tabelle 2.9: Präfix- und Postfix-Inkrement und -Dekrement

Hinweis

In Java sind Inkrement (`++`) und Dekrement (`--`) für alle numerischen Datentypen erlaubt, also auch für Fließkommazahlen:

```
double d = 12;
System.out.println( --d ); // 11.0
double e = 12.456;
System.out.println( --e ); // 11.456
```

**Einige Kuriositäten ***

Wir wollen uns abschließend noch mit einer Besonderheit des Post-Inkrement und Prä-Inkrement beschäftigen, die nicht nachahmenswert ist:

```
a = 2;
a = ++a; // a = 3
b = 2;
b = b++; // b = 2
```

Im ersten Fall bekommen wir den Wert 3 und im zweiten Fall den Wert 2. Der erste Fall überrascht nicht, denn `a = ++a` erhöht den Wert 2 um 1, und anschließend wird 3 der Variablen `a` zugewiesen. Bei `b` ist es raffinierter: Der Wert von `b` ist 2, und dieser Wert wird intern vermerkt. Anschließend erhöht `b++` die Variable `b`. Doch die Zuweisung setzt `b` auf den gemerkten Wert, der 2 war. Also ist `b = 2`.



Hinweis

Das Post-Inkrement finden wir auch im Namen der Programmiersprache C++. Es soll ausdrücken, dass es »C-mit-eins-drauf« ist, also ein verbessertes C. Mit dem Wissen über den Postfix-Operator ist klar, dass diese Erhöhung aber erst nach der Nutzung auftritt – also ist C++ auch nur C, und der Vorteil kommt später. Einer der Entwickler von Java, Bill Joy, hat einmal Java als C++- beschrieben. Er meinte damit C++ ohne die schwer zu pflegenden Eigenschaften.

2.4.6 Die relationalen Operatoren und die Gleichheitsoperatoren

Relationale Operatoren sind *Vergleichsoperatoren*, die Ausdrücke miteinander verglichen und einen Wahrheitswert vom Typ `boolean` ergeben. Die von Java für numerische Vergleiche zur Verfügung gestellten Operatoren sind:

- Größer (`>`)
- Kleiner (`<`)
- Größer-gleich (`>=`)
- Kleiner-gleich (`<=`)

Weiterhin gibt es einen Spezial-Operator `instanceof` zum Testen von Referenzeigenschaften.

Zudem kommen zwei Vergleichsoperatoren hinzu, die Java als *Gleichheitsoperatoren* bezeichnet:

- Test auf Gleichheit (`==`)
- Test auf Ungleichheit (`!=`)

Dass Java hier einen Unterschied zwischen Gleichheitsoperatoren und Vergleichsoperatoren macht, liegt an einem etwas anderen Vorrang, der uns aber nicht weiter beschäftigen soll.

Ebenso wie arithmetische Operatoren passen die relationalen Operatoren ihre Operanden an einen gemeinsamen Typ an. Handelt es sich bei den Typen um Referenztypen, so sind nur die Vergleichsoperatoren `==` und `!=` erlaubt.

Kaum Verwechslungsprobleme durch `==` und `=`

Die Verwendung des relationalen Operators `==` und der Zuweisung `=` führt bei Einsteigern oft zu Problemen, da die Mathematik für Vergleiche und Zuweisungen immer nur ein Gleichheitszeichen kennt. Glücklicherweise ist das Problem in Java nicht so drastisch wie beispielsweise in C++, da die Typen der Operatoren unterschiedlich sind. Der Vergleichsoperator ergibt immer nur den Rückgabewert `boolean`. Zuweisungen von numerischen Typen ergeben jedoch wieder einen numerischen Typ. Es kann also kein Problem wie das folgende geben:

```
int a = 10, b = 11;
boolean result1 = ( a = b );           // ☹ Compilerfehler
boolean result2 = ( a == b );
```

Beispiel

zB

Die Wahrheitsvariable `hasSign` soll dann `true` sein, wenn das Zeichen `sign` gleich dem Minus ist:

```
boolean hasSign = (sign == '-');
```

Die Auswertungsreihenfolge ist folgende: Erst wird das Ergebnis des Vergleichs berechnet, und dieser Wahrheitswert wird anschließend in `hasSign` kopiert.

(Anti-)Stil

!

Bei einem Vergleich mit `==` können beide Operanden vertauscht werden – wenn die beiden Seiten keine beeinflussenden Seiteneffekte produzieren, also etwa Zustände ändern. Am Ergebnis ändert sich nichts, denn der Vergleichsoperator ist kommutativ. So sind

```
if ( worldExpoShanghaiCostInUSD == 58000000000 )
und
if ( 58000000000 == worldExpoShanghaiCostInUSD )
```



(Anti-)Stil (Forts.)

semantisch gleich. Bei einem Gleichheitsvergleich zwischen Variable und Literal werden viele Entwickler mit einer Vergangenheit in der Programmiersprache C die Konstanten links und die Variable rechts setzen. Der Grund für diesen sogenannten *Yoda-Stil*²⁴ ist die Vermeidung von Fehlern. Fehlt in C ein Gleichheitszeichen, so ist `if(worldExpoShanghaiCostInUSD = 58000000000)` als Zuweisung compilierbar (wenn auch mittlerweile mit einer Warnung), `if(58000000000 = worldExpoShanghaiCostInUSD)` aber nicht. Die erste fehlerhafte Version initialisiert eine Variable und springt immer in die if-Anweisung, da in C jeder Ausdruck (hier von der Zuweisung, die ja ein Ausdruck ist) ungleich 0 als wahr interpretiert wird. Das ist ein logischer Fehler, den die zweite Schreibweise verhindert, denn sie führt zu einem Compilerfehler. In Java ist dieser Fehlertyp nicht zu finden – es sei denn, der Variablenotyp ist `boolean`, was sehr selten vorkommt –, und so sollte diese Yoda-Schreibweise vermieden werden.

2.4.7 Logische Operatoren: Nicht, Und, Oder, Xor

Die Abarbeitung von Programmcode ist oft an Bedingungen geknüpft. Diese Bedingungen sind oftmals komplex zusammengesetzt, wobei drei Operatoren am häufigsten vorkommen:

- *Nicht (Negation)*: Dreht die Aussage um: Aus *wahr* wird *falsch*, und aus *falsch* wird *wahr*.
- *Und (Konjunktion)*: Beide Aussagen müssen wahr sein, damit die Gesamtaussage wahr wird.
- *Oder (Disjunktion)*: Eine der beiden Aussagen muss wahr sein, damit die Gesamtaussage wahr wird.

Mit logischen Operatoren werden Wahrheitswerte nach definierten Mustern verknüpft. Logische Operatoren operieren nur auf `boolean`-Typen, andere Typen führen zu Com-

²⁴ Yoda ist eine Figur aus Star Wars, die eine für uns ungewöhnliche Satzstellung nutzt. Anstatt Sätze mit Subjekt + Prädikat + Objekt (SPO) aufzubauen, nutzt Yoda die Form Objekt + Subjekt + Prädikat (OSP), etwa bei »Begun the Clone War has«. Objekt und Subjekt sind umgedreht, so wie die Operanden aus dem Beispiel auch, sodass dieser Ausdruck sich so lesen würde: »Wenn 58000000000 gleich `worldExpoShanghaiCostInUSD` ist« statt der üblichen SPO-Lesung »wenn `worldExpoShanghaiCostInUSD` ist gleich 58000000000«. Im Arabischen ist diese OSP-Stellung üblich, sodass Entwickler aus dem arabischen Sprachraum diese Form eigentlich natürlich finden könnten. Wenn das mal nicht eine Studie wert ist ...

pilerfehlern. Java bietet die Operatoren *Nicht* (!), *Und* (&&), *Oder* (||) und *Xor* (^) an. Xor ist eine Operation, die genau dann wahr liefert, wenn genau einer der beiden Operanden wahr ist. Sind beide Operanden gleich (also entweder true oder false), so ist das Ergebnis false. Xor heißt auch *exklusives* beziehungsweise *ausschließendes Oder*. Im Deutschen trifft die Formulierung »entweder ... oder« diesen Sachverhalt gut: Entweder ist es das eine *oder* das andere, aber nicht beides zusammen. Beispiel: »Willst du entweder ins Kino oder DVD schauen?«

boolean a	boolean b	! a	a && b	a b	a ^ b
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Tabelle 2.10: Verknüpfungen der logischen Operatoren »Nicht«, »Und«, »Oder« und »Xor«

Die logischen Operatoren arbeiten immer auf dem Typ boolean. In Abschnitt 18.1.1, »Die Bit-Operatoren Komplement, Und, Oder und Xor«, werden wir sehen, dass sich die gleichen Verknüpfungen auf jedem Bit einer Ganzzahl durchführen lassen.

Ausblick auf die Aussagenlogik

Verknüpfungen dieser Art sind in der Aussagenlogik bzw. booleschen Algebra sehr wichtig. Die für uns gängigen Begriffe Und, Oder, Xor sind dort auch unter anderen Namen bekannt. Die Und-Verknüpfung nennt sich *Konjunktion*, die Oder-Verknüpfung *Disjunktion*, und das exklusive Oder heißt *Kontravalenz*. Die drei binären Operatoren Und, Oder, Xor decken bestimmte Verknüpfungen ab, jedoch nicht alle, die prinzipiell möglich sind. In der Aussagenlogik gibt es weiterhin die *Implikation* (Wenn-Dann-Verknüpfung) und die *Äquivalenz*. Für beide gibt es keinen eigenen Operator. Bei der Implikation ist es das Ergebnis von $a \mid\mid !b$, und bei der logischen Äquivalenz ist es die Negation der Kontravalenz (Xor), daher auch Exklusiv-nicht-oder-Verknüpfung genannt, also ein $!(a \wedge b)$. Logische Äquivalenz herrscht demnach immer dann, wenn beide Wahrheitswerte gleich sind, also entweder a und b true oder a und b false sind. Wird die Konjunktion (also das Und) negiert, entspricht das in der digitalen Elektronik dem *NAND-Gatter* bzw. dem *Shefferschen Strich*. Es ist also definiert als $!(a \wedge b)$. Die Negation eines Oders, also $!(a \mid\mid b)$, entspricht der *Peirce-Funktion* – in der digitalen Elektronik einem *NOR-Gatter*. Auch dafür gibt es in Java keine eigenen Operatoren. Wem jetzt der Kopf raucht: Kein Problem, dass brauchen wir alles nicht.



2.4.8 Kurzschluss-Operatoren

Eine Besonderheit sind die beiden Operatoren `&&` (Und) beziehungsweise `||` (Oder). In der Regel muss ein logischer Ausdruck nur dann weiter ausgewertet werden, wenn er das Endergebnis noch beeinflussen kann. Zwei Operatoren bieten sich zur Optimierung der Ausdrücke an:

- *Und*: Ist einer der beiden Ausdrücke falsch, so kann der Ausdruck schon nicht mehr wahr werden. Das Ergebnis ist falsch.
- *Oder*: Ist mindestens einer der Ausdrücke schon wahr, so ist auch der gesamte Ausdruck wahr.

Der Compiler bzw. die Laufzeitumgebung kann den Programmfluss abkürzen. Daher nennen sich die beiden Operatoren auch *Kurzschluss-Operatoren* (engl. *short-circuit operators*).²⁵ Kürzt der Compiler ab, wertet er nur den ersten Ausdruck aus und den zweiten dann nicht mehr.

Nicht-Kurzschluss-Operatoren *

In einigen Fällen ist es erwünscht, dass die Laufzeitumgebung alle Teilausdrücke auswertet. Das kann der Fall sein, wenn Methoden Nebenwirkungen haben sollen, etwa Zustände ändern. Daher bietet Java zusätzlich die nicht über einen Kurzschluss arbeitenden Operatoren `|` und `&` an, die eine Auswertung aller Teilausdrücke erzwingen.

zB Beispiel

In der ersten und dritten Anweisung wird die Methode `boolean f()` *nicht* aufgerufen, in der zweiten und vierten schon.

```
System.out.println( true || f() ); // true, f() wird nicht aufgerufen
System.out.println( true | f() ); // true, f() wird aufgerufen
System.out.println( false && f() ); // false, f() wird nicht aufgerufen
System.out.println( false & f() ); // false, f() wird aufgerufen
```

Für Xor kann es keinen Kurzschluss-Operator geben, da immer beide Operanden ausgewertet werden müssen, bevor das Ergebnis feststeht.

²⁵ Den Begriff verwendet die Java-Sprachdefinition nicht! Siehe dazu auch http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.23.

2.4.9 Der Rang der Operatoren in der Auswertungsreihenfolge

Aus der Schule ist der Spruch »Punktrechnung geht vor Strichrechnung« bekannt, so dass sich der Ausdruck $1 + 2 * 3$ zu 7 und nicht zu 9 auswertet.²⁶

Beispiel

zB

Auch wenn bei Ausdrücken wie $a() + b() * c()$ zuerst das Produkt gebildet wird, schreibt doch die Auswertungsreihenfolge von binären Operatoren vor, dass der linke Operand zuerst ausgewertet werden muss, was bedeutet, dass Java zuerst die Methode $a()$ aufruft.

In den meisten Programmiersprachen gibt es eine Unzahl von Operatoren neben Plus und Mal, die alle ihre eigenen Vorrangregeln besitzen.²⁷ Der Multiplikationsoperator besitzt zum Beispiel eine höhere Priorität und damit eine andere Auswertungsreihenfolge als der Plus-Operator. Die *Rangordnung* der Operatoren (engl. *operator precedence*) legt folgende Tabelle fest, wobei der arithmetische Typ für Ganz- und Fließkommazahlen steht und der integrale Typ für `char` und Ganzzahlen:

Operator	Rang	Typ	Beschreibung
<code>++, --</code>	1	arithmetisch	Inkrement und Dekrement
<code>+, -</code>	1	arithmetisch	unäres Plus und Minus
<code>~</code>	1	integral	bitweises Komplement
<code>!</code>	1	boolean	logisches Komplement
<code>(Typ)</code>	1	jeder	Cast
<code>*, /, %</code>	2	arithmetisch	Multiplikation, Division, Rest
<code>+, -</code>	3	arithmetisch	Addition und Subtraktion
<code>+</code>	3	String	String-Konkatenation
<code><<</code>	4	integral	Verschiebung links
<code>>></code>	4	integral	Rechtsverschiebung mit Vorzeichenerweiterung
<code>>>></code>	4	integral	Rechtsverschiebung ohne Vorzeichenerweiterung

Tabelle 2.11: Operatoren mit Rangordnung in Java

²⁶ Dass von diesen Rechnungen eine gewisse Spannung ausgeht, zeigen diverse Fernsehkanäle, die damit ihr Abendprogramm füllen.

²⁷ Es gibt Programmiersprachen wie APL, die keine Vorrangregeln kennen. Sie werten die Ausdrücke streng von rechts nach links oder umgekehrt aus.

Operator	Rang	Typ	Beschreibung
<, <=, >, >=	5	arithmetisch	numerische Vergleiche
instanceof	5	Objekt	Typvergleich
==, !=	6	primitiv	Gleich-/Ungleichheit von Werten
==, !=	6	Objekt	Gleich-/Ungleichheit von Referenzen
&	7	integral	bitweises Und
&	7	boolean	logisches Und
^	8	integral	bitweises Xor
^	8	boolean	logisches Xor
	9	integral	bitweises Oder
	9	boolean	logisches Oder
&&	10	boolean	logisches konditionales Und, Kurzschluss
	11	boolean	logisches konditionales Oder, Kurzschluss
?:	12	jeder	Bedingungsoperator
=	13	jeder	Zuweisung
*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, =	14	jeder	Zuweisung mit Operation

Tabelle 2.11: Operatoren mit Rangordnung in Java (Forts.)

Die Rechenregel für »Mal vor Plus« kann sich jeder noch leicht merken. Auch ist leicht zu merken, dass die typischen arithmetischen Operatoren wie Plus und Mal eine höhere Priorität als Vergleichsoperationen haben. Komplizierter ist die Auswertung bei den zahlreichen Operatoren, die seltener im Programm vorkommen.

zB Beispiel

Wie ist die Auswertung bei dem nächsten Ausdruck?

```
boolean A = false,
      B = false,
      C = true;
System.out.println( A && B || C );
```

Das Ergebnis könnte je nach Rangordnung true oder false sein. Doch die Tabelle lehrt uns, dass im Beispiel A && B || C das Und stärker als das Oder bindet, also der Ausdruck mit der Belegung A=false, B=false, C=true zu true ausgewertet wird.

Vermutlich gibt es Programmierer, die dies wissen oder eine Tabelle mit Rangordnungen am Monitor kleben haben. Aber beim Durchlesen von fremdem Code ist es nicht schön, immer wieder die Tabelle konsultieren zu müssen, die verrät, ob nun das binäre Xor oder das binäre Und stärker bindet.

Tipp

Alle Ausdrücke, die über die einfache Regel »Punktrechnung geht vor Strichrechnung« hinausgehen, sollten geklammert werden. Da die unären Operatoren ebenfalls sehr stark binden, kann eine Klammerung wegfallen.

**Links- und Rechtsassoziativität ***

Bei den Operatoren + und * gilt die mathematische Kommutativität und Assoziativität. Das heißt, die Operanden können prinzipiell umgestellt werden, und das Ergebnis sollte davon nicht beeinträchtigt sein. Bei der Division unterscheiden wir zusätzlich *Links- und Rechtsassoziativität*. Deutlich wird das am Beispiel A / B / C. Den Ausdruck wertet Java von links nach rechts aus, und zwar als (A / B) / C; daher ist der Divisionsoperator linksassoziativ. Hier sind Klammern angemessen. Denn würde der Compiler den Ausdruck zu A / (B / C) auswerten, käme dies einem A * C / B gleich. In Java sind die meisten Operatoren linksassoziativ, aber es gibt Ausnahmen, wie Zuweisungen der Art A = B = C, die der Compiler zu A = (B = C) auswertet.

Hinweis

Die mathematische Assoziativität ist natürlich gefährdet, wenn durch Überläufe Rechenfehler mit im Spiel sind:

```
float a = -16777217F;  
float b = 16777216F;  
float c = 1F;  
System.out.println( a + b + c ); // 1.0  
System.out.println( a + (b + c) ); // 0.0
```

Mathematisch ergibt $-16777217 + 16777216$ den Wert -1 , und -1 plus $+1$ ist 0 . Im zweiten Fall liefert $-16777217 + (16777216 + 1) = -16777217 + 16777217 = 0$. Doch Java wertet $a + b$ durch die Beschränkung von float zu 0 aus, sodass mit c addiert, also 1 , die Ausgabe 1 statt 0 erscheint.

2.4.10 Die Typanpassung (das Casting)

Zwar ist Java eine getypte Sprache, aber sie ist nicht so stark getypt, dass es hinderlich ist. So übersetzt der Compiler die folgenden Zeilen problemlos:

```
int anInt = 1;
long long1 = 1;
long long2 = anInt;
```

Streng genommen *könnte* ein Compiler bei einer sehr starken Typisierung die letzten beiden Zeilen ablehnen, denn das Literal 1 ist vom Typ `int` und kein `1L`, also `long`, und in `long2 = anInt` ist die Variable `anInt` vom Typ `int` statt vom gewünschten Datentyp `long`.

Arten der Typanpassung

In der Praxis kommt es also vor, dass Datentypen konvertiert werden müssen. Dies nennt sich *Typanpassung* (engl. *typecast*, kurz *cast*). Java unterscheidet zwei Arten der Typanpassung:

- *Automatische (implizite) Typanpassung*: Daten eines kleineren Datentyps werden automatisch (implizit) dem größeren angepasst. Der Compiler nimmt diese Anpassung selbstständig vor. Daher funktioniert unser erstes Beispiel mit etwa `long2 = anInt`.
- *Explizite Typanpassung*: Ein größerer Typ kann einem kleineren Typ mit möglichem Verlust von Informationen zugewiesen werden.

Typanpassungen gibt es bei primitiven Datentypen und bei Referenztypen. Während die folgenden Absätze die Anpassungen bei einfachen Datentypen beschreiben, kümmert sich Kapitel 5, »Eigene Klassen schreiben«, um die Typkompatibilität bei Referenzen.

Automatische Anpassung der Größe

Werte der Datentypen `byte` und `short` werden bei Rechenoperationen automatisch in den Datentyp `int` umgewandelt. Ist ein Operand vom Datentyp `long`, dann werden alle Operanden auf `long` erweitert. Wird aber `short` oder `byte` als Ergebnis verlangt, dann ist dieses durch einen expliziten Typecast anzugeben, und nur die niederwertigen Bits des Ergebniswerts werden übergeben. Folgende Typumwandlungen führt Java automatisch aus:

Vom Typ	In den Typ
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Tabelle 2.12: Implizite Typanpassungen

Die Anpassung wird im Englischen *widening conversion* genannt, weil sie den Wertebereich automatisch erweitert.

Hinweis

Obwohl von der Datentypgröße her ein char (16 Bit) zwischen byte (8 Bit) und int (32 Bit) liegt, taucht der Typ in einer rechten Spalte der oberen Tabelle nicht auf, da char kein Vorzeichen speichern kann, während die anderen Datentypen byte, short, int, long, float, double alle ein Vorzeichen besitzen. Daher kann so etwas wie das Folgende nicht funktionieren:

```
byte b = 'b';
char c = b; // ☹ Type mismatch: cannot convert from byte to char
```

Explizite Typanpassung

Die explizite Anpassung engt einen Typ ein, sodass diese Operation im Englischen *narrowing conversion* genannt wird. Der gewünschte Typ für eine Typanpassung wird vor den umzuwandelnden Datentyp in Klammern gesetzt.

Beispiel

zB

Umwandlung einer Fließkommazahl in eine Ganzzahl:

```
int n = (int) 3.1415; // n = 3
```

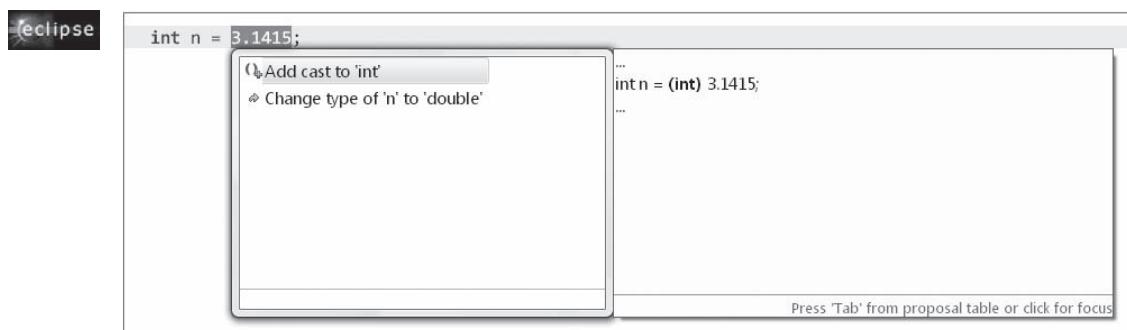


Abbildung 2.5: Passt der Typ eines Ausdrucks nicht, lässt er sich mit [Strg] + [1] korrigieren.

Eine Typumwandlung hat eine sehr hohe Priorität. Daher muss der Ausdruck gegebenenfalls geklammert werden.

zB Beispiel

Die Zuweisung an n verfehlt das Ziel:

```
int n = (int) 1.0315 + 2.1;  
int m = (int)(1.0315 + 2.1); // das ist korrekt
```

Typumwandlung von Fließkommazahlen in Ganzzahlen

Bei der expliziten Typumwandlung von double und float in einen Ganzzahltyp kann es selbstverständlich zum Verlust von Genauigkeit kommen sowie zur Einschränkung des Wertebereichs. Bei der Konvertierung von Fließkommazahlen verwendet Java eine Rundung gegen null.

```
System.out.println( (int) +12.34 ); // 12  
System.out.println( (int) +67.89 ); // 67  
System.out.println( (int) -12.34 ); // -12  
System.out.println( (int) -67.89 ); // -67
```

Automatische Typanpassung bei Berechnungen mit byte und short auf int *

Eine Operation vom Typ int mit int liefert den Ergebnistyp int, und long mit long liefert ein long.

Listing 2.11: AutoConvert.java, main()

```
int i1 = 1, i2 = 2;
int i3 = i1 + i2;
long l1 = 1, l2 = 2;
long l3 = l1 + l2;
```

Diese Zeilen übersetzt der Compiler wie erwartet. Und so erscheint es logisch, dass das Gleiche auch für die Datentypen `short` und `byte` gilt.

```
short s1 = 1, s2 = 2;
byte b1 = 1, b2 = 2;
// short s3 = s1 + s2; // ☹ Type mismatch: cannot convert from int to short
// byte b3 = b1 + b2; // ☹ Type mismatch: cannot convert from int to byte
```

Die auskommentierten Zeilen machen schon deutlich: Es ist nicht möglich, ohne explizite Typumwandlung zwei `short`- oder `byte`-Zahlen zu addieren. Richtig ist:

```
short s3 = (short)(s1 + s2);
byte b3 = (byte)(b1 + b2);
```

Der Grund liegt beim Java-Compiler. Wenn Ganzzahl-Ausdrücke vom Typ kleiner `int` mit einem Operator verbunden werden, passt der Compiler eigenmächtig den Typ auf `int` an. Die Addition der beiden Zahlen im Beispiel arbeitet also nicht mit `short`- oder `byte`-Werten, sondern mit `int`-Werten; intern im Bytecode ist es ebenso realisiert. So führen also alle Ganzzahloperationen mit `short` und `byte` automatisch zum Ergebnistyp `int`. Und das führt bei der Zuweisung aus dem Beispiel zu einem Problem, denn steht auf der rechten Seite der Typ `int` und auf der linken Seite der kleinere Typ `byte` oder `short`, muss der Compiler einen Fehler melden. Mit der ausdrücklichen Typumwandlung erzwingen wir diese Konvertierung.

Dass der Compiler diese Anpassung vornimmt, müssen wir einfach akzeptieren. `int` und `int` bleibt `int`, `long` und `long` bleibt `long`. Wenn ein `int` mit einem `long` tanzt, wird der Ergebnistyp `long`. Arbeitet der Operator auf einem `short` oder `byte`, ist das Ergebnis automatisch `int`.

28 http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.6.1

**Tipp**

Kleine Typen wie short und byte führen oft zu Problemen. Wenn sie nicht absichtlich in großen Feldern verwendet werden und Speicherplatz nicht ein absolutes Kriterium ist, erweist sich int als die beste Wahl – auch weil Java nicht durch besonders intuitive Typ-Konvertierungen glänzt, wie das Beispiel mit dem unären Minus und Plus zeigt:

```
byte b = 0;
b = -b;           // ☹ Cannot convert from int to byte
b = +b;           // ☹ Cannot convert from int to byte
```

Der Compiler meldet einen Fehler, denn der Ausdruck auf der rechten Seite wird durch den unären Operator in ein int umgewandelt, was immer für die Typen byte, short und char gilt.²⁸

Keine Typanpassung zwischen einfachen Typen und Referenztypen

Allgemeine Umwandlungen zwischen einfachen Typen und Referenztypen gibt es nicht. Falsch sind zum Beispiel:

Listing 2.12: TypecastPrimRef.java, main() Teil 1

```
String s = (String) 1;      // ☹ Cannot cast from int to String
int i = (int) "1";         // ☹ Cannot cast from String to int
```

**Getrickse mit Boxing**

Einiges sieht dagegen nach Typanpassung aus, ist aber in Wirklichkeit eine Technik, die sich Autoboxing nennt (Abschnitt 8.2, »Wrapper-Klassen und Autoboxing«, geht näher darauf ein):

Listing 2.13: TypecastPrimRef.java, main() Teil 2

```
Long lōng = (Long) 2L;      // Alternativ: Long lōng = 2L;
System.out.println( (Boolean) true );
((Integer)2).toString();
```

Typanpassung beim Verbundoperator *

Beim Verbundoperator wird noch etwas mehr gemacht, als E1 #= E2 zu E1 = (E1) # (E2) aufzulösen, wobei # symbolisch für einen binären Operator steht. Interessanterweise

kommt auch noch der Typ von E1 ins Spiel, denn der Ausdruck E1 # E2 wird vor der Zuweisung auf den Datentyp von E1 gebracht, sodass es genau heißen muss: E1 += E2 wird zu E1 = (Typ von E1)((E1) # (E2)).

Beispiel

zB

Der Verbundoperator soll eine Ganzzahl zu einer Fließkommazahl addieren.

```
int i = 1973;
i += 30.2;
```

Die Anwendung des Verbundoperators ist in Ordnung, denn der Übersetzer nimmt eine implizite Typanpassung vor, sodass die Bedeutung bei i = (int)(i + 30.2) liegt. So viel dazu, dass Java eine intuitive und einfache Programmiersprache sein soll.

2.4.11 Überladenes Plus für Strings

Obwohl sich in Java die Operatoren fast alle auf primitive Datentypen beziehen, gibt es doch eine weitere Verwendung des Plus-Operators. Diese wurde in Java eingeführt, da ein Aneinanderhängen von Zeichenketten oft benötigt wird. Objekte vom Typ String können durch den Plus-Operator mit anderen Strings und Datentypen verbunden werden. Falls zusammenhängende Teile nicht alle den Datentyp String annehmen, werden sie automatisch in einen String umgewandelt. Der Ergebnistyp ist immer String.

Beispiel

zB

Setze fünf Teile zu einem String zusammen:

```
String s = " " + "Extrem Sandmännchen" + " " + " frei ab " + 18;
//      char  String           char  String      int
System.out.println( s ); // "Extrem Sandmännchen" frei ab 18
```

Besteht der Ausdruck aus mehreren Teilen, so muss die Auswertungsreihenfolge beachtet werden, andernfalls kommt es zu seltsamen Zusammensetzungen. So ergibt "Aufruf von " + 1 + 0 + 0 + " Ökonomen" tatsächlich »Aufruf von 100 Ökonomen« und nicht »Aufruf von 1 Ökonomen«, da der Compiler die Konvertierung in Strings dann startet, wenn er einen Ausdruck als String-Objekt erkannt hat.

Schauen wir uns die Auswertungsreihenfolge vom Plus an einem Beispiel an:

Listing 2.14: PlusString.java, main()

```
System.out.println( 1 + 2 );           // 3
System.out.println( "1" + 2 + 3 );     // 123
System.out.println( 1 + 2 + "3" );      // 33
System.out.println( 1 + 2 + "3" + 4 + 5 ); // 3345
System.out.println( 1 + 2 + "3" + (4 + 5) ); // 339
```

Nur eine Zeichenkette in doppelten Anführungszeichen ist ein String, und der Plus-Operator entfaltet seine besondere Wirkung. Ein einzelnes Zeichen in einfachen Hochkomata wird lediglich auf ein int gecastet, und Additionen sind Ganzzahl-Additionen.

```
System.out.println( '0' + 2 );      // 50 - ASCII value for '0' is 48
System.out.println( 'A' + 'a' );     // 162 - 'A'=65, 'a'=97
```

zB Beispiel

Der Plus-Operator für Zeichenketten geht streng von links nach rechts vor und bereitet mit eingebetteten arithmetischen Ausdrücken mitunter Probleme. Eine Klammerung hilft, wie im Folgenden zu sehen ist:

```
"Ist 1 größer als 2? " + (1 > 2 ? "ja" : "nein");
```

Wäre der Ausdruck um den Bedingungsoperator nicht geklammert, dann würde der Plus-Operator an die Zeichenkette die 1 anhängen, und es käme der >-Operator. Der erwartet aber kompatible Datentypen, die in unserem Fall – links stünde die Zeichenkette und rechts die Ganzzahl 2 – nicht gegeben sind.

2.4.12 Operator vermisst *

Da es in Java keine Pointer-Operationen gibt, existieren die unter C(++) bekannten Operatorzeichen zur Referenzierung (&) und Dereferenzierung (*) nicht. Ebenso ist ein sizeof unnötig, da das Laufzeitsystem und der Compiler immer die Größe von Klassen kennen beziehungsweise die primitiven Datentypen immer eine feste Länge haben. Eine abgeschwächte Version vom Komma-Operator ist in Java nur im Kopf von for-Schleifen erlaubt. Einige Programmiersprachen haben einen Potenz-Operator (etwa **), den es in Java ebenfalls nicht gibt. Skript-Sprachen wie Perl oder Python bieten nicht nur einfache Datentypen, sondern definieren zum Beispiel Listen oder Assoziativspeicher. Damit sind automatisch Operatoren assoziiert, etwa um die Datenstrukturen nach

Werten zu fragen oder Elemente einzufügen. Zudem erlauben viele Skript-Sprachen das Prüfen von Zeichenketten gegen reguläre Ausdrücke, etwa Perl mit den Operatoren `=~` bzw. `!~`.

2.5 Bedingte Anweisungen oder Fallunterscheidungen

Kontrollstrukturen dienen in einer Programmiersprache dazu, Programmteile unter bestimmten Bedingungen auszuführen. Java bietet zum Ausführen verschiedener Programmteile eine `if`- und `if-else`-Anweisung sowie die `switch`-Anweisung. Neben der Verzweigung dienen Schleifen dazu, Programmteile mehrmals auszuführen. Bedeutend im Wort »Kontrollstrukturen« ist der Teil »Struktur«, denn die Struktur zeigt sich schon durch das bloße Hinsehen. Als es noch keine Schleifen und »hochwertigen« Kontrollstrukturen gab, sondern nur ein Wenn/Dann und einen Sprung, war die Logik des Programms nicht offensichtlich; das Resultat nannte sich *Spaghetti-Code*. Obwohl ein allgemeiner Sprung in Java mit `goto` nicht möglich ist, besitzt die Sprache dennoch eine spezielle Sprungvariante. In Schleifen erlauben `continue` und `break` definierte Sprungziele.

2.5.1 Die if-Anweisung

Die `if`-Anweisung besteht aus dem Schlüsselwort `if`, dem zwingend ein Ausdruck mit dem Typ `boolean` in Klammern folgt. Es folgt eine Anweisung, die oft eine Blockanweisung ist.

Mit der `if`-Anweisung wollen wir testen, ob der Anwender eine Zufallszahl richtig geraten hat.

Listing 2.15: WhatsYourNumber.java

```
public class WhatsYourNumber
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);

        System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
        int guess = new java.util.Scanner( System.in ).nextInt();
```

```

if ( number == guess )
    System.out.println( "Super getippt!" );

System.out.println( "Starte das Programm noch einmal und rate erneut!" );
}
}

```

Die Abarbeitung der Ausgabe-Anweisungen hängt vom Ausdruck im if ab.

- Ist das Ergebnis des Ausdrucks wahr (`number == guess` wird zu `true` ausgewertet), wird die folgende Anweisung, also die Konsolenausgabe "Super getippt!" ausgeführt.
- Ist das Ergebnis des Ausdrucks falsch (`number == guess` wird zu `false` ausgewertet), so wird die Anweisung übersprungen, und es wird mit der ersten Anweisung *nach* der if-Anweisung fortgefahren.



Programmiersprachenvergleich

Im Gegensatz zu C(++) und vielen Skriptsprachen muss in Java der Testausdruck für die Bedingung der if-Anweisung ohne Ausnahme vom Typ boolean sein – für Schleifenbedingungen gilt das Gleiche. C(++) bewertet einen numerischen Ausdruck als wahr, wenn das Ergebnis des Ausdrucks ungleich 0 ist – so ist auch `if (10)` gültig, was in Java einem `if(true)` entspräche.

if-Abfragen und Blöcke

Hinter dem if und der Bedingung erwartet der Compiler eine Anweisung. Sind mehrere Anweisungen in Abhängigkeit von der Bedingung auszuführen, ist ein Block zu setzen; andernfalls ordnet der Compiler nur die nächstfolgende Anweisung der Fallunterscheidung zu, auch wenn mehrere Anweisungen optisch abgesetzt sind.²⁹ Dies ist eine große Gefahr für Programmierer, die optisch Zusammenhänge schaffen wollen, die in Wirklichkeit nicht existieren.

Dazu ein Beispiel: Eine if-Anweisung soll testen, ob die geratene Zahl gleich der Zufallszahl war, und dann, wenn das der Fall war, eine Ausgabe liefern und die Variable `number` mit einer neuen Zufallszahl belegen. Zunächst die semantisch falsche Variante:

²⁹ In der Programmiersprache Python bestimmt die Einrückung die Zugehörigkeit.

```
if ( number == guess )
    number = (int) (Math.random() * 5 + 1);
    System.out.println( "Super getippt!" );
```

Die Implementierung ist semantisch falsch, da unabhängig vom Test immer die Ausgabe erscheint. Der Compiler interpretiert die Anweisungen in folgendem Zusammenhang, wobei die Einrückung die tatsächliche Ausführung widerspiegelt:

```
if ( number == guess )
    number = (int) (Math.random() * 5 + 1);
    System.out.println( "Super getippt!" );
```

Für unsere gewünschte Logik, beide Anweisungen zusammen in Abhängigkeit von der Bedingung auszuführen, heißt es, sie in einen Block zu setzen:

```
if ( number == guess )
{
    number = (int) (Math.random() * 5 + 1);
    System.out.println( "Super getippt!" );
}
```

Tipp

Einrückungen ändern nicht die Semantik des Programms! Einschübe können das Verständnis nur empfindlich stören. Damit das Programm korrekt wird, müssen wir einen Block verwenden und die Anweisungen zusammensetzen. Entwickler sollten Einrückungen konsistent zur Verdeutlichung von Abhängigkeiten nutzen.



Zusammengesetzte Bedingungen

Die bisherigen Abfragen waren sehr einfach, doch kommen in der Praxis viel komplexere Bedingungen vor. Oft im Einsatz sind die logischen Operatoren `&&` (Und), `||` (Oder), `!` (Nicht).

Wenn wir etwa testen wollen, ob

- eine geratene Zahl `number` entweder gleich der Zufallszahl `guess` ist *oder*
- eine gewisse Anzahl von Versuchen schon überschritten ist (`trials` größer 10),

dann schreiben wir die zusammengesetzte Bedingung so:

```
if ( number == guess || trials > 10 )
    ...
}
```

Sind die logisch verknüpften Ausdrücke komplexer, so sollten zur Unterstützung der Lesbarkeit die einzelnen Bedingungen in Klammern gesetzt werden, da nicht jeder sofort die Tabelle mit den Vorrangregeln für die Operatoren im Kopf hat.

2.5.2 Die Alternative mit einer if-else-Anweisung wählen

Neben der einseitigen Alternative existiert die zweiseitige Alternative. Das optionale Schlüsselwort `else` mit angehängter Anweisung veranlasst die Ausführung einer Alternative, wenn der `if`-Test falsch ist.

Rät der Benutzer aus unserem kleinen Spiel die Zahl nicht, wollen wir ihm die Zufallszahl präsentieren:

Listing 2.16: GuessTheNumber.java

```
public class GuessTheNumber
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);

        System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
        int guess = new java.util.Scanner( System.in ).nextInt();

        if ( number == guess )
            System.out.println( "Super getippt!" );
        else
            System.out.printf( "Tja, stimmt nicht, habe mir %s gedacht!", number );
    }
}
```

Falls der Ausdruck `number == guess` wahr ist, wird die erste Anweisung ausgeführt, andernfalls die zweite Anweisung. Somit ist sichergestellt, dass in jedem Fall eine Anweisung ausgeführt wird.

Das Dangling-Else-Problem

Bei Verzweigungen mit `else` gibt es ein bekanntes Problem, das *Dangling-Else-Problem* genannt wird. Zu welcher Anweisung gehört das folgende `else`?

```
if ( Ausdruck1 )
    if ( Ausdruck2 )
        Anweisung1;
else
    Anweisung2;
```

Die Einrückung suggeriert, dass das `else` die Alternative zur ersten `if`-Anweisung ist. Dies ist aber nicht richtig. Die Semantik von Java (und auch fast aller anderen Programmiersprachen) ist so definiert, dass das `else` zum innersten `if` gehört. Daher lässt sich nur der Programmertipp geben, die `if`-Anweisungen zu klammern:

```
if ( Ausdruck1 )
{
    if ( Ausdruck2 )
    {
        Anweisung1;
    }
}
else
{
    Anweisung2;
}
```

So kann eine Verwechslung gar nicht erst aufkommen. Wenn das `else` immer zum innersten `if` gehört und das nicht erwünscht ist, können wir, wie gerade gezeigt, mit geschweiften Klammern arbeiten oder auch eine leere Anweisung im `else`-Zweig hinzufügen:

```
if ( x >= 0 )
    if ( x != 0 )
        System.out.println( "x echt größer null" );
    else
        ; // x ist gleich null
else
    System.out.println( "x echt kleiner null" );
```

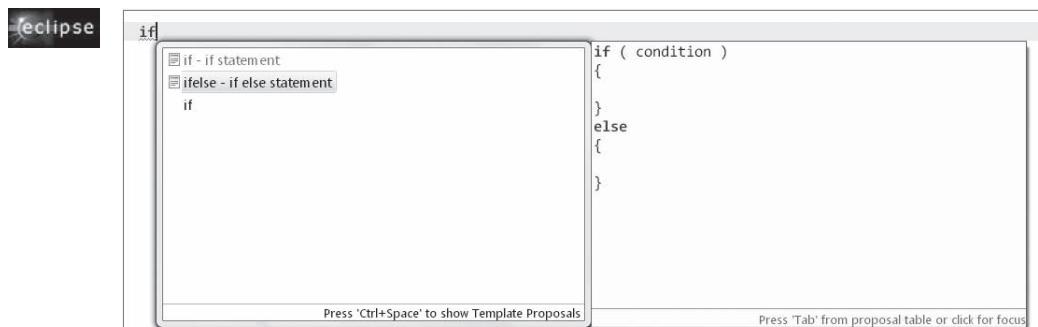


Abbildung 2.6: »if« und **Strg** + Leertaste bietet an, eine if-Anweisung mit Block anzulegen.

Das böse Semikolon

An dieser Stelle ist ein Hinweis angebracht: Ein Programmieranfänger schreibt gerne hinter die schließende Klammer der if-Anweisung ein Semikolon. Das führt zu einer ganz anderen Ausführungsfolge. Ein Beispiel:

```
int age = 29;
if ( age < 0 ) ;           // ☹ logischer Fehler
    System.out.println( "Aha, noch im Mutterleib" );
if ( age > 150 ) ;         // ☹ logischer Fehler
    System.out.println( "Aha, ein neuer Moses" );
```

Das Semikolon führt dazu, dass die leere Anweisung in Abhängigkeit von der Bedingung ausgeführt wird und unabhängig vom Inhalt der Variablen `age` immer die Ausgabe »Aha, noch im Mutterleib« und »Aha, ein neuer Moses« erzeugt. Das ist sicherlich nicht beabsichtigt. Das Beispiel soll ein warnender Hinweis sein, in jeder Zeile nur eine Anweisung zu schreiben – und die leere Anweisung durch das Semikolon ist eine Anweisung.

Folgen hinter einer if-Anweisung zwei Anweisungen, die durch keine Blockanweisung zusammengefasst sind, dann wird die eine folgende else-Anweisung als Fehler bemängelt, da der zugehörige if-Zweig fehlt. Der Grund ist, dass der if-Zweig nach der ersten Anweisung ohne else zu Ende ist:

```
int age = 29;
if ( age < 0 )
;
System.out.println( "Aha, noch im Mutterleib" );
```

```
else if ( age > 150 ) ;      // ☠ Compiler-Fehlermeldung: 'else' without 'if'
System.out.println( "Aha, ein neuer Moses" );
```

Mehrfachverzweigung beziehungsweise geschachtelte Alternativen

if-Anweisungen zur Programmführung kommen sehr häufig in Programmen vor, und noch häufiger werden sie genutzt, um eine Variable auf einen bestimmten Wert zu prüfen. Dazu werden if- und if-else-Anweisungen gerne verschachtelt (kaskadiert). Wenn eine Variable einem Wert entspricht, dann wird eine Anweisung ausgeführt, sonst wird die Variable mit einem anderen Wert getestet und so weiter.

Beispiel

zB

Kaskadierte if-Anweisungen sollen uns helfen, die Variable days passend nach dem Monat (vorbelegte Variable month) und der Information, ob das Jahr ein Schaltjahr ist (vorbelegte boolean-Variable isLeapYear), zu belegen:

```
if ( month == 4 )
    days = 30;
else if ( month == 6 )
    days = 30;
else if ( month == 9 )
    days = 30;
else if ( month == 11 )
    days = 30;
else if ( month == 2 )
    if ( isLeapYear )      // Sonderbehandlung im Fall eines Schaltjahrs
        days = 29;
    else
        days = 28;
    else
        days = 31;
```

Die eingerückten Verzweigungen nennen sich auch *angehäufte if-Anweisungen* oder *if-Kaskade*, da jede else-Anweisung ihrerseits weitere if-Anweisungen enthält, bis alle Abfragen gemacht sind.

Angewendet auf unser Zahlenratespiel, wollen wir dem Benutzer einen Tipp geben, ob seine eingegebene Zahl kleiner oder größer als die zu ratende Zahl war.

Listing 2.17: GuessTheNumber2.java

```

public class GuessTheNumber2
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);

        System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
        int guess = new java.util.Scanner( System.in ).nextInt();

        if ( number == guess )
            System.out.println( "Super getippt!" );
        else if ( number > guess )
            System.out.println( "Nee, meine Zahl ist größer als deine!" );
        else if ( number < guess )
            System.out.println( "Nee, meine Zahl ist kleiner als deine!" );
    }
}

```

2.5.3 Der Bedingungsoperator

In Java gibt es ebenso wie in C(++) einen Operator, der drei Operanden benutzt. Dies ist der *Bedingungsoperator*, der auch *Konditionaloperator*, *ternärer Operator* beziehungsweise *trinärer Operator* genannt wird. Er erlaubt es, den Wert eines Ausdrucks von einer Bedingung abhängig zu machen, ohne dass dazu eine `if`-Anweisung verwendet werden muss. Die Operanden sind durch `:` beziehungsweise `;` voneinander getrennt.

zB Beispiel

Die Bestimmung des Maximums ist eine schöne Anwendung des trinären Operators:

```
max = ( a > b ) ? a : b;
```

Der Wert der Variablen `max` wird in Abhängigkeit von der Bedingung `a > b` gesetzt. Der Ausdruck entspricht folgender `if`-Anweisung:

zB

2

Beispiel (Forts.)

```
if ( a > b )
    max = a;
else
    max = b;
```

Drei Ausdrücke kommen im Bedingungsoperator vor, daher heißt der Operator auch ternärer/trinärer Operator, vom lateinischen *ternarius* (»aus drei bestehend«). Der erste Ausdruck – in unserem Fall der Vergleich `a > b` – muss vom Typ boolean sein. Ist die Bedingung erfüllt, dann erhält die Variable den Wert des zweiten Ausdrucks, andernfalls wird der Variablen `max` der Wert des dritten Ausdrucks zugewiesen. Der Bedingungsoperator kann eingesetzt werden, wenn der zweite und dritte Operand ein numerischer Typ, boolescher Typ oder Referenztyp ist. Der Aufruf von Methoden, die demnach void zurückgeben, ist nicht gestattet.

Mit dem Rückgabewert können wir alles Mögliche machen, etwa ihn direkt ausgeben:

```
System.out.println( ( a > b ) ? a : b );
```

Das wäre mit if-else nur mit temporären Variablen möglich oder eben mit zwei `println()`-Anweisungen.

Beispiele

Der Bedingungsoperator findet sich häufig in kleinen Methoden:

- Das Maximum oder Minimum zweier Zahlen liefern die Ausdrücke `a > b ? a : b` beziehungsweise `a < b ? a : b`.
- Den Absolutwert einer Zahl liefert `x >= 0 ? x : -x`.
- Ein Ausdruck soll eine Zahl `n`, die zwischen 0 und 15 liegt, in eine Hexadezimalzahl konvertieren: `(char)((n < 10) ? ('0' + n) : ('a' - 10 + n))`.

Geschachtelte Anwendung vom Bedingungsoperator *

Die Anwendung des trinären Operators führt schnell zu schlecht lesbaren Programmen, und er sollte daher vorsichtig eingesetzt werden. In C(++) führt die unbeabsichtigte Mehrfachauswertung in Makros zu schwer auffindbaren Fehlern. Gut, dass uns das in

Java nicht passieren kann! Durch ausreichende Klammerung muss sichergestellt werden, dass die Ausdrücke auch in der beabsichtigten Reihenfolge ausgewertet werden. Im Gegensatz zu den meisten Operatoren ist der Bedingungsoperator rechtsassoziativ (die Zuweisung ist ebenfalls rechtsassoziativ).

Der Ausdruck

`b1 ? a1 : b2 ? a2 : a3`

ist demnach gleichbedeutend mit:

`b1 ? a1 : (b2 ? a2 : a3)`

zB Beispiel

Wollen wir eine Methode schreiben, die für eine Zahl n abhängig vom Vorzeichen $-1, 0$ oder 1 liefert, lösen wir das Problem mit einem geschachtelten trinären Operator:

```
public static int sign( int n )
{
    return (n < 0) ? -1 : (n > 0) ? 1 : 0;
}
```

Der Bedingungsoperator ist kein lvalue *

Der trinäre Operator liefert als Ergebnis einen Ausdruck zurück, der auf der rechten Seite einer Zuweisung verwendet werden kann. Da er rechts vorkommt, nennt er sich auch *rvalue*. Er lässt sich nicht derart auf der linken Seite einer Zuweisung einsetzen, dass er eine Variable auswählt, der ein Wert zugewiesen wird.³⁰

zB Beispiel

Die folgende Anwendung des trinären Operators ist in Java *nicht* möglich:

```
((direction >= 0) ? up : down) = true; // ☠ Compilerfehler
```

³⁰ In C(++) kann dies durch `*((Bedingung) ? &a : &b) = Ausdruck;` über Pointer gelöst werden.

2.5.4 Die switch-Anweisung bietet die Alternative

Eine Kurzform für speziell gebaute, angehäufte if-Anweisungen bietet switch. Im switch-Block gibt es eine Reihe von unterschiedlichen Sprungzielen, die mit case markiert sind. Die switch-Anweisung erlaubt die Auswahl von:

- Ganzzahlen
- Wrapper-Typen (mehr dazu in Kapitel 8)
- Aufzählungen (enum)
- Strings (seit Java 7)

! Interna

Im Bytecode gibt es nur eine switch-Variante für Ganzzahlen. Bei Strings, Aufzählungen und Wrapper-Objekten wendet der Compiler Tricks an, um diese auf Ganzzahl-switch-Konstruktionen zu reduzieren.

switch bei Ganzzahlen (und somit auch chars)

Ein einfacher Taschenrechner für vier binäre Operatoren ist mit switch schnell implementiert (und wir nutzen die Methode charAt(0), um von der String-Eingabe auf das erste Zeichen zuzugreifen, um ein char zu bekommen):

Listing 2.18: Calculator.java

```
public class Calculator
{
    public static void main( String[] args )
    {
        double x = new java.util.Scanner( System.in ).nextDouble();
        char operator = new java.util.Scanner( System.in ).nextLine().charAt( 0 );
        double y = new java.util.Scanner( System.in ).nextDouble();

        switch ( operator )
        {
            case '+':
                System.out.println( x + y );
                break;
        }
    }
}
```

```

    case '-':
        System.out.println( x - y );
        break;
    case '*':
        System.out.println( x * y );
        break;
    case '/':
        System.out.println( x / y );
        break;
    }
}
}
}

```

Die Laufzeitumgebung sucht eine bei `case` genannte *Sprungmarke* (auch *Sprungziel* genannt) – eine Konstante `-`, die mit dem in `switch` angegebenen Ausdruck übereinstimmt. Gibt es einen Treffer, so werden alle auf dem `case` folgenden Anweisungen ausgeführt, bis ein (optionales) `break` die Abarbeitung beendet. (Ohne `break` geht die Ausführung im nächsten `case`-Block automatisch weiter; mehr zu diesem »It's not a bug, it's a feature!« folgt später). Stimmt keine Konstante eines `case`-Blocks mit dem `switch`-Ausdruck überein, werden erst einmal keine Anweisungen im `switch`-Block ausgeführt. Die `case`-Konstanten müssen unterschiedlich sein, andernfalls gibt es einen Compilerfehler.

Die `switch`-Anweisung hat einige Einschränkungen:

- Die JVM kann `switch` nur auf Ausdrücken vom Datentyp `int` ausführen. Elemente vom Datentyp `byte`, `char` und `short` sind somit erlaubt, da der Compiler den Typ automatisch auf `int` anpasst. Ebenso sind die Aufzählungen und die Wrapper-Objekte `Character`, `Byte`, `Short`, `Integer` möglich, da Java automatisch die Werte entnimmt – mehr dazu folgt in Abschnitt 8.2, »Wrapper-Klassen und Autoboxing«. Es können nicht die Datentypen `boolean`, `long`, `float` und `double` benutzt werden. Zwar sind auch Aufzählungen und Strings als `switch`-Ausdruckstypen möglich, doch intern werden sie auf Ganzzahlen abgebildet. Allgemeine Objekte sind sonst nicht erlaubt.
- Die bei `switch` genannten Werte müssen konstant sein. Dynamische Ausdrücke, etwa Rückgaben aus Methodenaufrufen, sind nicht möglich.
- Es sind keine Bereichsangaben möglich. Das wäre etwa bei Altersangaben nützlich, um zum Beispiel die Bereiche 0–18, 19–60, 60–99 zu definieren. Als Lösung bleiben nur angehäuften `if`-Anweisungen.

Hinweis

Die Angabe bei case muss konstant sein, aber kann durchaus aus einer Konstanten (finalen Variablen) kommen:

```
final char PLUS = '+';
switch ( operator ) {
    case PLUS:
        ...
}
```

Alles andere mit default abdecken

Soll ein Programmteil in genau dem Fall abgearbeitet werden, in dem es keine Übereinstimmung mit irgendeiner case-Konstanten gibt, so lässt sich die besondere Sprungmarke **default** einsetzen. Soll zum Beispiel im Fall eines unbekannten Operators das Programm eine Fehlermeldung ausgeben, so schreiben wir:

```
switch ( operator )
{
    case '+':
        System.out.println( x + y );
        break;
    case '-':
        System.out.println( x - y );
        break;
    case '*':
        System.out.println( x * y );
        break;
    case '/':
        System.out.println( x / y );
        break;
    default:
        System.err.println( "Unbekannter Operator " + operator );
}
```

Der Nutzen von **default** ist der, falsch eingegebene Operatoren zu erkennen, denn die Anweisungen hinter **default** werden immer dann ausgeführt, wenn keine case-Kon-

stante gleich dem switch-Ausdruck war. default kann auch zwischen den case-Blöcken auftauchen, doch das ist wenig übersichtlich und nicht für allgemeine Anwendungen zu empfehlen. Somit würde der default-Programmteil auch dann abgearbeitet, wenn ein dem default vorangehender case-Teil kein break hat. Nur ein default ist erlaubt.

switch hat Durchfall

Bisher haben wir in die letzte Zeile eine break-Anweisung gesetzt. Ohne ein break würden nach einer Übereinstimmung alle nachfolgenden Anweisungen ausgeführt. Sie laufen somit in einen neuen Abschnitt herein, bis ein break oder das Ende von switch erreicht ist. Da dies vergleichbar mit einem Spielzeug ist, bei dem Kugeln von oben nach unten durchfallen, nennt sich dieses auch *Fall-Through*. Ein häufiger Programmierfehler ist, das break zu vergessen, und daher sollte ein beabsichtigter Fall-Through immer als Kommentar angegeben werden.

Über dieses Durchfallen ist es möglich, bei unterschiedlichen Werten immer die gleiche Anweisung ausführen zu lassen. Das nutzt auch das nächste Beispiel, was über eine Zeichenkette läuft (mit einer Schleife und einem Konstrukt, das wir bisher noch nicht kennengelernt haben, aber damit ist das Beispiel etwas saftiger) und für jeden Buchstaben ausgibt, ob er Vokal ist oder nicht:

Listing 2.19: VowelTest.java

```
public class VowelTest
{
    public static void main( String[] args )
    {
        for ( char charToTestIfVowel : "Come at me, @TheKevinButler".toCharArray() )
        {
            switch ( charToTestIfVowel )
            {
                case 'a': case 'A':      // Fällt durch
                case 'e': case 'E':
                case 'i': case 'I':
                case 'o': case 'O':
                case 'u': case 'U':
                    System.out.println( charToTestIfVowel + " ist Vokal" );
                    break;
            }
        }
    }
}
```

```
    default:  
        System.out.println( charToTestIfVowel + " ist kein Vokal" );  
    }  
}  
}  
}  
}
```

In dem Beispiel bestimmt eine case-Anweisung, ob die Variable `charToTestIfVowel` einen Vokal enthält. Fünf case-Anweisungen decken jeweils einen Buchstaben (Vokal) ab. Stimmt der Inhalt von `charToTestIfVowel` mit einer der Vokal-Konstanten überein, so »fällt« das Programm in den Zweig mit der Ausgabe, dass `charToTestIfVowel` ein Vokal ist. Dieses Durchfallen über die case-Zweige ist sehr praktisch, da der Programmcode für die Ausgabe so nicht dupliziert werden muss. Tritt auf der anderen Seite keine Bedingung im switch-Teil ein, so gibt die Anweisung im default-Teil aus, dass `charToTestIfVowel` kein Vokal ist. Stehen mehrere case-Blöcke untereinander, um damit Bereiche abzubilden, nennt sich das auch *Stack-Case-Labels*.

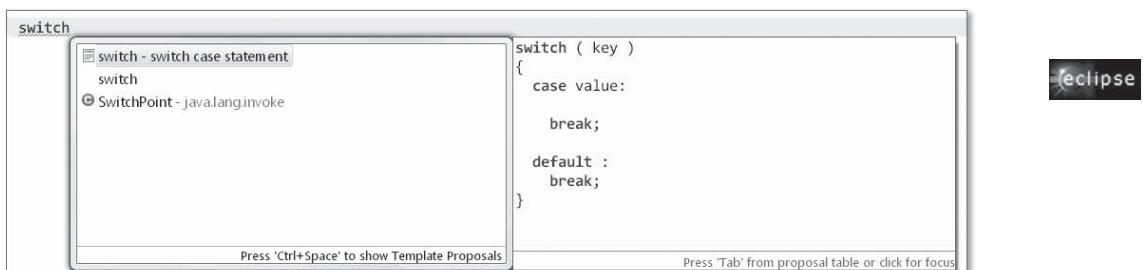


Abbildung 2.7: »switch« und `Strg` + Leertaste bietet an, ein Grundgerüst für eine switch-Fallunterscheidung anzulegen.

Hinweis

Obwohl ein fehlendes break zu lästigen Programmierfehlern führt, haben die Java-Entwickler dieses Verhalten vom syntaktischen Vorgänger C übernommen. Eine interessante Lösung wäre gewesen, das Verhalten genau umzudrehen und das Durchfallen explizit einzufordern, zum Beispiel mit einem Schlüsselwort. Dazu gibt es eine interessante Entwicklung. Java »erbt« diese Eigenschaft von C(++), die wiederum erbt es von der Programmiersprache B. Einer der »Erfinder« von B ist Ken Thompson, der heute bei Google arbeitet und an der neuen Programmiersprache Go beteiligt ist. In Go müssen Entwickler ausdrücklich die fallthrough-Anweisung verwenden, wenn ein case-Block zum nächsten weiterleiten soll.

switch auf Strings

Seit Java 7 sind switch-Anweisungen auf String-Objekten möglich.

Listing 2.20: SweetsLover.java, main()

```
String input = javax.swing.JOptionPane.showInputDialog( "Eingabe" );

switch ( input.toLowerCase() )
{
    case "kekse":
        System.out.println( "Ich mag Keeeekse" );
        break;
    case "kuchen":
        System.out.println( "Ich mag Kuchen" );
        break;
    case "schokolade":
    case "lakritze":
        System.out.println( "Hm. lecker" );
        break;
    default:
        System.out.printf( "Kann man %s essen?", input );
}
```

Obwohl Zeichenkettenvergleiche nun möglich sind, fallen Überprüfungen auf reguläre Ausdrücke leider heraus, die insbesondere Skriptsprachen anbieten.

Wie auch beim switch mit Ganzzahlen können die Zeichenketten beim String-case-Zweig aus finalen (also nicht änderbaren) Variablen stammen. Ist etwa String KEKSE = "kekse"; vordefiniert, ist case KEKSE: erlaubt.

2.6 Schleifen

Schleifen dienen dazu, bestimmte Anweisungen immer wieder abzuarbeiten. Zu einer Schleife gehören die Schleifenbedingung und der Rumpf. Die Schleifenbedingung, ein boolescher Ausdruck, entscheidet darüber, unter welcher Bedingung die Wiederholung ausgeführt wird. In Abhängigkeit von der Schleifenbedingung kann der Rumpf mehrmals ausgeführt werden. Dazu wird bei jedem Schleifendurchgang die Schleifenbedin-

gung geprüft. Das Ergebnis entscheidet, ob der Rumpf ein weiteres Mal durchlaufen (`true`) oder die Schleife beendet wird (`false`). Java bietet vier Typen von Schleifen:

- while-Schleife
- do-while-Schleife
- einfache for-Schleife
- erweiterte for-Schleife (auch *For-Each Loop* genannt)

Die ersten drei Schleifentypen erklären die folgenden Abschnitte, während die erweiterte for-Schleife nur bei Sammlungen nötig ist und daher später bei Feldern (siehe Kapitel 3, »Klassen und Objekte«) und dynamischen Datenstrukturen (siehe Kapitel 13, »Datenstrukturen und Algorithmen«) Erwähnung findet.

2.6.1 Die while-Schleife

Die while-Schleife ist eine abweisende Schleife, die vor jedem Schleifeneintritt die Schleifenbedingung prüft. Ist die Bedingung wahr, führt sie den Rumpf aus, andernfalls beendet sie die Schleife. Wie bei `if` muss auch bei den Schleifen der Typ der Bedingungen `boolean` sein.³¹

Beispiel

zB

Zähle von 100 bis 40 in Zehnerschritten herunter:

Listing 2.21: WhileLoop.java, main()

```
int cnt = 100;
while ( cnt >= 40 )
{
    System.out.printf( "Ich erblickte das Licht der Welt " +
                       "in Form einer %d-Watt-Glühbirne.%n", cnt );
    cnt -= 10;
}
```

Vor jedem Schleifendurchgang wird der Ausdruck neu ausgewertet, und ist das Ergebnis `true`, so wird der Rumpf ausgeführt. Die Schleife ist beendet, wenn das Ergebnis `false`

³¹ Wir hatten das Thema bei `if` schon angesprochen: In C(++) ließe sich `while (i)` schreiben, was in Java `while (i != 0)` wäre.

ist. Ist die Bedingung schon vor dem ersten Eintritt in den Rumpf nicht wahr, so wird der Rumpf erst gar nicht durchlaufen.



Hinweis

Wird innerhalb des Schleifenkopfs schon alles Interessante erledigt, so muss trotzdem eine Anweisung folgen. Dies ist der passende Einsatz für die leere Anweisung »;« oder den leeren Block »{}«.

```
while ( ! new java.io.File( "c:/dump.bin" ).exists() )
;
```

Nur wenn die Datei existiert, läutet dies das Ende der Schleife ein; andernfalls folgt sofort ein neuer Existenztest. Tipp an dieser Stelle: Anstatt direkt zum nächsten Dateitest überzugehen, sollte eine kurze Verzögerung eingebaut werden.

Endlosschleifen

Ist die Bedingung einer while-Schleife immer wahr, dann handelt es sich um eine Endlosschleife. Die Konsequenz ist, dass die Schleife endlos wiederholt wird:

Listing 2.22: WhileTrue.java

```
public class WhileTrue
{
    public static void main( String[] args )
    {
        while ( true )
        {
            // immer wieder und immer wieder
        }
    }
}
```

Endlosschleifen bedeuten normalerweise das Aus für jedes Programm. Doch es gibt Hilfe! Aus dieser Endlosschleife können wir mittels break entkommen; das schauen wir uns in Abschnitt 2.6.5 genauer an. Genau genommen beenden aber auch nicht aufgefangene Exceptions oder auch System.exit() die Programme.

In Eclipse lassen sich Programme von außen beenden. Dazu bietet die Ansicht CONSOLE eine rote Schaltfläche in Form eines Quadrats, die nach der Aktivierung jedes laufende Programm beendet.

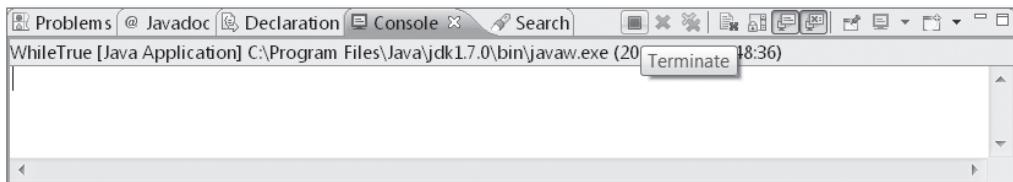


Abbildung 2.8: Die Ansicht »Console« mit der Schaltfläche zum Beenden von Programmen.

2.6.2 Die do-while-Schleife

Dieser Schleifentyp ist eine annehmende Schleife, da do-while die Schleifenbedingung erst nach jedem Schleifendurchgang prüft. Bevor es zum ersten Test kommt, ist der Rumpf also schon einmal durchlaufen worden. Der Schleifentyp hilft uns bei unserem Zahlenratespiel perfekt, denn es gibt ja mindestens einen Durchlauf mit einer Eingabe, und nur dann, wenn der Benutzer eine falsche Zahl eingibt, soll der Rumpf wiederholt werden.

Listing 2.23: TheFinalGuess.java

```
public class TheFinalGuess
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);
        int guess;

        do
        {
            System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
            guess = new java.util.Scanner( System.in ).nextInt();

            if ( number == guess )
                System.out.println( "Super getippt!" );
            else if ( number > guess )
                System.out.println( "Nee, meine Zahl ist größer als deine!" );
        }
    }
}
```

```

        else if ( number < guess )
            System.out.println( "Nee, meine Zahl ist kleiner als deine!" );
    }
    while ( number != guess );
}
}

```

Es ist wichtig, auf das Semikolon hinter der `while`-Anweisung zu achten. Liefert die Bedingung ein `true`, so wird der Rumpf erneut ausgeführt.³² Andernfalls wird die Schleife beendet, und das Programm wird mit der nächsten Anweisung nach der Schleife fortgesetzt. Interessant ist das Detail, dass wir die Variable `guess` nun außerhalb des `do-while`-Blockes deklarieren müssen, da eine im Schleifenblock deklarierte Variable für den Wiederholungstest in `while` nicht sichtbar ist. Auch weiß der Compiler, dass der `do-while`-Block mindestens einmal durchlaufen wird und `guess` auf jeden Fall initialisiert wird; der Zugriff auf nicht initialisierte Variablen ist verboten und wird vom Compiler als Fehler angesehen.

Äquivalenz einer `while`- und einer `do-while`-Schleife *

Die `do-while`-Schleife wird seltener gebraucht als die `while`-Schleife. Dennoch lassen sich beide ineinander überführen. Zunächst der erste Fall: Wir ersetzen eine `while`-Schleife durch eine `do-while`-Schleife:

```

while ( Ausdruck )
    Anweisung

```

Führen wir uns noch einmal vor Augen, was hier passiert. In Abhängigkeit vom Ausdruck wird der Rumpf ausgeführt. Da zunächst ein Test kommt, wäre die `do-while`-Schleife schon eine Blockausführung weiter. So fragen wir in einem ersten Schritt mit einer `if`-Anweisung ab, ob die Bedingung wahr ist oder nicht. Wenn ja, dann lassen wir den Programmcode in einer `do-while`-Schleife abarbeiten.

³² Das ist in Pascal und Delphi anders. Hier läuft eine Schleife der Bauart `repeat ... until` Bedingung (das Gegenstück zu Javas `do-while`) so lange, *bis* die Bedingung wahr wird, und bricht dann ab – ist die Bedingung nicht erfüllt, also falsch, geht es weiter mit einer Wiederholung. Ist in Java die Bedingung *nicht* erfüllt, bedeutet es das Ende der Schleifendurchläufe; das ist also genau das Gegenteil. Die Schleife vom Typ `while` Bedingung ... `do` in Pascal und Delphi entspricht aber genau der `while`-Schleife in Java.

Die äquivalente do-while-Schleife sieht also wie folgt aus:

```
if ( Ausdruck )
    do
        Anweisung
    while ( Ausdruck );
```

Nun der zweite Fall: Wir ersetzen die do-while-Schleife durch eine while-Schleife:

```
do
    Anweisung
    while ( Ausdruck );
```

Da zunächst die Anweisungen ausgeführt werden und anschließend der Test, schreiben wir für die while-Variante die Ausdrücke einfach vor den Test. So ist sichergestellt, dass diese zumindest einmal abgearbeitet werden:

```
Anweisung
while ( Ausdruck )
    Anweisung
```

2.6.3 Die for-Schleife

Die for-Schleife ist eine spezielle Variante einer while-Schleife und wird typischerweise zum Zählen benutzt. Genauso wie while-Schleifen sind for-Schleifen abweisend, der Rumpf wird also erst dann ausgeführt, wenn die Bedingung wahr ist.

Beispiel

zB

Gib die Zahlen von 1 bis 10 auf dem Bildschirm aus:

Listing 2.24: ForLoop.java, main()

```
for ( int i = 1; i <= 10; i++ )           // i ist Schleifenzähler
    System.out.println( i );
```

Eine genauere Betrachtung der Schleife zeigt die unterschiedlichen Segmente:

- *Initialisierung der Schleife:* Der erste Teil der for-Schleife ist ein Ausdruck wie $i = 1$, der vor der Durchführung der Schleife genau einmal ausgeführt wird. Dann wird das Ergebnis verworfen. Tritt in der Auswertung ein Fehler auf, so wird die Abarbeitung unterbrochen, und die Schleife kann nicht vollständig ausgeführt werden. Der erste

Teil kann lokale Variablen deklarieren und initialisieren. Diese Zählvariable ist dann außerhalb des Blocks nicht mehr gültig.³³ Es darf noch keine lokale Variable mit dem gleichen Namen geben.

- **Schleifentest/Schleifenbedingung:** Der mittlere Teil, wie `i <= 10`, wird vor dem Durchlaufen des Schleifenrumpfs – also vor jedem Schleifeneintritt – getestet. Ergibt der Ausdruck `false`, wird die Schleife nicht durchlaufen und beendet. Das Ergebnis muss, wie bei einer `while`-Schleife, vom Typ `boolean` sein. Ist kein Test angegeben, so ist das Ergebnis automatisch `true`.
- **Schleifen-Inkrement durch einen Fortschaltungsausdruck:** Der letzte Teil, wie `i++`, wird immer am Ende jedes Schleifendurchlaufs, aber noch vor dem nächsten Schleifeneintritt ausgeführt. Das Ergebnis wird nicht weiter verwendet. Ergibt die Bedingung des Tests `true`, dann befindet sich beim nächsten Betreten des Rumpfs der veränderte Wert im Rumpf.

Betrachten wir das Beispiel, so ist die Auswertungsreihenfolge folgender Art:

1. Initialisiere `i` mit 1.
2. Teste, ob `i <= 10` gilt.
3. Ergibt sich `true`, dann führe den Block aus, sonst ist es das Ende der Schleife.
4. Erhöhe `i` um 1.
5. Gehe zu Schritt 2.

Schleifenzähler

Wird die `for`-Schleife zum Durchlaufen einer Variablen genutzt, so heißt der *Schleifenzähler* entweder *Zählvariable* oder *Laufvariable*.

Wichtig sind die Initialisierung und die korrekte Abfrage am Ende. Schnell läuft die Schleife einmal zu oft durch und führt so zu falschen Ergebnissen. Die Fehler bei der Abfrage werden auch *off-by-one error* genannt, wenn zum Beispiel statt `<=` der Operator `<` steht. Dann nämlich läuft die Schleife nur bis 9. Ein anderer Name für den Schleifenfehler lautet *fencepost error* (Zaunpfahl-Fehler). Es geht um die Frage, wie viele Pfähle für

³³ Im Gegensatz zu C++ ist das Verhalten klar definiert, und es gibt kein Hin und Her. In C++ implementierten Compilerbauer die Variante einmal so, dass die Variable nur im Block galt, andere interpretierten die Sprachspezifikation so, dass diese auch außerhalb gültig blieb. Die aktuelle C++-Definition schreibt nun vor, dass die Variable außerhalb des Blocks nicht mehr gültig ist. Da es jedoch noch alten Programmcode gibt, haben viele Compilerbauer eine Option eingebaut, mit der das Verhalten der lokalen Variablen bestimmt werden kann.

einen 100 m langen Zaun nötig sind, sodass alle Pfähle einen Abstand von 10 m haben:
9, 10 oder 11?

Wann for- und wann while-Schleife?

Da sich die while- und die for-Schleife sehr ähnlich sind, ist die Frage berechtigt, wann die eine und wann die andere zu nutzen ist. Leider verführt die kompakte for-Schleife sehr schnell zu einer Überladung. Manche Programmierer packen gerne alles in den Schleifenkopf hinein, und der Rumpf besteht nur aus einer leeren Anweisung. Dies ist ein schlechter Stil und sollte vermieden werden.

for-Schleifen sollten immer dann benutzt werden, wenn eine Variable um eine konstante Größe erhöht wird. Tritt in der Schleife keine Schleifenvariable auf, die inkrementiert oder dekrementiert wird, sollte eine while-Schleife genutzt werden. Eine do-while-Schleife sollte dann eingesetzt werden, wenn die Abbruchbedingung erst am Ende eines Schleifendurchlaufs ausgewertet werden kann. Auch sollte die for-Schleife dort eingesetzt werden, wo sich alle drei Ausdrücke im Schleifenkopf auf dieselbe Variable beziehen. Vermieden werden sollten unzusammenhängende Ausdrücke im Schleifenkopf. Der schreibende Zugriff auf die Schleifenvariable im Rumpf ist eine schlechte Idee, wenn sie auch gleichzeitig im Kopf modifiziert wird – das ist schwer zu durchschauen und kann leicht zu Endlosschleifen führen.

Die for-Schleife ist nicht auf einen bestimmten Typ festgelegt, auch wenn for-Schleifen für das Hochzählen den impliziten Typ int suggerieren. Der Initialisierungsteil kann alles Mögliche vorbelegen, ob int, double oder eine Referenzvariable. Die Bedingung kann alles erdenkbare testen, nur das Ergebnis muss hier ein boolean sein.

Eine Endlosschleife mit for

Da alle drei Ausdrücke im Kopf der Schleife optional sind, können sie weggelassen werden, und es ergibt sich eine Endlosschleife. Diese Schreibweise ist somit mit while(true) semantisch äquivalent:

```
for ( ; ; )  
;
```

Die trennenden Semikola dürfen nicht verschwinden. Falls demnach keine Schleifenbedingung angegeben ist, ist der Ausdruck immer wahr. Es folgt keine Initialisierung und keine Auswertung des Fortschaltausdrucks.

Geschachtelte Schleifen

Schleifen, und das gilt insbesondere für `for`-Schleifen, können verschachtelt werden. Syntaktisch ist das auch logisch, da sich innerhalb des Schleifenrumpfs beliebige Anweisungen aufhalten dürfen. Um fünf Zeilen von Sternchen auszugeben, wobei in jeder Zeile immer ein Stern mehr erscheinen soll, schreiben wir:

```
for ( int i = 1; i <= 5; i++ )
{
    for ( int j = 1; j <= i; j++ )
        System.out.print( "*" );
    System.out.println();
}
```

Als besonderes Element ist die Abhängigkeit des Schleifenzählers `j` von `i` zu werten. Es folgt die Ausgabe:

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

Die übergeordnete Schleife nennt sich *äußere Schleife*, die untergeordnete *innere Schleife*. In unserem Beispiel wird die äußere Schleife die Zeilen zählen und die innere die Sternchen in eine Zeile ausgeben, also für die Spalte verantwortlich sein.

Da Schleifen beliebig tief geschachtelt werden können, muss besonderes Augenmerk auf die Laufzeit gelegt werden. Die inneren Schleifen werden immer so oft ausgeführt, wie die äußere Schleife durchlaufen wird.

for-Schleifen und ihr Komma-Operator *

Im ersten und letzten Teil einer `for`-Schleife lässt sich ein Komma einsetzen. Damit lassen sich entweder mehrere Variablen gleichen Typs deklarieren – wie wir es schon kennen – oder mehrere Ausdrücke nebeneinander schreiben, aber keine beliebigen Anweisungen oder sogar andere Schleifen. Mit den Variablen `i` und `j` können wir so eine kleine Multiplikationstabelle aufbauen:

```
for ( int i = 1, j = 9; i <= j; i++, j-- )
    System.out.printf( "%d * %d = %d%n", i, j, i*j );
```

Dann ist die Ausgabe:

```
1 * 9 = 9
2 * 8 = 16
3 * 7 = 21
4 * 6 = 24
5 * 5 = 25
```

Ein weiteres Beispiel mit komplexerer Bedingung wäre das folgende, das vor dem Schleifendurchlauf den Startwert für die Variablen `x` und `y` initialisiert, dann `x` und `y` heraufsetzt und die Schleife so lange ausführt, bis `x` und `y` beide 10 sind:

```
int x, y;
for ( x = initX(), y = initY(), x++, y++;
      ! (x == 10 && y == 10);
      x += xinc(), y += yinc() )
{
    // ...
}
```

Tipp

Komplizierte `for`-Schleifen sind lesbarer, wenn die drei `for`-Teile in getrennten Zeilen stehen.



2.6.4 Schleifenbedingungen und Vergleiche mit ==

Eine Schleifenabbruchbedingung kann ganz unterschiedlich aussehen. Beim Zählen ist es häufig der Vergleich auf einen Endwert. Oft steckt an dieser Stelle ein absoluter Vergleich mit `==`, der aus zwei Gründen problematisch werden kann.

Sehen wir uns das erste Problem anhand einiger Programmzeilen an:

```
for ( int i = 1; i != 11; i++ )          // Zählt bis 10, oder?
    System.out.println( i );
```

Ist der Wert der Variablen `i` kleiner als 11, so haben wir beim Zählen kein Problem, denn dann ist anschließend spätestens bei 11 Schluss und die Schleife bricht ab. Komt der Wert aber aus einer unbekannten Quelle und ist er echt größer als 11, so ist die Bedingung ebenso wahr, und der Schleifenrumpf wird ziemlich lange durchlaufen – genau

genommen so weit, bis wir durch einen Überlauf wieder bei 0 beginnen und dann auch bei 11 und dem Abbruch landen. Die Absicht ist sicherlich eine andere gewesen. Die Schleife sollte nur so lange zählen, wie *i kleiner 11* ist, und nicht einfach nur *ungleich 11*. Daher passt Folgendes besser:

```
for ( int i = 1; i < 11; i++ )           // Zählt immer nur bis 10
    System.out.println( i );
```

Jetzt rennt der Interpreter bei Zahlen größer 11 nicht endlos weiter, sondern stoppt die Schleife sofort ohne Durchlauf.

Rechenun genauigkeiten

Das zweite Problem ergibt sich bei Fließkommazahlen. Es ist sehr problematisch, echte Vergleiche zu fordern:

```
double d = 0.0;
while ( d != 1.0 )                  // Achtung! Problematischer Vergleich!
{
    d += 0.1;
    System.out.println( d );
}
```

Lassen wir das Programmsegment laufen, so sehen wir, dass die Schleife hurtig über das Ziel hinausschießt:

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
```

Und das so lange, bis das Auge müde wird ...

Bei Fließkommawerten bietet es sich daher immer an, mit den relationalen Operatoren `<`, `>`, `<=` oder `>=` zu arbeiten.

Eine zweite Möglichkeit neben dem echten Kleiner/Größer-Vergleich ist, eine erlaubte Abweichung (Delta) zu definieren. Mathematiker bezeichnen die Abweichung von zwei Werten mit dem griechischen Kleinbuchstaben Epsilon. Wenn wir einen Vergleich von zwei Fließkommazahlen anstreben und bei einem Gleichheitsvergleich eine Toleranz mit betrachten wollen, so schreiben wir einfach:

```
if ( Math.abs(x - y) <= epsilon )
    ...

```

Epsilon ist die erlaubte Abweichung. `Math.abs(x)` berechnet von einer Zahl `x` den Absolutwert.

Wie Bereichsangaben schreiben? *

Für Bereichsangaben der Form `a >= 23 && a <= 42` empfiehlt es sich, den unteren Wert in den Vergleich einzubeziehen, den Wert für die obere Grenze jedoch nicht (inklusive untere Grenzen und exklusive obere Grenzen). Für unser Beispiel, in dem `a` im Intervall bleiben soll, ist Folgendes besser: `a >= 23 && a < 43`. Die Begründung dafür ist einleuchtend:

- Die Größe des Intervalls ist die Differenz aus den Grenzen.
- Ist das Intervall leer, so sind die Intervallgrenzen gleich.
- Die untere Grenze ist nie größer als die obere Grenze.

Hinweis

Die Standardbibliothek verwendet diese Konvention auch durchgängig, etwa im Fall von `substring()` bei String-Objekten oder `subList()` bei Listen oder bei der Angabe von Array-Indexwerten.

Die Vorschläge können für normale Schleifen mit Vergleichen übernommen werden. So ist eine Schleife mit zehn Durchgängen besser in der Form

```
for ( i = 0; i < 10; i++ )           // Besser
```

formuliert als in der semantisch äquivalenten Form:

```
for ( i = 0; i <= 9; i++ )          // Nicht so gut
```

2.6.5 Ausbruch planen mit break und Wiedereinstieg mit continue

Wird innerhalb einer for-, while- oder do-while-Schleife eine break-Anweisung eingesetzt, so wird der Schleifendurchlauf beendet und die Abarbeitung bei der ersten Anweisung nach der Schleife fortgeführt.

Dass eine Endlossschleife mit break beendet werden kann ist nützlich, wenn eine Bedingung eintritt, die das Ende der Schleife bestimmt. Das lässt sich prima auf unser Zahlenratespiel übertragen:

Listing 2.25: GuessWhat.java

```
public class GuessWhat
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);

        while ( true )
        {
            System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
            int guess = new java.util.Scanner( System.in ).nextInt();

            if ( number == guess )
            {
                System.out.println( "Super getippt!" );
                break;                      // Ende der Schleife
            }
            else if ( number > guess )
                System.out.println( "Nee, meine Zahl ist größer als deine!" );
            else if ( number < guess )
                System.out.println( "Nee, meine Zahl ist kleiner als deine!" );
        }
    }
}
```

Die Fallunterscheidung stellt fest, ob der Benutzer noch einmal in einem weiteren Schleifendurchlauf neu raten muss oder ob der Tipp richtig war; dann beendet die break-Anweisung den Spuk.

Tipp

Da ein kleines `break` schnell im Programmtext verschwindet, seine Bedeutung aber groß ist, sollte ein kleiner Hinweis auf diese Anweisung gesetzt werden.

**Flaggen oder break**

`break` lässt sich gut verwenden, um aus einer Schleife vorzeitig auszubrechen, ohne Flags zu benutzen. Dazu ein Beispiel dafür, was vermieden werden sollte:

```
boolean endFlag = false;
do
{
    if ( Bedingung )
    {
        ...
        endFlag = true;
    }
} while ( AndereBedingung && ! endFlag );
```

Stattdessen schreiben wir:

```
do
{
    if ( Bedingung )
    {
        ...
        break;
    }
} while ( AndereBedingung );
```

Die alternative Lösung stellt natürlich einen Unterschied dar, wenn nach dem `if` noch Anweisungen in der Schleife stehen.

continue

Innerhalb einer `for`-, `while`- oder `do-while`-Schleife lässt sich eine `continue`-Anweisung einsetzen, die nicht wie `break` die Schleife beendet, sondern zum Schleifenkopf zurück geht. Nach dem Auswerten des Fortschaltausdrucks wird im nächsten Schritt erneut

geprüft, ob die Schleife weiter durchlaufen werden soll. Ein häufiges Einsatzfeld sind Schleifen, die im Rumpf immer wieder Werte so lange holen und testen, bis diese für die Weiterverarbeitung geeignet sind.

Dazu ein Beispiel wieder mit dem Ratespiel. Dem Benutzer wird bisher mitgeteilt, dass er nur Zahlen zwischen 1 und 5 (inklusive) eingeben soll, aber wenn er -1234567 eingibt, ist das auch egal. Das wollen wir ändern, indem wir einen Test vorschalten, der zurück zur Eingabe führt, wenn der Wertbereich falsch ist. `continue` hilft uns dabei, zurück zum Anfang des Blockes zu kommen, und der beginnt mit einer neuen Eingabeaufforderung.

Listing 2.26: GuessRight.java

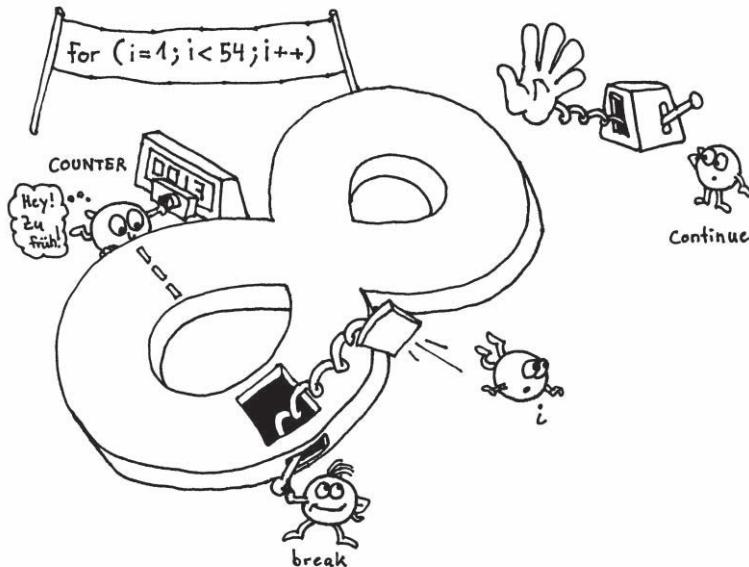
```
public class GuessRight
{
    public static void main( String[] args )
    {
        int number = (int) (Math.random() * 5 + 1);

        while ( true )
        {
            System.out.println( "Welche Zahl denke ich mir zwischen 1 und 5?" );
            int guess = new java.util.Scanner( System.in ).nextInt();

            if ( guess < 1 || guess > 5 )
            {
                System.out.println( "Nur Zahlen zwischen 1 und 5!" );
                continue;
            }

            if ( number == guess )
            {
                System.out.println( "Super getippt!" );
                break; // Ende der Schleife
            }
            else if ( number > guess )
                System.out.println( "Nee, meine Zahl ist größer als deine!" );
        }
    }
}
```

```
        else if ( number < guess )
            System.out.println( "Nee, meine Zahl ist kleiner als deine!" );
    }
}
}
```



Manche Programmstücke sind aber ohne `continue` lesbarer. Ein `continue` am Ende einer `if`-Abfrage kann durch einen `else`-Teil bedeutend klarer gefasst werden. Zunächst das schlechte Beispiel:

```
while ( Bedingung )          // Durch continue verzuckert
{
    if ( AndereBedingung )
    {
        // Code,Code,Code
        continue;
    }
    // Weiterer schöner Code
}
```

Viel deutlicher ist:

```
while ( Bedingung )
{
    if ( AndereBedingung )
    {
        // Code, Code, Code
    }
    else
    {
        // Weiterer schöner Code
    }
}
```

2.6.6 break und continue mit Marken *

Obwohl das Schlüsselwort `goto` in der Liste der reservierten Wörter auftaucht, erlaubt Java keine beliebigen Sprünge, und `goto` ist ohne Funktionalität. Allerdings lassen sich in Java Anweisungen – oder ein Block, der eine besondere Anweisung ist – markieren. Ein Grund für die Einführung von Markierungen ist der, dass `break` bzw. `continue` mehrdeutig ist:

- Wenn es zwei ineinander verschachtelte Schleifen gibt, würde ein `break` in der inneren Schleife nur die innere abbrechen. Was ist jedoch, wenn die äußere Schleife beendet werden soll? Das Gleiche gilt für `continue`, wenn die äußere Schleife weiter vorgesetzt werden soll und nicht die innere.
- Nicht nur Schleifen nutzen das Schlüsselwort `break`, sondern auch die `switch`-Anweisung. Was ist, wenn eine Schleife eine `switch`-Anweisung enthält, doch nicht der lokale `case`-Zweig mit `break` beendet werden soll, sondern die ganze Schleife mit `break` abgebrochen werden soll?

Die Sprachdesigner von Java haben sich dazu entschlossen, Markierungen einzuführen, sodass `break` und `continue` die markierte Anweisung entweder verlassen oder wieder durchlaufen können. Falsch eingesetzt, können sie natürlich zu Spaghetti-Code wie aus der Welt der unstrukturierten Programmiersprachen führen. Doch als verantwortungsvolle Java-Programmierer werden wir das Feature natürlich nicht missbrauchen.

break mit einer Marke für Schleifen

Betrachten wir ein erstes Beispiel mit einer Marke (engl. *label*), in dem `break` nicht nur aus der inneren Teufelsschleife ausbricht, sondern aus der äußeren gleich mit. Marken werden definiert, indem ein Bezeichner mit Doppelpunkt abgeschlossen und vor eine Anweisung gesetzt wird – die Anweisung wird damit markiert wie eine Schleife:

Listing 2.27: BreakAndContinueWithLabels.java, main()

```
heaven:  
while ( true )  
{  
    hell:  
        while ( true )  
        {  
            break /* continue */ heaven;  
        }  
        // System.out.println( "hell" );  
    }  
    System.out.println( "heaven" );
```

Ein `break` ohne Marke in der inneren `while`-Schleife beendet nur die innere Wiederholung, und ein `continue` würde zur Fortführung dieser inneren `while`-Schleife führen. Unser Beispiel zeigt die Anwendung einer Marke hinter den Schlüsselwörtern `break` und `continue`.

Das Beispiel benutzt die Marke `hell` nicht, und die Zeile mit der Ausgabe »hell« ist bewusst ausgeklammert, denn sie ist nicht erreichbar und würde andernfalls zu einem Compilerfehler führen. Dass die Anweisung nicht erreichbar ist, ist klar, denn mit einem `break heaven` kommt das Programm nie zur nächsten Anweisung hinter der inneren Schleife, und somit ist eine Konsolenausgabe nicht erreichbar.

Setzen wir statt `break heaven` ein `break hell` in die innere Schleife, ändert sich dies:

```
heaven:  
while ( true )  
{  
    hell:  
        while ( true )  
        {  
            break /* continue */ hell;
```

```

    }
    System.out.println( "hell" );
}
// System.out.println( "heaven" );

```

In diesem Szenario ist die Ausgabe »heaven« nicht erreichbar und muss auskommert werden. Das `break` in der inneren Schleife wirkt wie ein einfaches `break` ohne Marke, und das ablaufende Programm führt laufend zu Bildschirmausgaben von »hell«.



Rätsel

Warum übersetzt der Compiler Folgendes ohne Murren?

Listing 2.28: WithoutComplain.java

```

class WithoutComplain
{
    static void main( String[] args )
    {
        http://www.tutego.de/
        System.out.print( "Da gibt's Java-Tipps und -Tricks." );
    }
}

```

Mit dem `break` und einer Marke aus dem `switch` aussteigen

Da dem `break` mehrere Funktionen in der Sprache Java zukommen, kommt es zu einer Mehrdeutigkeit, wenn im `case`-Block einer `switch`-Anweisung ein `break` eingesetzt wird.

Im folgenden Beispiel läuft eine Schleife einen String ab. Der Zugriff auf ein Zeichen im String realisiert die String-Objektmethode `charAt()`; die Länge eines Strings liefert `length()`. Als Zeichen im String sollen C, G, A, T erlaubt sein. Für eine Statistik über die Anzahl der einzelnen Buchstaben zählt eine `switch`-Anweisung beim Treffer jeweils die richtige Variable `c, g, a, t` um 1 hoch. Falls ein falsches Zeichen im String vorkommt, wird die Schleife beendet. Und genau hier bekommt die Markierung ihren Auftritt:

Listing 2.29: SwitchBreak.java

```

public class SwitchBreak
{
    public static void main( String[] args )

```

```
{  
    String dnaBases = "CGCAGTTCTTCGGXAC";  
    int a = 0, g = 0, c = 0, t = 0;  
  
    loop:  
        for ( int i = 0; i < dnaBases.length(); i++ )  
        {  
            switch ( dnaBases.charAt( i ) )  
            {  
                case 'A': case 'a':  
                    a++;  
                    break;  
                case 'G': case 'g':  
                    g++;  
                    break;  
                case 'C': case 'c':  
                    c++;  
                    break;  
                case 'T': case 't':  
                    t++;  
                    break;  
                default:  
                    System.err.println( "Unbekannte Nukleinbasen " + dnaBases.charAt( i ) );  
                    break loop;  
            }  
        }  
        System.out.printf( "Anzahl: A=%d, G=%d, C=%d, T=%d%n", a, g, c, t );  
    }  
}
```

Hinweis

Marken können vor allen Anweisungen (und Blöcke sind damit eingeschlossen) definiert werden; in unserem ersten Fall haben wir die Marke vor die while(true)-Schleife gesetzt. Interessanterweise kann ein break mit einer Marke nicht nur eine Schleife und case verlassen, sondern auch einen ganz einfachen Block:





Hinweis (Forts.)

```
label:  
{  
    ...  
    break label;  
    ...  
}
```

Somit entspricht das break label einem goto zum Ende des Blocks.

Das break kann nicht durch continue ausgetauscht werden, da continue in jedem Fall eine Schleife braucht. Und ein normales break ohne Marke wäre im Übrigen nicht gültig und könnte nicht den Block verlassen.



Rätsel

Wenn Folgendes in der main()-Methode stünde, würde es der Compiler übersetzen?

Was wäre die Ausgabe? Achte genau auf die Leerzeichen!

```
int val = 2;  
switch ( val )  
{  
    case 1:  
        System.out.println( 1 );  
  
    case2:  
        System.out.println( 2 );  
    default:  
        System.out.println( 3 );  
}
```

2.7 Methoden einer Klasse

In objektorientierten Programmen interagieren zur Laufzeit Objekte miteinander und senden sich gegenseitig Nachrichten als Aufforderung, etwas zu machen. Diese Aufforderungen resultieren in einem Methodenaufruf, in dem Anweisungen stehen, die dann ausgeführt werden. Das Angebot eines Objekts, also das, was es »kann«, wird in Java durch Methoden ausgedrückt.

Wir haben schon mindestens eine Methode kennengelernt: `println()`. Sie ist eine Methode vom `out`-Objekt. Ein anderes Programmstück schickt nun eine Nachricht an das `out`-Objekt, die `println()`-Methode auszuführen. Im Folgenden werden wir den aktiven Teil des Nachrichtenversendens nicht mehr so genau betrachten, sondern wir sagen nur noch, dass eine Methode aufgerufen wird.

Für die Deklaration von Methoden gibt es drei Gründe:

- Wiederkehrende Programmteile sollen nicht immer wieder programmiert, sondern an einer Stelle angeboten werden. Änderungen an der Funktionalität lassen sich dann leichter durchführen, wenn der Code lokal zusammengefasst ist.
- Komplexe Programme werden in kleine Teilprogramme zerlegt, damit die Komplexität des Programms heruntergebrochen wird. Damit ist der Kontrollfluss leichter zu erkennen.
- Die Operationen einer Klasse, also das Angebot eines Objekts, sind ein Grund für Methodendeklarationen in einer objektorientierten Programmiersprache. Daneben gibt es aber noch weitere Gründe, die für Methoden sprechen. Sie werden im Folgenden erläutert.

2.7.1 Bestandteil einer Methode

Eine Methode setzt sich aus mehreren Bestandteilen zusammen. Dazu gehören der *Methodenkopf* (kurz *Kopf*) und der *Methodenrumpf* (kurz *Rumpf*). Der Kopf besteht aus einem *Rückgabetyp* (auch *Ergebnistyp* genannt), dem *Methodennamen* und einer optionalen *Parameterliste*.

Nehmen wir die bekannte statische `main()`-Methode:

```
public static void main( String[] args )
{
    System.out.println( "Wie siehst du denn aus? Biste gerannt?" );
}
```

Sie hat folgende Bestandteile:

- Die statische Methode liefert keine Rückgabe, daher ist der »Rückgabetyp« `void`. (An dieser Stelle sollte bemerkt werden, dass `void` in Java kein Typ ist.) `void` heißt auf Deutsch übersetzt: »frei«, »die Leere« oder »Hohlraum«.
- Der Methodenname ist `main`.

- Die Parameterliste ist `String[] args`.
- Der Rumpf besteht nur aus der Bildschirmausgabe.



Namenskonvention

Methodennamen beginnen wie Variablennamen mit Kleinbuchstaben und werden in der gemischten Groß-/Kleinschreibung verfasst. Bezeichner dürfen nicht wie Schlüsselwörter heißen.³⁴

Die Signatur einer Methode

Der Methodename und die Parameterliste bestimmen die *Signatur* einer Methode; der Rückgabetyp gehört nicht dazu. Die Parameterliste ist durch die Anzahl, die Reihenfolge und die Typen der Parameter beschrieben. Pro Klasse darf es nur eine Methode mit derselben Signatur geben, sonst meldet der Compiler einen Fehler. Da die Methoden `void main(String[] args)` und `String main(String[] arguments)` die gleiche Signatur (`main, String[]`) besitzen – die Namen der Parameter spielen keine Rolle –, können sie nicht zusammen in einer Klasse deklariert werden (später werden wir sehen, dass Unterklassen durchaus gewisse Sonderfälle zulassen).



Duck-Typing

Insbesondere Skriptsprachen, wie Python, Ruby oder Groovy, erlauben Methodendeklarationen ohne Parametertyp, sodass die Methoden mit unterschiedlichen Argumenttypen aufgerufen werden können:

```
add( a, b ) return a + b
```

Aufgrund des nicht bestimmten Parametertyps lässt sich die Methode mit Ganzzahlen, Fließkommazahlen oder Strings aufrufen. Es ist die Aufgabe der Laufzeitumgebung, diesen dynamischen Typ zu erkennen und die Addition auf dem konkreten Typ auszuführen. In Java ist das nicht möglich. Der Typ muss überall stehen. Der Name »Duck Typing« stammt aus einem Gedicht von James Whitcomb Riley, in dem es heißt: »When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.«. Auf unseren Fall übertragen: Wenn die Parameter `a` und `b` die Operation »addieren« unterstützen, dann sind die Werte eben addierbar.

³⁴ Das führte bei manchen Bibliotheken (JUnit sei hier als Beispiel genannt) zu Überraschungen. In Java 1.4 etwa wurde das Schlüsselwort `assert` eingeführt, das JUnit als Methodename wählte. Unzählige Zeilen Programmcode mussten daraufhin von `assert()` nach `assertTrue()` konvertiert werden.

2.7.2 Signatur-Beschreibung in der Java-API

In der Java-Dokumentation sind alle Methoden mit ihren Rückgaben und Parametern genau definiert. Betrachten wir die Dokumentation der statischen Methode `max()` der Klasse `Math`:

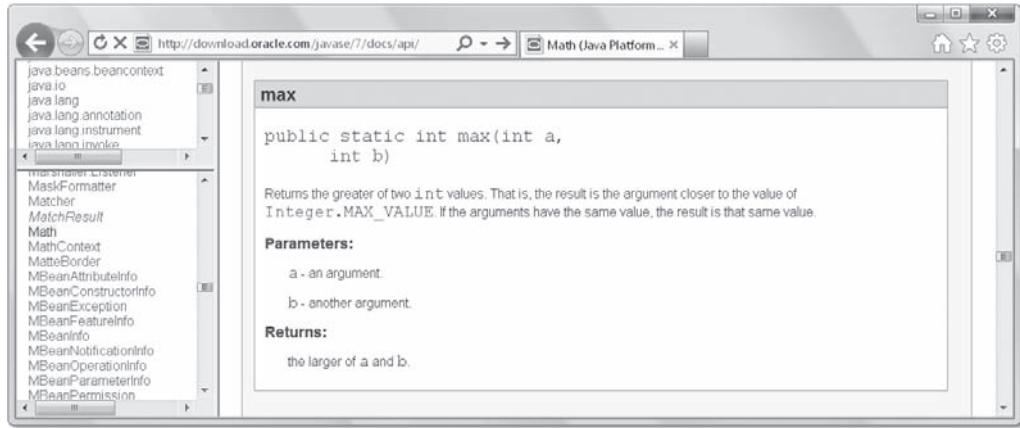


Abbildung 2.9: Die Online-API-Dokumentation für `Math.max()`

Die Hilfe gibt Informationen über die komplette Signatur der Methode. Der Rückgabewert ist ein `double`, die statische Methode heißt `max()`, und sie erwartet genau zwei `double`-Zahlen. Verschwiegen haben wir die Schlüsselwörter `public static` – die Modifizierer. `public` gibt die Sichtbarkeit an und sagt, wer diese Methode nutzen kann. Im Fall von `public` bedeutet es, dass jeder diese Methode verwenden kann. Das Gegenteil ist `private`: In dem Fall kann nur das Objekt selbst diese Methode nutzen. Das ist sinnvoll, wenn Methoden benutzt werden, um die Komplexität zu verkleinern und Teilprobleme zu lösen. Private Methoden werden in der Regel nicht in der Hilfe angezeigt, da sie ein Implementierungsdetail sind. Das Schlüsselwort `static` zeigt an, dass sich die Methode mit dem Klassennamen nutzen lässt, also kein Exemplar eines Objekts nötig ist.

Es gibt Methoden, die noch andere Modifizierer und eine erweiterte Signatur besitzen. Ein weiteres Beispiel aus der API (siehe Abbildung 2.10).

Die Sichtbarkeit dieser Methode ist `protected`. Das bedeutet: Nur abgeleitete Klassen und Klassen im gleichen Verzeichnis (Paket) können diese Methode nutzen. Ein zusätzlicher Modifizierer ist `final`, der in einer Vererbung der Unterklasse nicht erlaubt, die Methode zu überschreiben und ihr neuen Programmcode zu geben. Zum Schluss folgt hinter dem Schlüsselwort `throws` eine Ausnahme. Diese sagt etwas darüber aus, welche Fehler die Methode verursachen kann und worum sich der Programmierer kümmern

muss. Im Zusammenhang mit der Vererbung werden wir noch über `protected` und `final` sprechen. Dem Ausnahmezustand widmen wir Kapitel 6, »Exceptions«. Die Dokumentation zeigt mit »Since: JDK 1.1« an, dass es die Methode seit Java 1.1 gibt. Die Information kann auch an der Klasse festgemacht sein.

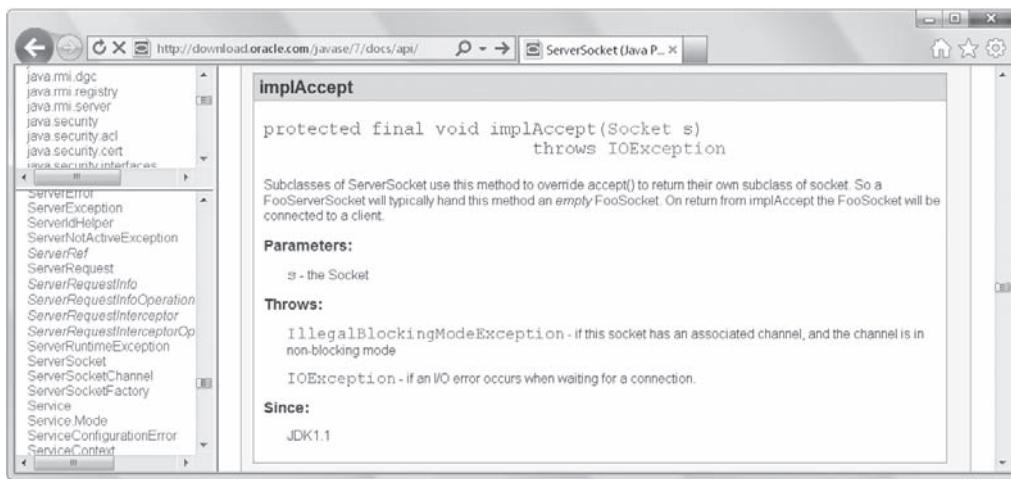


Abbildung 2.10: Ausschnitt aus der API-Dokumentation für die Klasse `java.net.ServerSocket`

2.7.3 Aufruf einer Methode

Da eine Methode immer einer Klasse oder einem Objekt zugeordnet ist, muss der Eigentümer beim Aufruf angegeben werden. Im Fall von `System.out.println()` ist `println()` eine Methode vom `out`-Objekt. Wenn wir das Maximum zweier Fließkommazahlen mit `Math.max(a, b)` bilden, dann ist `max()` eine (statische) Methode der Klasse `Math`. Für den Aufrufer ist damit immer ersichtlich, wer diese Methode anbietet, also auch, wer diese Nachricht entgegennimmt. Was der Aufrufer nicht sieht, ist die Arbeitsweise der Methode. Der Methodenaufruf verzweigt in den Programmcode, aber der Aufrufer weiß nicht, was dort geschieht. Er betrachtet nur das Ergebnis.

Die aufgerufene Methode wird mit ihrem Namen genannt. Die Parameterliste wird durch ein Klammerpaar umschlossen. Diese Klammern müssen auch dann gesetzt werden, wenn die Methode keine Parameter enthält. Eine Methode wie zum Beispiel `System.out.println()` gibt nichts als Ergebnis einer Berechnung zurück. Anders ist die statische Methode `max();` sie liefert ein Ergebnis. Damit ergeben sich vier unterschiedliche Typen von Methoden:

Methode	Ohne Rückgabewert	Mit Rückgabewert
Ohne Parameter	System.out.println()	System.currentTimeMillis()
Mit Parameter	System.out.println(4)	Math.max(12, 33)

Tabelle 2.13: Methoden mit Rückgabewerten und Parametern

Die statische Methode `System.currentTimeMillis()` gibt die Anzahl der verstrichenen Millisekunden ab dem 1.1.1970 als long zurück.

2.7.4 Methoden ohne Parameter deklarieren

Die einfachste Methode besitzt keinen Rückgabewert und keine Parameter. Der Programmcode steht in geschweiften Klammern hinter dem Kopf und bildet damit den Körper der Methode. Gibt die Methode nichts zurück, dann wird `void` vor den Methodennamen geschrieben. Falls die Methode etwas zurückgibt, wird der Typ der Rückgabe anstelle von `void` geschrieben.

Schreiben wir eine statische Methode ohne Rückgabe und Parameter, die etwas auf dem Bildschirm ausgibt:

Listing 2.30: FriendlyGreeter.java

```
class FriendlyGreeter
{
    static void greet()
    {
        System.out.println( "Guten Morgen. Oh, und falls wir uns nicht mehr" +
                            " sehen, guten Tag, guten Abend und gute Nacht!" );
    }

    public static void main( String[] args )
    {
        greet();
    }
}
```

Eigene Methoden können natürlich wie Bibliotheksmethoden heißen, da sie zu unterschiedlichen Klassen gehören. Statt `greet()` hätten wir also den Namen `println()` verwenden dürfen.

**Tipp**

Die Vergabe eines Methodennamens ist gar nicht so einfach. Nehmen wir zum Beispiel an, wir wollen eine Methode schreiben, die eine Datei kopiert. Spontan kommen uns zwei Wörter in den Sinn, die zu einem Methodennamen verbunden werden wollen: »file« und »copy«. Doch in welcher Kombination? Soll es `copyFile()` oder `fileCopy()` heißen? Wenn dieser Konflikt entsteht, sollte das Verb die Aktion anführen, unsere Wahl also auf `copyFile()` fallen. Methodennamen sollten immer das Tätigkeitswort vorne haben und das Was, das Objekt, an zweiter Stelle.



Eine gedrückte `Strg`-Taste und ein Mausklick auf einen Bezeichner lässt Eclipse zur Deklaration springen. Ein Druck auf `F3` hat den gleichen Effekt. Steht der Cursor in unserem Beispiel auf dem Methodenaufruf `greet()` und wird `F3` gedrückt, dann springt Eclipse zur Definition in Zeile 3 und hebt den Methodennamen hervor.

2.7.5 Statische Methoden (Klassenmethoden)

Bisher arbeiten wir nur mit statischen Methoden (auch Klassenmethoden genannt). Das Besondere daran ist, dass die statischen Methoden nicht an einem Objekt hängen und daher immer ohne explizit erzeugtes Objekt aufgerufen werden können. Das heißt, statische Methoden gehören zu Klassen an sich und sind nicht mit speziellen Objekten verbunden. Am Aufruf unserer statischen Methode `greet()` lässt sich ablesen, dass hier kein Objekt gefordert ist, mit dem die Methode verbunden ist. Das ist möglich, denn die Methode ist als `static` deklariert, und innerhalb der Klasse lassen sich alle Methoden einfach mit ihrem Namen nutzen.

Statische Methoden müssen explizit mit dem Schlüsselwort `static` kenntlich gemacht werden. Fehlt der Modifizierer `static`, so deklarieren wir damit eine Objektmethode, die wir nur aufrufen können, wenn wir vorher ein Objekt angelegt haben. Das heben wir uns aber bis zum nächsten Kapitel, »Klassen und Objekte«, auf. Die Fehlermeldung sollte Ihnen aber keine Angst machen. Lassen wir von der `greet()`-Deklaration das `static` weg und ruft die statische `main()`-Methode wie jetzt ohne Aufbau eines Objekts die dann nicht mehr statische Methode `greet()` auf, so gibt es den Compilerfehler »*Cannot make a static reference to the non-static method greet() from the type FriendlyGreeter*«.

Ist die statische Methode in der gleichen Klasse wie der Aufrufer deklariert – in unserem Fall `main()` und `greet()` –, so ist der Aufruf allein mit dem Namen der Methode eindeutig. Befinden sich jedoch Methodendeklaration und Methodenaufruf in unterschiedli-

chen Klassen, so muss der Aufrufer den Namen der Klasse nennen; wir haben so etwas schon einmal bei Aufrufen wie `Math.max()` gesehen.

```
class FriendlyGreeter          class FriendlyGreeterCaller
{
    static void greet()
    {
        System.out.println( "Moin!" );
    }
}

public static void main( String[] args )
{
    FriendlyGreeter.greet();
}
```



2.7.6 Parameter, Argument und Wertübergabe

Einer Methode können Werte übergeben werden, die sie dann in ihre Arbeitsweise einbeziehen kann. Der Methode `println(2001)` ist zum Beispiel ein Wert übergeben worden. Sie wird damit zur *parametrisierten Methode*.

Beispiel

zB

Werfen wir einen Blick auf die Methodendeklaration `printMax()`, die den größeren der beiden übergebenen Werte auf dem Bildschirm ausgibt.

```
static void printMax( double a, double b )
{
    if ( a > b )
        System.out.println( a );
    else
        System.out.println( b );
}
```

Um die an Methoden übergebenen Werte anzusprechen, gibt es *formale Parameter*. Von unserer statischen Methode `printMax()` sind `a` und `b` die formalen Parameter der Parameterliste. Jeder Parameter wird durch ein Komma getrennt aufgelistet, wobei für jeden Parameter der Typ angegeben sein muss; eine Kurzform wie bei der sonst üblichen Variablen-deklaration wie `double a, b` ist nicht möglich. Jede Parametervariable einer Methodendeklaration muss natürlich einen anderen Namen tragen.

Argumente (aktuelle Parameter)

Der Aufrufer der Methode muss für jeden Parameter ein Argument angeben. Die im Methodenkopf deklarierten Parameter sind letztendlich lokale Variablen im Rumpf der Methode. Beim Aufruf initialisiert die Laufzeitumgebung die lokalen Variablen mit den an die Methode übergebenen Argumenten. Rufen wir unsere parametrisierte Methode etwa mit `printMax(10, 20)` auf, so sind die Literale 10 und 20 **Argumente** (aktuelle Parameter der Methode). Beim Aufruf der Methode setzt die Laufzeitumgebung die Argumente in die lokalen Variablen, kopiert also den Wert 10 in die Parametervariable `a` und 20 in die Parametervariable `b`. Innerhalb des Methodenkörpers gibt es so Zugriff auf die von außen übergebenen Werte.

Das Ende des Blocks bedeutet automatisch das Ende für die Parametervariablen. Der Aufrufer weiß auch nicht, wie die Parametervariablen heißen. Eine Typanpassung von `int` auf `double` nimmt der Compiler in unserem Fall automatisch vor. Die Argumente müssen vom Typ her natürlich passen, und es gelten die für die Typanpassung bekannten Regeln.

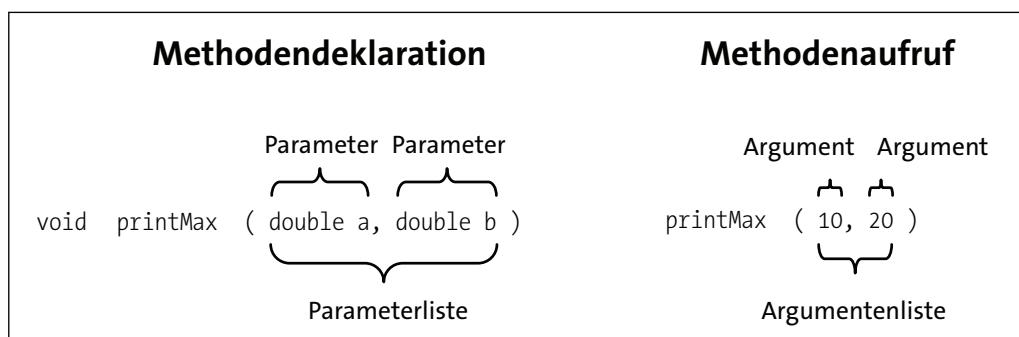


Abbildung 2.11: Die Begriffe »Parameter« und »Argument«

Wertübergabe per Call by Value

Wenn eine Methode aufgerufen wird, dann gibt es in Java ein bestimmtes Verfahren, in dem jedes Argument einer Parametervariablen übergeben wird. Diese Technik heißt *Parameterübergabemechanismus*. Viele Programmiersprachen verfügen oft über eine ganze Reihe von verwirrenden Möglichkeiten. Java kennt nur einen einfachen Mechanismus der *Wertübergabe* (engl. *call by value*, selten auch *copy by value* genannt). Ein Beispiel zum Methodenaufruf macht das deutlich:

```
int i = 2;
printMax( 10, i );
```

Unsere aufgerufene Methode `printMax(double a, double b)` bekommt zunächst 10 in die Variable `a` kopiert und dann den Inhalt der Variablen `i` (in unserem Beispiel 2) in `b`. Auf keinen Fall gibt der Aufrufer Informationen über den Speicherbereich von `i` an die Methode mit. In dem Moment, in dem die Methode aufgerufen wird, erfragt die Laufzeitumgebung die Belegung von `i` und initialisiert damit die Parametervariable `b`. Ändert `printMax()` die Variable `b`, so ändert dies nur die lokale Variable `b` (überschreibt also 2), aber die Änderung in der Methode ist für das außenstehende `i` ohne Konsequenz; `i` bleibt weiterhin bei 2. Wegen dieser Aufrufart kommt auch der Name »copy by value« zustande. Lediglich der Wert wird übergeben (kopiert) und kein Verweis auf die Variable.³⁵

Auswertung der Argumentliste von links nach rechts *

Bei einem Methodenaufruf werden erst alle Argumente ausgewertet und anschließend der Methode übergeben. Dies bedeutet im Besonderen, dass Untermethoden ausgewertet und Zuweisungen gemacht werden können. Fehler führen dann zu einem Abbruch des Methodenaufrufs. Bis zum Fehler werden alle Ausdrücke ausgewertet.

2.7.7 Methoden vorzeitig mit return beenden

Läuft eine Methode bis zum Ende durch, dann ist die Methode damit beendet, und es geht zurück zum Aufrufer. In Abhängigkeit von einer Bedingung kann eine Methode jedoch vor dem Ende des Ablaufs mit einer `return`-Anweisung beendet werden. Das ist nützlich bei Methoden, die abhängig von Parametern vorzeitig aussteigen wollen. Wir können uns vorstellen, dass vor dem Ende der Methode automatisch ein verstecktes `return` steht. Ein unnötiges `return` am Ende des Methodenkörpers sollte nicht geschrieben werden.

Beispiel

zB

Eine statische Methode `printSqrt()` soll die Wurzel einer Zahl auf dem Bildschirm ausgeben. Bei Zahlen kleiner null erscheint eine Meldung, und die Methode wird verlassen. Andernfalls wird die Wurzelberechnung durchgeführt:

³⁵ ... wie dies Referenzen in C++ tun.

zB**Beispiel (Forts.)**

```
static void printSqrt( double d )
{
    if ( d < 0 )
    {
        System.out.println( "Keine Wurzel aus negativen Zahlen!" );
        return;
    }
    System.out.println( Math.sqrt( d ) );
}
```

Die Realisierung wäre natürlich auch mit einer else-Anweisung möglich gewesen.

2.7.8 Nicht erreichbarer Quellcode bei Methoden *

Folgt direkt hinter einer return-Anweisung Quellcode, so ist dieser nicht erreichbar – im Sinne von nicht ausführbar. return beendet also immer die Methode und kehrt zum Aufrufer zurück. Folgt nach dem return noch Quelltext, meldet der Compiler einen Fehler:

```
public static void main( String[] args )
{
    int i = 1;
    return;
    i = 2;           // Unreachable code!
}
```

Reduzieren wir eine Anweisung bis auf das Nötigste, das Semikolon, so führt dies bisweilen zu amüsanten Ergebnissen:

```
public static void main( String[] args )
{
    ;return;;
}
```

Das Beispiel enthält zwei Null-Anweisungen: eine vor dem return und eine dahinter. Doch das zweite Semikolon hinter dem return ist unzulässig, da es eine nicht erreichbare Anweisung darstellt.

Tipp

In manchen Fällen ist ein `return` in der Mitte einer Methode gewollt. Soll etwa eine Methode in der Testphase nicht komplett durchlaufen, sondern in der Mitte beendet werden, so können wir uns mit einer Anweisung wie `if (true) return; retten.`



2.7.9 Methoden mit Rückgabewerten

Statische Methoden wie `Math.max()` liefern in Abhängigkeit von den Argumenten ein Ergebnis zurück. Für den Aufrufer ist die Implementierung egal; er abstrahiert und nutzt lediglich die Methode statt eines Ausdrucks. Damit Methoden Rückgabewerte an den Aufrufer liefern können, müssen zwei Dinge gelten:

- Eine Methodendeklaration bekommt einen Rückgabetyp ungleich `void`.
- Eine `return`-Anweisung gibt einen Wert zurück.

Beispiel

zB

Eine statische Methode bildet den Mittelwert und gibt diesen zurück:

```
static double avg( double x, double y )
{
    return (x + y) / 2;
}
```

Fehlt der Ausdruck, und ist es nur ein einfaches `return`, meldet der Compiler einen Programmfehler.

Obwohl einige Programmierer den Ausdruck gerne klammern, ist das nicht nötig. Klammern sollen lediglich komplexe Ausdrücke besser lesbar machen. Geklammerte Ausdrücke erinnern sonst nur an einen Methodenaufruf, und diese Verwechslungsmöglichkeit sollte bei Rückgabewerten nicht bestehen.

Der Rückgabewert muss an der Aufrufstelle nicht zwingend benutzt werden. Berechnet unsere Methode den Durchschnitt zweier Zahlen, ist es wohl eher ein Programmierfehler, den Rückgabewert nicht zu verwenden.



Was andere können

Obwohl zwar mehrere Parameter deklariert werden können, kann doch nur höchstens ein Wert an den Aufrufer zurückgegeben werden. In der Programmiersprache Python lassen sich auch mehrere Werte über ein sogenanntes Tupel zurückgeben. In Java lässt sich das über eine zurückgegebene Sammlung nachbilden.



Eclipse erkennt, ob ein Rückgabetyp fehlt, und schlägt über **Strg** + **1** einen passenden Typ vor.

Mehrere Ausstiegspunkte mit return

Für Methoden mit Rückgabewert gilt ebenso wie für void-Methoden, dass es mehr als ein `return` geben kann. Aber wird irgendein `return` »getroffen«, ist Schluss mit der Methode und es geht zurück zum Aufrufer.

Schauen wir uns dazu eine Methode an, die das Vorzeichen einer Zahl ermittelt und +1, 0 oder -1 zurückgibt, wenn die Zahl entweder positiv, null oder negativ ist:

```
static int sign( int value )
{
    if ( value < 0 )
        return -1;
    else if ( value > 0 )
        return +1;
    else
        return 0;
}
```

Bei genauer Betrachtung fällt auf, dass auf keinen Fall ein anderer Test gemacht werden kann, wenn ein `return` die Methode beendet. Der Programmcode lässt sich umschreiben, denn in `if`-Anweisungen mit weiteren `else-if`-Alternativen und Rücksprung ist die Semantik die gleiche, wenn das `else-if` durch ein einfaches `if` ersetzt wird. Eine äquivalente `sign()`-Methode wäre also:

```
static int sign( int value )
{
    if ( value < 0 )
        return -1;
    else
        return 0;
```

```

if ( value > 0 )
    return +1;
return 0;
}

```

Schummeln ohne return geht nicht

Jeder denkbare Programmfluss einer Methode mit Rückgabe muss mit einem `return` value; beendet werden. Der Compiler verfügt über ein scharfes Auge und merkt, wenn es einen Programmpfad gibt, der nicht zu einem `return`-Abschluss führt.

Beispiel

zB

Die statische Methode `isLastBitSet()` soll 0 zurückgeben, wenn das letzte Bit einer Ganzzahl nicht gesetzt ist, und 1, wenn es gesetzt ist. Den Bit-Test erledigt der Und-Operator:

```

static int isLastBitSet( int i )      // ☹ Compilerfehler
{
    switch ( i & 1 ) {
        case 0: return 0;
        case 1: return 1;
    }
}

```

Obwohl ein Bit nur gesetzt oder nicht gesetzt sein kann – dazwischen gibt es nichts –, lässt sich die Methode nicht übersetzen. Der Fehler ist: »This method must return a result of type int.«

Bei den Dingen, die für den Benutzer meistens offensichtlich sind, muss der Compiler passen, da er nicht hinter die Bedeutung sehen kann. Ähnliches würde für eine Wochen-Methode gelten, die mit einem Ganzzahl-Argument (0 bis 6) einen Wochentag als String zurückgibt. Wenn wir die Fälle 0 = Montag bis 6 = Sonntag beachten, dann kann in unseren Augen ein Wochentag nicht 99 sein. Der Compiler kennt aber die Methode nicht und weiß nicht, dass der Wertebereich beschränkt ist. Das Problem ließe sich mit einem default leicht beheben.

zB Beispiel

Die statische Methode posOrNeg() soll eine Zeichenkette mit der Information liefern, ob die übergebene Fließkommazahl positiv oder negativ ist:

```
static String posOrNeg( double d )    // ☠ Compilerfehler
{
    if ( d >= 0 )
        return "pos";

    if ( d < 0 )
        return "neg";
}
```

Überraschenderweise ist dieser Programmcode ebenfalls fehlerhaft. Denn obwohl er offensichtlich für positive oder negative Zahlen den passenden String zurückgibt, gibt es einen Fall, den diese Methode nicht abdeckt. Wieder gilt, dass der Compiler nicht erkennen kann, dass der zweite Ausdruck eine Negation des ersten sein soll. Es gibt aber noch einen zweiten Grund, der damit zu tun hat, dass es in Java spezielle Werte gibt, die keine Zahlen sind. Denn die Zahl `d` kann auch eine NaN (*Not a Number*) als Quadratwurzel aus einer negativen Zahl sein. Diesen speziellen Wert überprüft `posOrNeg()` nicht. Als Lösung für den einfachen Fall ohne NaN reicht es, aus dem zweiten `if` und der Abfrage einfach ein `else` zu machen oder die Anweisung auch gleich wegzulassen beziehungsweise mit dem Bedingungsoperator im Methodenrumpf kompakt zu schreiben: `return d >= 0 ? "pos" : "neg";`.

Methoden, die einen Fehlerwert wie 1 zurückliefern, sind häufig so implementiert, dass am Ende immer automatisch der Fehlerwert zurückgeliefert und dann in der Mitte die Methode bei passendem Ende verlassen wird.

Fallunterscheidungen mit Ausschlussprinzip *

Eine Methode `between(x, a, b)` soll testen, ob ein Wert `x` zwischen `a` (untere Schranke) und `b` (obere Schranke) liegt. Bei Methoden dieser Art ist es immer sehr wichtig, darauf zu achten und es zu dokumentieren, ob der Test auf echt kleiner (`<`) oder kleiner gleich (`<=`) durchgeführt werden soll. Wir wollen hier auch die Gleichheit betrachten.

In der Implementierung gibt es zwei Lösungen, wobei die meisten Programmierer zur ersten Lösung neigen. Die erste Lösungsidee zeigt sich in einer mathematischen Glei-

chung. Wir möchten gerne `a <= x <= b` schreiben, doch ist dies in Java nicht erlaubt.³⁶ So müssen wir einen Und-Vergleich anstellen, der etwa so lautet: Ist `a <= x && x <= b`, dann liefere `true` zurück.

Die zweite Methode zeigt, dass sich das Problem auch ohne Und-Vergleich durch das Ausschlussprinzip lösen lässt:

```
static boolean between( int x, int a, int b )
{
    if ( x < a )
        return false;

    if ( x <= b )
        return true;

    return false;
}
```

Mit verschachtelten Anfragen sieht das dann so aus:

```
static boolean between( int x, int a, int b )
{
    if ( a <= x )
        if ( x <= b )
            return true;

    return false;
}
```

2.7.10 Methoden überladen

Eine Methode ist gekennzeichnet durch Rückgabewert, Name, Parameter und unter Umständen durch Ausnahmefehler, die sie auslösen kann. Java erlaubt es, den Namen der Methode beizubehalten, aber andere Parameter einzusetzen. Eine *überladene Methode* ist eine Methode mit dem gleichen Namen wie eine andere Methode, aber einer unterschiedlichen Parameterliste. Das ist auf zwei Arten möglich:

³⁶ ... im Gegensatz zur Programmiersprache Python.

- Eine Methode heißt gleich, akzeptiert aber eine unterschiedliche Anzahl von Argumenten.
- Eine Methode heißt gleich, hat aber für den Compiler unterscheidbare Parametertypen.

Anwendungen für den ersten Fall gibt es viele. Der Name einer Methode soll ihre Aufgabe beschreiben, aber nicht die Typen der Parameter, mit denen sie arbeitet, extra erwähnen. Das ist bei anderen Sprachen üblich, doch nicht in Java. Sehen wir uns als Beispiel die in der Mathe-Klasse `Math` angebotene statische Methode `max()` an. Sie ist mit den Parametertypen `int`, `long`, `float` und `double` deklariert – das ist viel schöner als etwa separate Methoden `maxInt()` und `maxDouble()`.

zB Beispiel

Eine unterschiedliche Anzahl von Parametern ist ebenfalls eine sinnvolle Angelegenheit. Die statische Methode `avg()` könnten wir so für zwei und drei Parameter deklarieren:

```
static double avg( double x, double y ) {
    return (x + y) / 2;
}
static double avg( double x, double y, double z ) {
    return (x + y + z) / 3;
}
```

print() und println() sind überladen *

Das bekannte `print()` und `println()` sind überladene Methoden, die etwa wie folgt deklariert sind:

```
class PrintStream
{
    void print( Object arg ) { ... }
    void print( String arg ) { ... }
    void print( char[] arg ) { ... }
    ...
}
```

Wird nun die Methode `print()` mit irgendeinem Typ aufgerufen, dann wird die am besten passende Methode herausgesucht. Versucht der Programmierer beispielsweise die

Ausgabe eines Objekts `Date`, dann stellt sich die Frage, welche Methode sich darum kümmern kann. Glücklicherweise ist die Antwort nicht schwierig, denn es existiert auf jeden Fall eine `print()`-Methode, die Objekte ausgibt. Und da `Date`, wie auch alle anderen Klassen, eine Unterklasse von `Object` ist, wird `print(Object)` gewählt (natürlich kann nicht erwartet werden, dass das Datum in einem bestimmten Format – etwa nur das Jahr – ausgegeben wird, jedoch wird eine Ausgabe auf dem Schirm sichtbar). Denn jedes Objekt kann sich durch den Namen identifizieren, und dieser würde in dem Fall ausgegeben. Obwohl es sich so anhört, als ob immer die Methode mit dem ParameterTyp `Object` aufgerufen wird, wenn der Datentyp nicht angepasst werden kann, ist dies nicht ganz richtig. Wenn der Compiler keine passende Klasse findet, dann wird die nächste Oberklasse im Ableitungsbaum gesucht, für die in unserem Fall eine Ausgabemethode existiert.

Negative Beispiele und schlaue Leute *

Oft verfolgt auch die Java-Bibliothek die Strategie mit gleichen Namen und unterschiedlichen Typen. Es gibt allerdings einige Ausnahmen. In der Grafik-Bibliothek finden sich die folgenden drei Methoden:

- `drawString(String str, int x, int y)`
- `drawChars(char[] data, int offset, int length, int x, int y)`
- `drawBytes(byte[] data, int offset, int length, int x, int y)`

Das ist äußerst hässlich und schlechter Stil.

Ein anderes Beispiel findet sich in der Klasse `DataOutputStream`. Hier heißen die Methoden etwa `writeInt()`, `writeChar()` und so weiter. Obwohl wir dies auf den ersten Blick verteufeln würden, ist diese Namensgebung sinnvoll. Ein Objekt vom Typ `DataOutputStream` dient zum Schreiben von primitiven Werten in einen Datenstrom. Gäbe es in `DataOutputStream` die überladenen Methoden `write(byte)`, `write(short)`, `write(int)`, `write(long)` und `write(char)` und würden wir sie mit `write(21)` füttern, dann hätten wir das Problem, dass eine Typkonvertierung die Daten automatisch anpassen und der Datenstrom mehr Daten beinhalten würde, als wir wünschen. Denn `write(21)` ruft nicht etwa `write(short)` auf und schreibt zwei Bytes, sondern `write(int)` und schreibt somit vier Bytes. Um also die Übersicht über die geschriebenen Bytes zu behalten, ist eine ausdrückliche Kennzeichnung der Datentypen in manchen Fällen gar nicht so dumm.



Sprachvergleich

Überladene Methoden sind in anderen Programmiersprachen nichts Selbstverständliches. Zum Beispiel erlauben C# und C++ überladene Methoden, JavaScript, PHP und C tun dies jedoch nicht. In Sprachen ohne überladene Methoden wird der Methode/Funktion ein Feld mit Argumenten übergeben. So ist die Typisierung der einzelnen Elemente ein Problem genauso wie die Beschränkung auf eine bestimmte Anzahl von Parametern.

2.7.11 Sichtbarkeit und Gültigkeitsbereich

In jedem Block und auch in jeder Klasse³⁷ können Variablen deklariert werden. Jede Variable hat einen *Geltungsbereich* (engl. *scope*), auch *Gültigkeitsbereich* genannt. Nur in ihrem Gültigkeitsbereich kann der Entwickler auf die Variable zugreifen, außerhalb des Gültigkeitsbereichs nicht. Genauer gesagt: Im Block und in den tiefer geschachtelten Blöcken ist die Variable gültig. Der lesende Zugriff ist nur dann erlaubt, wenn die Variable auch initialisiert wurde.

Der Gültigkeitsbereich bestimmt direkt die *Lebensdauer* der Variable. Eine Variable ist nur in dem Block »lebendig«, in dem sie deklariert wurde. In dem Block ist die Variable lokal.

Dazu ein Beispiel mit zwei statischen Methoden:

Listing 2.31: Scope.java

```
public class Scope
{
    public static void main( String[] args )
    {
        int foo = 0;

        {
            int bar = 0;           // bar gilt nur in diesem Block
            System.out.println( bar );
            System.out.println( foo );
        }
    }
}
```

³⁷ Das sind die sogenannten Objektvariablen oder Klassenvariablen, doch dazu später mehr.

```

        double foo = 0.0;           // ☣ Fehler: Duplicate local variable foo
    }

    System.out.println( foo );
    System.out.println( bar ); // ☣ Fehler: bar cannot be resolved
}

static void qux()
{
    int foo, baz;          // foo hat nichts mit foo aus main() zu tun

    {
        int baz;           // ☣ Fehler: Duplicate local variable baz
    }
}
}

```

Zu jeder Zeit können Blöcke aufgebaut werden. Außerhalb des Blocks sind deklarierte Variablen nicht sichtbar. Nach Abschluss des inneren Blocks, der `bar` deklariert, ist ein Zugriff auf `bar` nicht mehr möglich; auf `foo` ist der Zugriff innerhalb der statischen Methode `main()` weiterhin erlaubt. Dieses `foo` ist aber ein anderes `foo` als in der statischen Methode `qux()`. Eine Variable im Block ist so lange gültig, bis der Block durch eine schließende geschweifte Klammer beendet ist. Innerhalb des Blocks kann die Variable auch nicht umdefiniert werden. Daher schlägt der Versuch fehl, die Variable `foo` ein zweites Mal vom Typ `double` zu deklarieren.

Innerhalb eines Blocks können Variablennamen nicht genauso gewählt werden wie Namen lokaler Variablen eines äußeren Blocks oder wie die Namen für die Parameter einer Methode. Das zeigt die zweite statische Methode am Beispiel der Deklaration `baz`. Obwohl andere Programmiersprachen diese Möglichkeit erlauben – und auch eine Syntax anbieten, um auf eine überschriebene lokale Variable eines höheren Blocks zuzugreifen –, haben sich die Java-Sprachentwickler dagegen entschieden. Gleiche Namen in den inneren und äußeren Blöcken sind nicht erlaubt. Das ist auch gut so, denn es minimiert Fehlerquellen. Die in Methoden deklarierten Parameter sind ebenfalls lokale Variablen und gehören zum Methodenblock.

2.7.12 Vorgegebener Wert für nicht aufgeführte Argumente *

Überladene Methoden lassen sich gut verwenden, wenn vorinitialisierte Werte bei nicht vorhandenen Argumenten genutzt werden sollen. Ist also ein Parameter nicht belegt, soll ein Standardwert eingesetzt werden. Um das zu erreichen, überladen wir einfach die Methode und rufen die andere Methode mit dem Standardwert passend auf (die Sprache C++ definiert in der Sprachgrammatik eine Möglichkeit, die wir in Java nicht haben).

zB Beispiel

Zwei überladene statische Methoden, tax(double cost, double taxRate) und tax(double cost), sollen die Steuer berechnen. Wir möchten, dass der Steuersatz automatisch 19 % ist, wenn die statische Methode tax(double cost) aufgerufen wird und der Steuersatz nicht explizit gegeben ist; im anderen Fall können wir taxRate beliebig wählen.

```
static double tax( double cost, double taxRate )
{
    return cost * taxRate / 100;
}
static double tax( double cost )
{
    return tax( cost, 19.0 );
}
```

2.7.13 Finale lokale Variablen

In einer Methode können Parameter oder lokale Variablen mit dem Modifizierer `final` deklariert werden. Dieses zusätzliche Schlüsselwort verbietet nochmalige Zuweisungen an diese Variable, sodass sie nicht mehr verändert werden kann:

```
static void foo( final int a )
{
    int i = 2;
    final int j = 3;
    i = 3;
    j = 4;      // ☹ Compilerfehler
    a = 2;      // ☹ Compilerfehler
}
```

Aufgeschobene Initialisierung *

Java erlaubt bei finalen Werten eine aufgeschobene Initialisierung. Das heißt, dass nicht zwingend zum Zeitpunkt der Variablen Deklaration ein Wert zugewiesen werden muss. Dies kann auch genau einmal im Programmcode geschehen. Folgendes ist gültig:

```
final int a;
a = 2;
```

Obwohl auch Objektvariablen und Klassenvariablen final sein können, gibt es dort nur beschränkt eine aufgeschobene Initialisierung. Bei der Deklaration müssen wir die Variablen entweder direkt belegen oder im Konstruktor zuweisen. Wir werden uns dies später noch einmal genauer ansehen. Werden finale Variablen vererbt, so können Unterklassen diesen Wert auch nicht mehr überschreiben. (Das wäre ein Problem, aber vielleicht auch ein Vorteil für manche Konstanten.)

2.7.14 Rekursive Methoden *

Wir wollen den Einstieg in die Rekursion mit einem kurzen Beispiel beginnen.

Auf dem Weg durch den Wald begegnet uns eine Fee (engl. *fairy*). Sie sagt zu uns: »Du hast drei Wünsche frei.« Tolle Situation. Um das ganze Unglück aus der Welt zu räumen, entscheiden wir uns nicht für eine egozentrische Wunscherfüllung, sondern für die sozialistische: »Ich möchte Frieden für alle, Gesundheit und Wohlstand für jeden.« Und schwupps, so war es geschehen, und alle lebten glücklich bis ...

Einige Leser werden vielleicht die Hand vor den Kopf schlagen und sagen: »Quatsch! Selbst gießende Blumen, das letzte Ü-Ei in der Sammlung und einen Lebenspartner, der die Trägheit des Morgens duldet.« Glücklicherweise können wir das Dilemma mit der Rekursion lösen. Die Idee ist einfach – und in unseren Träumen schon erprobt –, sie besteht nämlich darin, den letzten Wunsch als »Nochmal drei Wünsche frei« zu formulieren.

Beispiel

zB

Eine kleine Wunsch-Methode:

```
static void fairy()
{
    wish();
    wish();
    fairy();
}
```

Durch den dauernden Aufruf der `fairy()`-Methode haben wir unendlich viele Wünsche frei. *Rekursion* ist also das Aufrufen der eigenen Methode, in der wir uns befinden. Dies kann auch über einen Umweg funktionieren. Das nennt sich dann nicht mehr *direkte Rekursion*, sondern *indirekte Rekursion*. Sie ist ein sehr alltägliches Phänomen, das wir auch von der Rückkopplung Mikrofon/Lautsprecher oder dem Blick mit einem Spiegel in den Spiegel kennen.

Unendliche Rekursionen

Wir müssen nun die Fantasie-Programme (deren Laufzeit und Speicherbedarf auch sehr schwer zu berechnen sind) gegen Java-Methoden austauschen.

zB

Beispiel

Eine Endlos-Rekursion:

Listing 2.32: EndlessRecursion.java, down()

```
static void down( int n )
{
    System.out.print( n + ", " );
    down( n - 1 );
}
```

Rufen wir `down(10)` auf, dann wird die Zahl 10 auf dem Bildschirm ausgegeben und anschließend `down(9)` aufgerufen. Führen wir das Beispiel fort, so ergibt sich eine endlose Ausgabe, die so beginnt und die irgendwann mit einem *StackOverflowError* abbricht:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, ...



```
static void down( int n )
{
    System.out.print( n + ", " );
    down( n - 1 );
}
```

Abbildung 2.12: Steht der Cursor auf einem Methodenaufruf, so markiert Eclipse automatisch alle gleichen anderen Aufrufe oder die Deklaration der Methode.

Abbruch der Rekursion

An dieser Stelle erkennen wir, dass Rekursion prinzipiell etwas Unendliches ist. Für Programme ist dies aber ungünstig. Wir müssen daher ähnlich wie bei Schleifen eine Abbruchbedingung formulieren und dann keinen Rekursionsaufruf mehr starten. Die Abbruchbedingung sieht so aus, dass eine Fallunterscheidung das Argument prüft und mit `return` die Abarbeitung beendet.

Listing 2.33: Recursion.java, down1()

```
static void down1( int n )
{
    if ( n <= 0 )                  // Rekursionsende
        return;

    System.out.print( n + ", " );
    down1( n - 1 );
}
```

Die statische `down1()`-Methode ruft jetzt nur noch so lange `down1(n - 1)` auf, wie das `n` größer null ist. Das ist die *Abbruchbedingung* einer Rekursion.

Unterschiedliche Rekursionsformen

Ein Kennzeichen der bisherigen Programme war, dass nach dem Aufruf der Rekursion keine Anweisung stand, sondern die Methode mit dem Aufruf beendet wurde. Diese Rekursionsform nennt sich Endrekursion. Diese Form ist verhältnismäßig einfach zu verstehen. Schwieriger sind Rekursionen, bei denen hinter dem Methodenaufruf Anweisungen stehen. Betrachten wir folgende Methoden, von denen die erste bekannt und die zweite neu ist:

Listing 2.34: Recursion.java, down1() und down2()

```
static void down1( int n )
{
    if ( n <= 0 )      // Rekursionsende
        return;

    System.out.print( n + ", " );
```

```

    down1( n - 1 );
}

static void down2( int n )
{
    if ( n <= 0 ) // Rekursionsende
        return;

    down2( n - 1 );

    System.out.print( n + ", " );
}

```

Der Unterschied besteht darin, dass `down1()` zuerst die Zahl `n` ausgibt und anschließend rekursiv `down1()` aufruft. Die Methode `down2()` steigt jedoch erst immer tiefer ab, und die Rekursion muss beendet sein, bis es zum ersten `print()` kommt. Daher gibt im Gegensatz zu `down1()` die statische Methode `down2()` die Zahlen in aufsteigender Reihenfolge aus:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Dies ist einleuchtend, wenn wir die Ablaufreihenfolge betrachten. Beim Aufruf `down2(10)` ist der Vergleich von `n` mit null falsch, also wird ohne Ausgabe wieder `down2(9)` aufgerufen. Ohne Ausgabe deshalb, da `print()` ja erst nach dem Methodenaufruf steht. Es geht rekursiv tiefer, bis `n` gleich null ist. Dann endet die letzte Methode mit `return`, und die Ausgabe wird nach dem `down2()`, dem Aufrufer, fortgeführt. Dort ist `print()` die nächste Anweisung. Da wir nun noch tief verschachtelt stecken, gibt `print(n)` die Zahl 1 aus. Dann ist die Methode `down2()` wieder beendet (ein unsichtbares, nicht direkt geschriebenes `return`), und sie springt zum Aufrufer zurück. Das war wieder die Methode `down2()`, aber mit der Belegung `n = 2`. Das geht so weiter, bis es zurück zum Aufrufer kommt, der `down(10)` aufgerufen hat, zum Beispiel der statischen `main()`-Methode. Der Trick bei der Sache besteht nun darin, dass jede Methode ihre eigene lokale Variable besitzt.



Die Tastenkombination **[Strg]** + **[Alt]** + **[H]** zeigt die Aufrufhierarchie an. So ist zu sehen, wer eine Methode aufruft. In den Aufrufen von `down2()` tauchen also wiederum wegen des rekursiven Aufrufs `down2()` sowie `main()` auf.

Rekursion und der Stack sowie die Gefahr eines StackOverflowError *

Am Beispiel haben wir gesehen, dass der Aufruf von `down2(10)` zum Aufruf von `down2(9)` führt. Und `down2(10)` kann erst dann beendet werden, wenn `down2(9)` komplett abgearbeitet wurde. `down2(10)` ist sozusagen so lange »offen«, bis der Schwanz von untergeordneten Aufrufen beendet ist. Nun muss sich die Laufzeitumgebung natürlich bei einem Methodenaufruf merken, wo es nach dem Methodenaufruf weitergeht. Dazu nutzt sie den Stack. Beim Aufruf von `down2(9)` etwa wird der Stack mit der Rücksprungadresse gefüllt, der zum Kontext von `down2(10)` zurückführt. In jedem Kontext gibt es auch wieder die alten lokalen Variablen.

Gibt es bei Rekursionen keine Abbruchbedingung, so kommen immer mehr Rücksprungadressen auf den Stapel, bis der Stapel keinen Platz mehr hat. Dann folgt ein `java.lang.StackOverflowError`, und das Programm (der Thread) bricht ab. In der Regel deutet der `StackOverflowError` auf einen Programmierfehler hin, es gibt aber Programme, die einen wirklich großen Stack benötigen und für die die Stack-Größe einfach zu klein ist.

Standardmäßig ist die Stack-Größe 512 KiB. Sie lässt sich über einen JVM-Schalter³⁸ vergrößern, der `-Xss:n` heißt (oder etwas länger in der Schreibweise `-XX:ThreadStackSize=n`). Um ihn auf 2 MiB (2048 KiB) zu setzen, schreiben wir:

```
$ java -XXs:2048 MyApplication
```

Die Stack-Größe gilt damit für alle Threads in der JVM, was natürlich bei großen Stacks und vielen Threads zu einem Speicherproblem führen kann. Umgekehrt lässt sich auch Speicher einsparen, wenn das System sehr viele Threads nutzt und die Stack-Größe verringert wird.

2.7.15 Die Türme von Hanoi *

Die Legende der Türme von Hanoi soll erstmalig von Ed Lucas in einem Artikel in der französischen Zeitschrift »Cosmo« im Jahre 1890 veröffentlicht worden sein.³⁹ Der Legende nach standen vor langer Zeit im Tempel von Hanoi drei Säulen. Die erste war aus Kupfer, die zweite aus Silber und die dritte aus Gold. Auf der Kupfersäule waren einhundert Scheiben aufgestapelt. Die Scheiben hatten in der Mitte ein Loch und waren aus

³⁸ <http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp>

³⁹ Wir halten uns hier an eine Überlieferung von C. H. A. Koster aus dem Buch »Top-down Programming with Elan« von Ellis Horwood (Verlag Ellis Horwood Ltd , ISBN 0139249370, 1987).

Porphy⁴⁰. Die Scheibe mit dem größten Umfang lag unten, und alle kleiner werdenden Scheiben lagen obenauf. Ein alter Mönch stellte sich die Aufgabe, den Turm der Scheiben von der Kupfersäule zur Goldsäule zu bewegen. In einem Schritt sollte aber nur eine Scheibe bewegt werden, und zudem war die Bedingung, dass eine größere Scheibe niemals auf eine kleinere bewegt werden durfte. Der Mönch erkannte schnell, dass er die Silbersäule nutzen musste; er setzte sich an einen Tisch, machte einen Plan, überlegte und kam zu einer Entscheidung. Er konnte sein Problem in drei Schritten lösen. Am nächsten Tag schlug der Mönch die Lösung an die Tempeltür:

- Falls der Turm aus mehr als einer Scheibe besteht, bitte deinen ältesten Schüler, einen Turm von $(n - 1)$ Scheiben von der ersten zur dritten Säule unter Verwendung der zweiten Säule umzusetzen.
- Trage selbst die erste Scheibe von einer zur anderen Säule.
- Falls der Turm aus mehr als einer Scheibe besteht, bitte deinen ältesten Schüler, einen Turm aus $(n - 1)$ Scheiben von der dritten zu der anderen Säule unter Verwendung der ersten Säule zu transportieren.

Und so rief der alte Mönch seinen ältesten Schüler zu sich und trug ihm auf, den Turm aus 99 Scheiben von der Kupfersäule zur Goldsäule unter Verwendung der Silbersäule umzuschichten und ihm den Vollzug zu melden. Nach der Legende würde das Ende der Welt nahe sein, bis der Mönch seine Arbeit beendet hätte. Nun, so weit die Geschichte. Wollen wir den Algorithmus zur Umschichtung der Porphyrscheiben in Java programmieren, so ist eine rekursive Lösung recht einfach. Werfen wir einen Blick auf das folgende Programm, das die Umschichtungen über die drei Pflöcke (engl. *pegs*) vornimmt.

Listing 2.35: TowerOfHanoi.java

```
class TowerOfHanoi
{
    static void move( int n, String fromPeg, String toPeg, String usingPeg )
    {
        if ( n > 1 )
        {
            move( n - 1, fromPeg, usingPeg, toPeg );
            System.out.printf( "Bewege Scheibe %d von der %s zur %s.%n", n, fromPeg, toPeg );
            move( n - 1, usingPeg, toPeg, fromPeg );
        }
    }
}
```

⁴⁰ Gestein vulkanischen Ursprungs. Besondere Eigenschaften von Porphyrr sind: hohe Bruchfestigkeit, hohe Beständigkeit gegen physikalisch-chemische Wirkstoffe und hohe Wälz- und Gleitreibung.

```
    }
else
    System.out.printf( "Bewege Scheibe %d von der %s zur %s.%n", n, fromPeg, toPeg );
}

public static void main( String[] args )
{
    move( 4, "Kupfersäule", "Silbersäule", "Goldsäule" );
}
}
```

Starten wir das Programm mit vier Scheiben, so bekommen wir folgende Ausgabe:

```
Bewege Scheibe 1 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 2 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 1 von der Goldsäule zur Silbersäule.
Bewege Scheibe 3 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 1 von der Silbersäule zur Kupfersäule.
Bewege Scheibe 2 von der Silbersäule zur Goldsäule.
Bewege Scheibe 1 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 4 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 1 von der Goldsäule zur Silbersäule.
Bewege Scheibe 2 von der Goldsäule zur Kupfersäule.
Bewege Scheibe 1 von der Silbersäule zur Kupfersäule.
Bewege Scheibe 3 von der Goldsäule zur Silbersäule.
Bewege Scheibe 1 von der Kupfersäule zur Goldsäule.
Bewege Scheibe 2 von der Kupfersäule zur Silbersäule.
Bewege Scheibe 1 von der Goldsäule zur Silbersäule.
```

Schon bei vier Scheiben haben wir 15 Bewegungen. Selbst wenn unser Prozessor mit vielen Millionen Operationen pro Sekunde arbeitet, benötigt ein Computer für die Abarbeitung Tausende geologischer Erdzeitalter. An diesem Beispiel wird eines deutlich: Viele Dinge sind im Prinzip berechenbar, nur praktisch ist so ein Algorithmus nicht.

2.8 Zum Weiterlesen

Die allumfassende Super-Quelle ist die *Java Language Specification*, die online unter <http://java.sun.com/docs/books/jls/> zu finden ist. Zweifelsfälle löst die Spezifikation auf, obwohl die Informationen zum Teil etwas verstreut sind.

Der niederländische Maler Maurits Cornelis Escher (1898–1972) machte die Rekursion auch in Bildern berühmt. Seiten mit Bildern und seine Vita finden sich zum Beispiel unter http://de.wikipedia.org/wiki/M._C._Escher.

Zu Beginn eines Projekts sollten Entwickler Kodierungsstandards (engl. *code conventions*) festlegen. Eine erste Informationsquelle ist <http://tutego.de/go/codeconv>. Amüsanter ist dazu auch <http://tutego.de/go/unmain> zu lesen.



Kapitel 3

Klassen und Objekte

3

»Nichts auf der Welt ist so gerecht verteilt wie der Verstand. Denn jedermann ist davon überzeugt, dass er genug davon habe.«
– René Descartes (1596–1650)

3.1 Objektorientierte Programmierung (OOP)

In einem Buch über Java-Programmierung müssen mehrere Teile vereinigt werden: die grundsätzliche Programmierung nach dem imperativen Prinzip für einfache statische Methoden und eine neue Grammatik für Java, dann die Objektorientierung und die Bibliotheken. Dieses Kapitel stellt das Paradigma der Objektorientierung in den Mittelpunkt und zeigt die Syntax, wie etwa in Java Vererbung realisiert wird.

Hinweis

Java ist natürlich nicht die erste objektorientierte Sprache (OO-Sprache), auch C++ war nicht die erste. Klassischerweise gelten Smalltalk und insbesondere Simula-67 als Stammväter aller OO-Sprachen. Die eingeführten Konzepte sind bis heute aktuell, darunter die vier allgemein anerkannten Prinzipien der OOP: *Abstraktion, Kapselung, Vererbung und Polymorphie*.¹



3.1.1 Warum überhaupt OOP?

Da Menschen die Welt in Objekten wahrnehmen, wird auch die Analyse von Systemen häufig schon objektorientiert modelliert. Doch mit prozeduralen Systemen, die lediglich Unterprogramme als Ausdrucksmittel haben, wird die Abbildung des objektorientierten Designs in eine Programmiersprache schwer, und es entsteht ein Bruch. Im

¹ Keine Sorge, alle vier Grundsäulen werden in den nächsten Kapiteln ausführlich beschrieben!

Laufe der Zeit entwickeln sich Dokumentation und Implementierung auseinander; die Software ist dann schwer zu warten und zu erweitern.

Die in der Software abgebildeten Objekte haben drei wichtige Eigenschaften:

- Jedes Objekt hat eine Identität.
- Jedes Objekt hat einen Zustand.
- Jedes Objekt zeigt ein Verhalten.

Diese drei Eigenschaften haben wichtige Konsequenzen: zum einen, dass die Identität des Objekts während seines Lebens bis zu seinem Tod die gleiche bleibt und sich nicht ändern kann. Zum anderen werden die Daten und der Programmcode zur Manipulation dieser Daten als zusammengehörig behandelt. In prozeduralen Systemen finden sich oft Szenarien wie das folgende: Es gibt einen großen Speicherbereich, auf den alle Unterprogramme irgendwie zugreifen können. Bei den Objekten ist das anders, da sie logisch ihre eigenen Daten verwalten und die Manipulation überwachen.

In der objektorientierten Softwareentwicklung geht es also darum, in Objekten zu modellieren und dann zu programmieren. Das Design nimmt dabei eine zentrale Stellung ein; große Systeme werden zerlegt und immer feiner beschrieben. Hier passt sehr gut die Aussage des französischen Schriftstellers François Duc de La Rochefoucauld (1613–1680):

»Wer sich zu viel mit dem Kleinen abgibt, wird unfähig für Großes.«

3.1.2 Denk ich an Java, denk ich an Wiederverwendbarkeit

Bei jedem neuen Projekt fällt auf, dass in früheren Projekten schon ähnliche Probleme gelöst werden mussten. Natürlich sollen bereits gelöste Probleme nicht neu implementiert, sondern sich wiederholende Teile bestmöglich in unterschiedlichen Kontexten wiederverwendet werden; das Ziel ist die bestmögliche Wiederverwendung von Komponenten.

Wiederverwendbarkeit von Programmteilen gibt es nicht erst seit den objektorientierten Programmiersprachen, objektorientierte Programmiersprachen erleichtern aber die Programmierung wiederverwendbarer Softwarekomponenten. So sind auch die vielen tausend Klassen der Bibliothek ein Beispiel dafür, dass sich Entwickler nicht ständig um die Umsetzung etwa von Datenstrukturen oder um die Pufferung von Datenströmen kümmern müssen.

Auch wenn Java eine objektorientierte Programmiersprache ist, muss das kein Garant für tolles Design und optimale Wiederverwendbarkeit sein. Eine objektorientierte Programmiersprache erleichtert objektorientiertes Programmieren, aber auch in einer einfachen Programmiersprache wie C lässt sich objektorientiert programmieren. In Java sind auch Programme möglich, die aus nur einer Klasse bestehen und dort 5.000 Zeilen Programmcode mit statischen Methoden unterbringen. Bjarne Stroustrup (der Schöpfer von C++, von seinen Freunden auch Stumpy genannt) sagte treffend über den Vergleich von C und C++:

»C makes it easy to shoot yourself in the foot, C++ makes it harder, but when you do, it blows away your whole leg.«²

Im Sinne unserer didaktischen Vorgehensweise wird dieses Kapitel zunächst einige Klassen der Standardbibliothek verwenden. Wir beginnen mit der Klasse `Point`, die zweidimensionale Punkte repräsentiert. In einem zweiten Schritt werden wir eigene Klassen programmieren. Anschließend kümmern wir uns um das Konzept der Modularität in Java, nämlich darum, wie Gruppen zusammenhängender Klassen gestaltet werden.

3.2 Eigenschaften einer Klasse

Klassen sind das wichtigste Merkmal objektorientierter Programmiersprachen. Eine Klasse definiert einen neuen Typ, beschreibt die Eigenschaften der Objekte und gibt somit den Bauplan an. Jedes Objekt ist ein *Exemplar* (auch *Instanz*³ oder *Ausprägung* genannt) einer Klasse.

Eine Klasse deklariert im Wesentlichen zwei Dinge:

- Attribute (was das Objekt hat)
- Operationen (was das Objekt kann)

Attribute und Operationen heißen auch *Eigenschaften* eines Objekts; einige Autoren nennen allerdings nur Attribute Eigenschaften. Welche Eigenschaften eine Klasse tatsächlich besitzen soll, wird in der Analyse- und Designphase festgesetzt. Diese wird in diesem Buch kein Thema sein; für uns liegen die Klassenbeschreibungen schon vor.

² ... oder wie es Bertrand Meyer sagt: »Do not replace legacy software by legacy-c++ software.«

³ Ich vermeide das Wort *Instanz* und verwende dafür durchgängig das Wort *Exemplar*. An die Stelle von *instanziieren* tritt das einfache Wort *erzeugen*. Instanz ist eine irreführende Übersetzung des englischen Ausdrucks »instance«.

Die Operationen einer Klasse setzt die Programmiersprache Java durch Methoden um. Die Attribute eines Objekts definieren die Zustände, und sie werden durch Variablen implementiert (die auch *Felder*⁴ genannt werden).

Um sich einer Klasse zu nähern, können wir einen lustigen *Ich-Ansatz (Objektansatz)* verwenden, der auch in der Analyse- und Designphase eingesetzt wird. Bei diesem Ich-Ansatz versetzen wir uns in das Objekt und sagen »Ich bin ...« für die Klasse, »Ich habe ...« für die Attribute und »Ich kann ...« für die Operationen. Meine Leser sollten dies bitte an den Klassen Mensch, Auto, Wurm und Kuchen testen.

3.2.1 Die Klasse Point

Bevor wir uns mit eigenen Klassen beschäftigen, wollen wir zunächst einige Klassen aus der Standardbibliothek kennenlernen. Eine einfache Klasse ist Point. Sie beschreibt durch die Koordinaten x und y einen Punkt in einer zweidimensionalen Ebene und bietet einige Operationen an, mit denen sich Punkt-Objekte verändern lassen. Testen wir einen Punkt wieder mit dem Objektansatz:

Klassenname	Ich bin ein Punkt.
Attribute	Ich habe eine x- und y-Koordinate.
Operationen	Ich kann mich verschieben und meine Position festlegen.

Tabelle 3.1: OOP-Begriffe und was sie bedeuten

Zu unserem Punkt können wir in der API-Dokumentation (<http://download.oracle.com/javase/7/docs/api/java.awt/Point.html>) von Oracle nachlesen, dass dieser die Variablen x und y definiert, unter anderem eine Methode setLocation() besitzt und einen Konstruktor anbietet, der zwei Ganzzahlen annimmt.

3.3 Die UML (Unified Modeling Language) *

Für die Darstellung einer Klasse lässt sich Programmcode verwenden, also eine Textform, oder aber eine grafische Notation. Eine dieser grafischen Beschreibungsformen ist die UML. Grafische Abbildungen sind für Menschen deutlich besser zu verstehen und erhöhen die Übersicht.

⁴ Den Begriff *Feld* benutze ich im Folgenden nicht. Er bleibt für Arrays reserviert.

Im ersten Abschnitt eines UML-Diagramms lassen sich die Attribute ablesen, im zweiten die Operationen. Das + vor den Eigenschaften zeigt an, dass sie öffentlich sind und jeder sie nutzen kann. Die Typenangabe ist gegenüber Java umgekehrt: Zuerst kommt der Name der Variable, dann der Typ beziehungsweise bei Methoden der Typ des Rückgabewerts.

java.awt.Point
+ x: int
+ y: int
+ Point()
+ Point(p: Point)
+ Point(x: int, y: int)
+ getX(): double
+ getY(): double
+ getLocation(): Point
+ setLocation(p: Point)
+ setLocation(x: int, y: int)
+ setLocation(x: double, y: double)
+ move(x: int, y: int)
+ translate(dx: int, dy: int)
+ equals(obj: Object): boolean
+ toString(): String

Abbildung 3.1: Die Klasse java.awt.Point in der UML-Darstellung

3.3.1 Hintergrund und Geschichte der UML

Die UML ist mehr als eine Notation zur Darstellung von Klassen. Mit ihrer Hilfe lassen sich Analyse und Design im Softwareentwicklungsprozess beschreiben. Mittlerweile hat sich die UML jedoch zu einer allgemeinen Notation für andere Beschreibungen entwickelt, zum Beispiel für Datenbanken oder Workflow-Anwendungen.

Vor der UML waren andere Darstellungsvarianten wie OMT oder Booch verbreitet. Diese waren eng mit einer Methode verbunden, die einen Entwicklungsprozess und ein Vorgehensmodell beschrieb. Methoden versuchten, eine Vorgehensweise beim Entwurf von Systemen zu beschreiben, etwa »erst Vererbung einsetzen und dann die Attribute finden« oder »erst die Attribute finden und dann mit Vererbung verfeinern«. Bekannte OO-Methoden sind etwa Shlaer/Mellor, Coad/Yourdon, Booch, OMT und OOSE/Objectory. Aus dem Wunsch heraus, OO-Methoden zusammenzufassen, entstand die UML – anfangs stand die Abkürzung noch für *Unified Method*. Die Urversion 0.8 wurde im Jahre 1995 veröffentlicht. Die Initiatoren waren Jim Rumbaugh und Grady Booch. Später kam Ivar Jacobson dazu, und die drei »Amigos« erweiterten die UML, die in der Version

1.0 bei der *Object Management Group* (OMG) als Standardisierungsvorschlag eingereicht wurde. Die Amigos nannten die UML nun *Unified Modeling Language*, was deutlich macht, dass die UML keine Methode ist, sondern lediglich eine Modellierungssprache. Die Spezifikation erweitert sich ständig mit dem Aufkommen neuer Software-Techniken, und so bildet die UML 2.0 Konzepte wie *Model-Driven Architecture* (MDA) und *Geschäftsprozessmodellierung* (BPM) ab und unterstützt *Echtzeitmodellierung* (RT) durch spezielle Diagrammtypen.

Eine aktuelle Version des Standards lässt sich unter <http://tutego.de/go/uml> einsehen.

3.3.2 Wichtige Diagrammtypen der UML

UML definiert diverse Diagrammtypen, die unterschiedliche Sichten auf die Software beschreiben können. Für die einzelnen Phasen im Softwareentwurf sind jeweils andere Diagramme wichtig. Wir wollen kurz vier Diagramme und ihr Einsatzgebiet besprechen:

Anwendungsfalldiagramm

Ein *Anwendungsfalldiagramm* (Use-Cases-Diagramm) entsteht meist während der Anforderungsphase und beschreibt die Geschäftsprozesse, indem es die Interaktion von Personen – oder von bereits existierenden Programmen – mit dem System darstellt. Die handelnden Personen oder aktiven Systeme werden *Aktoren* genannt und sind im Diagramm als kleine (geschlechtslose) Männchen angedeutet. Anwendungsfälle (Use Cases) beschreiben dann eine Interaktion mit dem System.

Klassendiagramm

Für die statische Ansicht eines Programmerturfs ist das *Klassendiagramm* einer der wichtigsten Diagrammtypen. Ein Klassendiagramm stellt zum einen die Elemente der Klasse dar, also die Attribute und Operationen, und zum anderen die Beziehungen der Klassen untereinander. Klassendiagramme werden in diesem Buch häufiger eingesetzt, um insbesondere die Assoziation und Vererbung zu anderen Klassen zu zeigen. Klassen werden in einem solchen Diagramm als Rechteck dargestellt, und die Beziehungen zwischen den Klassen werden durch Linien angedeutet.

Objektdiagramm

Ein Klassendiagramm und ein Objektdiagramm sind sich auf den ersten Blick sehr ähnlich. Der wesentliche Unterschied besteht aber darin, dass ein *Objektdiagramm* die

Belegung der Attribute, also den Objektzustand, visualisiert. Dazu werden sogenannte *Ausprägungsspezifikationen* verwendet. Mit eingeschlossen sind die Beziehungen, die das Objekt zur Laufzeit mit anderen Objekten hält. Beschreibt zum Beispiel ein Klassendiagramm eine Person, so ist es nur ein Rechteck im Diagramm. Hat diese Person zur Laufzeit Freunde (es gibt also Assoziationen zu anderen Personen-Objekten), so können sehr viele Personen in einem Objektdiagramm verbunden sein, während ein Klassendiagramm diese Ausprägung nicht darstellen kann.

Sequenzdiagramm

Das *Sequenzdiagramm* stellt das dynamische Verhalten von Objekten dar. So zeigt es an, in welcher Reihenfolge Operationen aufgerufen und wann neue Objekte erzeugt werden. Die einzelnen Objekte bekommen eine vertikale Lebenslinie, und horizontale Linien zwischen den Lebenslinien der Objekte beschreiben die Operationen oder Objekterzeugungen. Das Diagramm liest sich somit von oben nach unten.

Da das Klassendiagramm und das Objektdiagramm eher die Struktur einer Software beschreiben, heißen die Modelle auch *Strukturdiagramme* (neben Paketdiagramm, Komponentendiagramm, Kompositionssstrukturdiagramm und Verteilungsdiagramm). Ein Anwendungsfalldiagramm und ein Sequenzdiagramm zeigen daher eher das dynamische Verhalten und werden *Verhaltensdiagramme* genannt. Weitere Verhaltensdiagramme sind das Zustandsdiagramm, das Aktivitätsdiagramm, das Interaktionsübersichtsdiagramm, das Kommunikationsdiagramm und das Zeitverlaufsdigramm. In der UML ist es aber wichtig, die zentralen Aussagen des Systems in einem Diagramm festzuhalten, sodass sich problemlos Diagrammtypen mischen lassen.

3.3.3 UML-Werkzeuge

In der Softwareentwicklung gibt es nicht nur den Java-Compiler und die Laufzeitumgebung, sondern viele weitere Tools. Eine Kategorie von Produkten bilden Modellierungswerkzeuge, die bei der Abbildung einer Realwelt auf die Softwarewelt helfen. Insbesondere geht es um Software, die alle Phasen im Entwicklungsprozess abbildet.

UML-Werkzeuge formen eine wichtige Gruppe, und ihr zentrales Element ist ein grafisches Werkzeug. Mit ihm lassen sich die UML-Diagramme zeichnen und verändern. Im nächsten Schritt kann ein gutes UML-Tool aus diesen Zeichnungen Java-Code erzeugen. Noch weiter als eine einfache Codeerzeugung gehen Werkzeuge, die aus Java-Code umgekehrt UML-Diagramme generieren. Diese *Reverse-Engineering-Tools* haben jedoch eine schwere Aufgabe, da Java-Quellcode semantisch so reichhaltig ist, dass entweder

das UML-Diagramm »zu voll« ist, völlig unzureichend formatiert ist oder Dinge nicht kompakt abgebildet werden. Die Königsdisziplin der UML-Tools bildet das *Roundtrip-Engineering*. Im Optimalfall sind dann das UML-Diagramm und der Quellcode synchron, und jede Änderung der einen Seite spiegelt sich sofort in einer Änderung auf der anderen Seite wider.

UML-Produkte

Global gesehen, ist die Anzahl der UML-Tools groß, doch schmilzt die Zahl der Werkzeuge, die in Eclipse eingebunden werden können, rasch zusammen. Noch kleiner ist die Zahl freier UML-Tools. Hier sind einige Empfehlungen:

- *eUML2* (<http://www.soyatec.com/>) und *OMONDO* (<http://www.omondo.de/>): Dies sind Eclipse-basierte UML-Tools. Es gibt freie, eingeschränkte Varianten. Einige Entwickler von OMONDO haben sich abgespalten, und daraus ist eUML2 entstanden.
- *ArgoUML* (<http://argouml.tigris.org/>) ist ein freies UML-Werkzeug mit UML 1.4-Notation.
- *Apollo for Eclipse* und *Poseidon for UML* (basiert auf NetBeans) stammen von der deutschen Gentleware AG (<http://www.gentleware.com/>) in Hamburg. Die Firma wirbt mit folgender Aussage: »*Poseidon for UML is the world's most downloaded commercial UML tool, with over 1,200,000 copies distributed to over 100 countries.*«⁵
- *Together* (<http://www.borland.com/de/products/together/>) ist ein alter Hase unter den UML-Tools. Ursprünglich von Togethersoft als eigenständige UML-Software entwickelt, ist es dann in die Hände Borlands gekommen und in die JBuilder-IDE gewandert. Die aktuelle Version *Borland Together* basiert auf Eclipse.
- *Rational Rose* (<http://www-01.ibm.com/software/de/rational/design.html>): Das professionelle UML-Werkzeug von IBM zeichnet sich durch seinen Preis aus, aber auch durch die Integration einer ganzen Reihe weiterer Werkzeuge, etwa für Anforderungsdokumente, Tests usw.
- *NetBeans 6* (<http://www.netbeans.org/features/uml/>). NetBeans brachte viele Jahre ein schönes UML-Tool mit. Wie die NetBeans-IDE auch, war es frei. Leider wurde die Weiterentwicklung auf Eis gelegt, und es ist nur für ältere NetBeans-Versionen lauffähig. Die Entwickler sagen, dass die Code-Qualität so schlecht war, dass eine Weiterentwicklung für NetBeans 7 nicht mehr sinnvoll war; vielmehr wollte man eine Neuimplementierung beginnen.

⁵ Der Spruch steht schon ein bisschen länger auf der Webseite.

3.4 Neue Objekte erzeugen

Eine Klasse beschreibt also, wie ein Objekt aussehen soll. In einer Mengen- beziehungsweise Element-Beziehung ausgedrückt, entsprechen Objekte den Elementen und Klassen den Mengen, in denen die Objekte als Elemente enthalten sind. Diese Objekte haben Eigenschaften, die sich nutzen lassen. Wenn ein Punkt Koordinaten repräsentiert, wird es Möglichkeiten geben, diese Zustände zu erfragen und zu ändern.

Im Folgenden wollen wir untersuchen, wie sich von der Klasse Point zur Laufzeit Exemplare erzeugen lassen und wie der Zugriff auf die Eigenschaften der Point-Objekte aussieht.

3.4.1 Ein Exemplar einer Klasse mit dem new-Operator anlegen

Objekte müssen in Java immer ausdrücklich erzeugt werden. Dazu definiert die Sprache den new-Operator.

Beispiel

zB

Anlegen eines Punkt-Objekts:

```
new java.awt.Point();
```

Hinter dem new-Operator folgt der Name der Klasse, von der ein Exemplar erzeugt werden soll. Der Klassename ist hier voll qualifiziert angegeben, da sich Point in einem Paket java.awt befindet. (Ein Paket ist eine Gruppe zusammengehöriger Klassen. Wir werden später bei den import-Deklarationen sehen, dass Entwickler diese Schreibweise auch abkürzen können.) Hinter dem Klassennamen folgt ein Paar runder Klammern für den Konstruktoraufzug. Dieser ist eine Art Methodenaufruf, über den sich Werte für die Initialisierung des frischen Objekts übergeben lassen.

Konnte die Speicherverwaltung von Java für das anzulegende Objekt freien Speicher reservieren und konnte der Konstruktor gültig durchlaufen werden, gibt der new-Ausdruck anschließend eine Referenz auf das frische Objekt an das Programm zurück.

Der Zusammenhang von new, Heap und Garbage-Collector

Bekommt das Laufzeitsystem die Anfrage, ein Objekt mit new zu erzeugen, so reserviert es so viel Speicher, dass alle Objekteigenschaften und Verwaltungsinformationen dort Platz finden. Ein Point-Objekt speichert die Koordinaten in zwei int-Werte, also sind

mindestens 2 mal 4 Byte nötig. Den Speicherplatz nimmt die Laufzeitumgebung vom *Heap*. Der hat eine vordefinierte Maximalgröße, damit ein Java-Programm nicht beliebig viel Speicher vom Betriebssystem abgreifen kann, was die Maschine möglicherweise in den Ruin treibt. In den frühen JVM-Versionen betrug standardmäßig die maximale Heap-Größe fest 64 MiB; ab Java 5 beträgt sie $\frac{1}{4}$ des Hauptspeichers, aber maximal 1 GiB⁶.



Hinweis

Es gibt in Java nur wenige Sonderfälle, wann neue Objekte nicht über den new-Operator angelegt werden. So erzeugt die auf nativem Code basierende Methode `newInstance()` vom Class- oder Constructor-Objekt ein neues Objekt. Auch `clone()` kann ein neues Objekt als Kopie eines anderen Objekts erzeugen. Bei der String-Konkatenation mit `+` ist für uns zwar kein new-Operator zu sehen, doch der Compiler wird ein `new` einsetzen, um das neue String-Objekt anzulegen.⁷

Ist das System nicht in der Lage, genügend Speicher für ein neues Objekt bereitzustellen, versucht der Garbage-Collector in einer letzten Rettungsaktion, alles wegzuräumen. Ist dann immer noch nicht ausreichend Speicher frei, generiert die Laufzeitumgebung einen `OutOfMemoryError` und bricht die Abarbeitung ab.

Heap und Stack

Die JVM-Spezifikation sieht für Daten fünf verschiedene Speicherbereiche (engl. *runtime data area*) vor.⁸ Neben dem *Heap-Speicher* wollen wir uns den *Stack-Speicher* (Stapelspeicher) kurz anschauen. Den nutzt die Java-Laufzeitumgebung zum Beispiel für lokale Variablen. Auch verwendet Java den Stack beim Methodenaufruf mit Parametern. Die Argumente kommen vor dem Methodenaufruf auf den Stapel, und die aufgerufene Methode kann über den Stack auf die Werte lesend oder schreibend zugreifen. Bei endlosen rekursiven Methodenaufrufen ist irgendwann die maximale Stack-Größe erreicht, und es kommt zu einer Exception vom Typ `java.lang.StackOverflowError`. Da mit jedem Thread ein JVM-Stack assoziiert ist, bedeutet das das Ende des Threads.

⁶ <http://download.oracle.com/javase/1.5.0/docs/guide/vm/gc-ergonomics.html>

⁷ Der Compiler generiert selbstständig zum Beispiel beim Ausdruck `s + t` einen Ausdruck wie `new StringBuilder().append(s).append(t).toString()`.

⁸ § 3.5 der JVM-Spezifikation, http://java.sun.com/docs/books/jvms/second_edition/html/Overview.doc.html#1732.

3.4.2 Garbage-Collector (GC) – Es ist dann mal weg

Wird das Objekt nicht mehr vom Programm referenziert, so bemerkt dies der Garbage-Collector (GC) und gibt den reservierten Speicher wieder frei.⁹ Der GC testet dazu regelmäßig, ob die Objekte auf dem Heap noch benötigt werden. Werden sie nicht benötigt, werden sie gelöscht. Es weht also immer ein Hauch von Friedhof über dem Heap, und nachdem die letzte Referenz vom Objekt genommen wird, ist es auch schon tot.

3.4.3 Deklarieren von Referenzvariablen

Das Ergebnis des `new`-Operators ist eine Referenz auf das neue Objekt. Die Referenz wird in der Regel in einer *Referenzvariablen* zwischengespeichert, um fortlaufende Eigenschaften vom Objekt nutzen zu können.

Beispiel

zB

Deklariere die Variable `p` vom Typ `java.awt.Point`. Die Variable `p` nimmt anschließend die Referenz von dem neuen Objekt auf, das mit `new` angelegt wurde.

```
java.awt.Point p;
p = new java.awt.Point();
```

Die Deklaration und die Initialisierung einer Referenzvariablen lassen sich kombinieren (auch eine lokale Referenzvariable ist zu Beginn uninitialisiert):

```
java.awt.Point p = new java.awt.Point();
```

Die Typen müssen natürlich kompatibel sein, und ein Punkt-Objekt geht nicht als Typ einer Socke durch. Der Versuch, ein Punktobjekt einer `int`- oder `String`-Variablen zuzuweisen, ergibt somit einen Compilerfehler.

```
int    p = new java.awt.Point(); // ☹ Type mismatch: cannot convert from
                                // Point to int
String s = new java.awt.Point(); // ☹ Type mismatch: cannot convert from
                                // Point to String
```

⁹ Mit dem gesetzten `java`-Schalter `-verbose:gc` gibt es immer Konsolenausgaben, wenn der GC nicht mehr referenzierte Objekte erkennt und wegräumt.

Damit speichert eine Variable entweder einen einfachen Wert (Variable vom Typ `int`, `boolean`, `double` ...) oder einen Verweis auf ein Objekt. Referenztypen gibt es in drei Ausführungen: *Klassentypen*, *Schnittstellentypen* (auch *Interface-Typen* genannt) und *Feldtypen* (auch *Array-Typen* genannt). In unserem Fall haben wir ein Beispiel für einen Klassentyp.

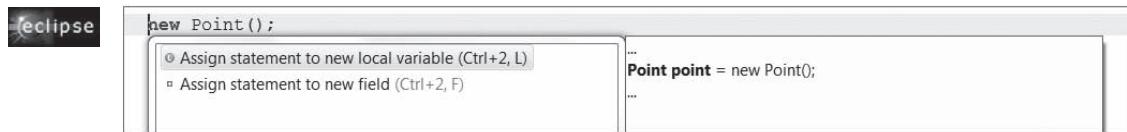


Abbildung 3.2: [Strg] + [1] ermöglicht es, entweder eine neue lokale Variable oder eine Objektvariable für den Ausdruck anzulegen.

3.4.4 Zugriff auf Objektattribute und -methoden mit dem ».«

Die in einer Klasse deklarierten Variablen heißen *Objektvariablen* beziehungsweise *Exemplar-, Instanz- oder Ausprägungsvariablen*. Jedes erzeugte Objekt hat seinen eigenen Satz von Objektvariablen¹⁰: Sie bilden den *Zustand des Objekts*.

Der Punkt-Operator ».« erlaubt auf Objekten den Zugriff auf die Methoden oder Zustände. Er steht zwischen einem Ausdruck, der eine Referenz liefert, und der Objekt-eigenschaft. Welche Möglichkeiten eine Klasse genau bietet, zeigt die API-Dokumentation.

zB Beispiel

Die Variable `p` referenziert ein `java.awt.Point`-Objekt. Die Objektvariablen `x` und `y` sollen initialisiert werden:

```
java.awt.Point p = new java.awt.Point();
p.x = 1;
p.y = 2 + p.x;
```

¹⁰ Es gibt auch den Fall, dass sich mehrere Objekte eine Variable teilen, sogenannte statische Variablen. Diesen Fall werden wir später betrachten.

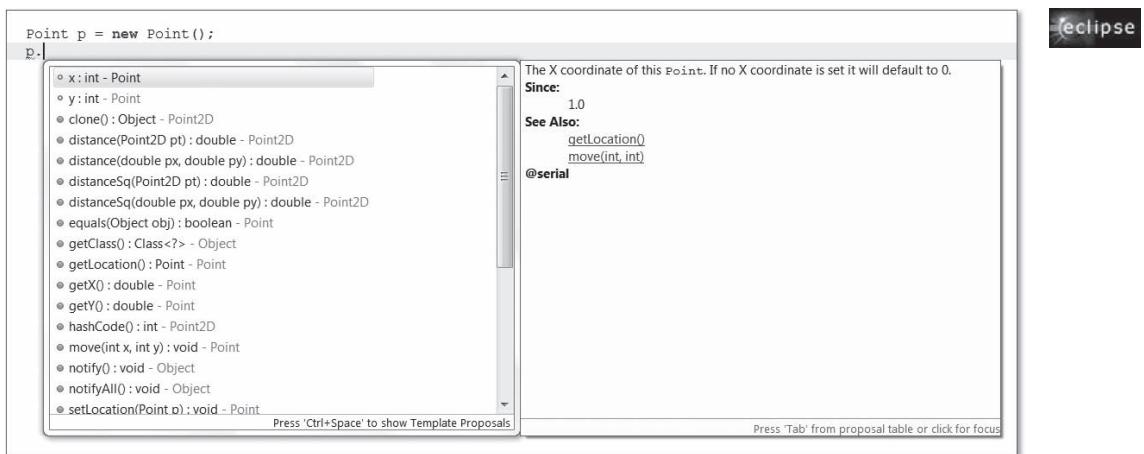


Abbildung 3.3: **Strg + Leertaste** zeigt an, welche Eigenschaften eine Referenz ermöglicht. Eine Auswahl mit **←** wählt die Eigenschaft aus und setzt insbesondere bei Methoden den Cursor zwischen das Klammerpaar.

Ein Methodenaufruf gestaltet sich genauso einfach wie ein Attributzugriff. Hinter dem Ausdruck mit der Referenz folgt nach dem Punkt der Methodenname.

Tür und Spieler auf dem Spielbrett

Punkt-Objekte erscheinen auf den ersten Blick als mathematische Konstrukte, doch sie sind allgemein nutzbar. Alles, was eine Position im zweidimensionalen Raum hat, lässt sich gut durch ein Punkt-Objekt repräsentieren. Der Punkt speichert für uns ja x und y, und hätten wir keine Punkt-Objekte, so müssten wir immer x und y extra speichern.

Nehmen wir an, wir wollen einen Spieler und eine Tür auf ein Spielbrett setzen. Natürlich haben die beiden Objekte Positionen. Ohne Objekte würde eine Speicherung der Koordinaten vielleicht so aussehen:

```

int playerX;
int playerY;
int doorX;
int doorY;

```

Die Modellierung ist nicht optimal, da wir mit der Klasse Point eine viel bessere Abstraktion haben, die zudem noch hübsche Methoden anbietet.

Ohne Abstraktion nur die nackten Daten	Kapselung der Zustände in ein Objekt
int playerX; int playerY;	java.awt.Point player;
int doorX; int doorY;	java.awt.Point door;

Tabelle 3.2: Objekte kapseln Zustände

Das folgende Beispiel erzeugt zwei Punkte, die die x/y-Koordinate eines Spielers und einer Tür auf einem Spielbrett repräsentieren. Nachdem die Punkte erzeugt wurden, werden die Koordinaten gesetzt, und es wird weiterhin getestet, wie weit der Spieler und die Tür voneinander entfernt sind:

Listing 3.1: PlayerAndDoorAsPoints.java

```
class PlayerAndDoorAsPoints
{
    public static void main( String[] args )
    {
        java.awt.Point player = new java.awt.Point();
        player.x = player.y = 10;

        java.awt.Point door = new java.awt.Point();
        door.setLocation( 10, 100 );

        System.out.println( player.distance( door ) ); // 90.0
    }
}
```

Im ersten Fall belegen wir die Variablen x, y des Spieles explizit. Im zweiten Fall setzen wir nicht direkt die Objektzustände über die Variablen, sondern verändern die Zustände über die Methode `setLocation()`. Die beiden Objekte besitzen eigene Koordinaten und kommen sich nicht in die Quere.



Abbildung 3.4: Die Abhangigkeit, dass eine Klasse einen `java.awt.Point` nutzt, zeigt das UML-Diagramm mit einer gestrichelten Linie an. Attribute und Operationen von Point sind nicht dargestellt.

toString()

Die Methode `toString()` liefert als Ergebnis ein String-Objekt, das den Zustand des Punkts preisgibt. Sie ist insofern besonders, als dass es immer auf jedem Objekt eine `toString()`-Methode gibt – nicht in jedem Fall ist die Ausgabe allerdings sinnvoll.

Listing 3.2: PointToStringDemo.java

```

class PointToStringDemo
{
    public static void main( String[] args )
    {
        java.awt.Point player = new java.awt.Point();
        java.awt.Point door   = new java.awt.Point();
        door.setLocation( 10, 100 );

        System.out.println( player.toString() ); // java.awt.Point[x=0,y=0]
        System.out.println( door );           // java.awt.Point[x=10,y=100]
    }
}
  
```

Tipp

Anstatt fur die Ausgabe explizit `println(obj.toString())` aufzurufen, funktioniert auch ein `println(obj)`. Das liegt daran, dass die Signatur `println(Object)` jedes beliebige Objekt als Argument akzeptiert und auf diesem Objekt automatisch die `toString()`-Methode aufruft.

Nach dem Punkt geht's weiter

Die Methode `toString()` liefert, wie wir gesehen haben, als Ergebnis ein String-Objekt:

```
java.awt.Point p = new java.awt.Point();
String s = p.toString();
System.out.println( s );                                // java.awt.Point[x=0,y=0]
```

Das String-Objekt besitzt selbst wieder Methoden. Eine davon ist `length()`, die die Länge der Zeichenkette liefert:

```
System.out.println( s.length() );                      // 23
```

Das Erfragen des String-Objekts und seiner Länge können wir zu einer Anweisung verbinden:

```
java.awt.Point p = new java.awt.Point();
System.out.println( p.toString().length() ); // 23
```

Objekterzeugung ohne Variablenzuweisung

Bei der Nutzung von Objekteigenschaften muss der Typ links vom Punkt immer eine Referenz sein. Ob die Referenz nun aus einer Variablen kommt, oder on-the-fly erzeugt wird, ist egal. Damit folgt, dass

```
java.awt.Point p = new java.awt.Point();
System.out.println( p.toString().length() );                // 23
```

genau das Gleiche bewirkt wie:

```
System.out.println( new java.awt.Point().toString().length() ); // 23
```

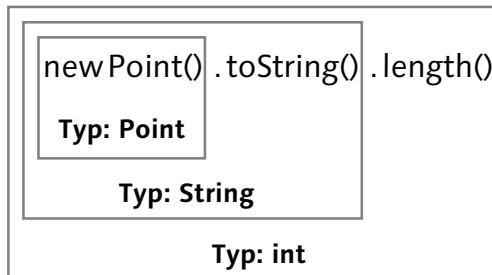


Abbildung 3.5: Jede Schachtelung ergibt einen neuen Typ.

Im Prinzip funktioniert auch Folgendes:

```
new java.awt.Point().x = 1;
```

Dies ist hier allerdings unsinnig, da zwar das Objekt erzeugt und ein Attribut gesetzt wird, anschließend das Objekt aber für den Garbage-Collector wieder Freiwild ist.

Beispiel

zB

Finde über ein File-Objekt heraus, wie groß eine Datei ist:

```
long size = new java.io.File( "file.txt" ).length();
```

Die Rückgabe der File-Methode length() ist die Länge der Datei in Bytes.

3.4.5 Überblick über Point-Methoden

Ein paar Methoden der Klasse Point kamen schon vor, und die API-Dokumentation zählt selbstverständlich alle Methoden auf. Die interessanteren sind:

```
class java.awt.Point
```

- double `getx()`
- double `getY()`
Liefert die x- bzw. y-Koordinate.
- void `setLocation(double x, double y)`
Setzt gleichzeitig die x/y-Koordinate.
- boolean `equals(Object obj)`
Prüft, ob ein anderer Punkt die gleichen Koordinaten besitzt. Dann ist die Rückgabe true, sonst false. Wird etwas anderes als ein Point übergeben, so wird der Compiler das nicht bemäkeln, nur wird das Ergebnis dann immer false sein.

Ein paar Worte über Vererbung und die API-Dokumentation

Eine Klasse besitzt nicht nur eigene Eigenschaften, sondern erbt auch immer welche von ihren Eltern. Im Fall von Point ist die Oberklasse Point2D – so sagt es die API-Dokumentation. Selbst Point2D erbt von Object, einer magischen Klasse, die alle Java-Klassen als Oberklasse haben. Der Vererbung widmen wir später noch einen sehr ausführlichen Abschnitt, aber es ist jetzt schon wichtig zu verstehen, dass die Oberklasse Attribute und Methoden vererbt. Sie sind in der API-Dokumentation einer Klasse nur kurz im Block

»Methods inherited from...« aufgeführt und gehen schnell unter. Für Entwickler ist es unabdingbar, nicht nur bei den Methoden der Klasse selbst zu schauen, sondern auch bei den geerbten Methoden. Bei Point sind es also nicht nur die Methoden dort selbst, sondern auch noch die Methoden aus Point2D und Object.

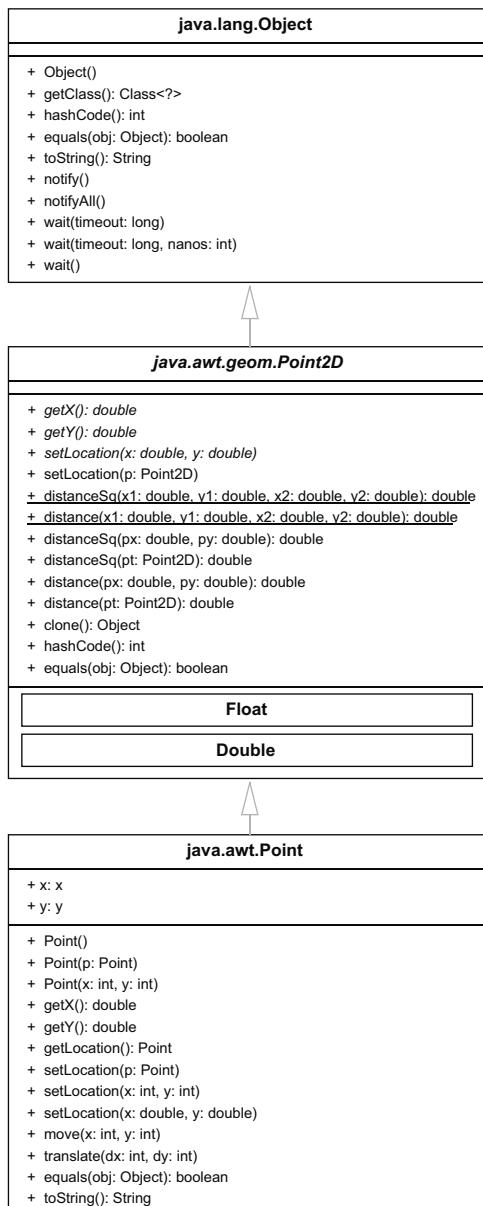


Abbildung 3.6: Vererbungshierarchie bei Point2D

Nehmen wir uns einige Methoden der Oberklasse vor. Die Klassendeklaration von `Point` trägt ein `extends Point2D`, was explizit klarmacht, dass es eine Oberklasse gibt:¹¹

```
class java.awt.Point
    extends Point2D
```

- `static double distance(double x1, double y1, double x2, double y2)`
Berechnet den Abstand zwischen den gegebenen Punkten nach der euklidischen Distanz.
- `double distance(double x, double y)`
Berechnet den Abstand des aktuellen Punktes zu angegebenen Koordinaten.
- `double distance(Point2D pt)`
Berechnet den Abstand des aktuellen Punktes zu den Koordinaten des übergebenen Punktes.

Sind zwei Punkte gleich?

Ob zwei Punkte gleich sind, sagt uns die `equals()`-Methode. Die Anwendung ist einfach. Stellen wir uns vor, wir wollen Koordinaten für einen Spieler, eine Tür und eine Schlange verwalten und dann testen, ob der Spieler auf der Tür steht und die Schlange auf der Position des Spielers:

Listing 3.3: PointEqualsDemo.java

```
class Distances
{
    public static void main( String[] args )
    {
        java.awt.Point player = new java.awt.Point();
        player.x = player.y = 10;

        java.awt.Point door = new java.awt.Point();
        door.setLocation( 10, 10 );

        System.out.println( player.equals( door ) );    // true
        System.out.println( door.equals( player ) );    // true
    }
}
```

¹¹ Damit ist die Klassendeklaration noch nicht vollständig, da ein `implements Serializable` fehlt, doch das soll uns jetzt erst einmal egal sein.

```

java.awt.Point snake = new java.awt.Point();
snake.setLocation( 20, 22 );

System.out.println( snake.equals( door ) );    // false
}
}

```

Da Spieler und Tür die gleichen Koordinaten besitzen, liefert equals() die Rückgabe true. Dabei ist es egal, ob wir den Spieler mit der Tür oder die Tür mit dem Spieler vergleichen – das Ergebnis bei equals() sollte immer symmetrisch sein.

Eine andere Testmöglichkeit ergibt sich durch distance(), denn ist der Abstand der Punkte null, so liegen die Punkte natürlich aufeinander und haben keinen Abstand.

Listing 3.4: Distances.java

```

class Distances
{
    public static void main( String[] args )
    {
        java.awt.Point player = new java.awt.Point();
        player.setLocation( 10, 10 );
        java.awt.Point door = new java.awt.Point();
        door.setLocation( 10, 10 );
        java.awt.Point snake = new java.awt.Point();
        snake.setLocation( 20, 10 );

        System.out.println( player.distance( door ) );    // 0.0
        System.out.println( player.distance( snake ) );    // 10.0
    }
}

```

Auf dem Spieler rufen wir die Methode distance() auf und übergeben den Verweis auf die Tür. Ob wir den Abstand vom Spieler zur Tür berechnen lassen oder den Abstand von der Tür zum Spieler, kommt wie bei equals() auf dasselbe raus.

3.4.6 Konstruktoren nutzen

Werden Objekte mit dem `new`-Operator angelegt, so wird ein Konstruktor aufgerufen. Der ist eine Art Methode mit besonderer Signatur. Ein Konstruktor hat keinen Rückgabewert und trägt auch denselben Namen wie die Klasse. Beim Anlegen eines Objekts sollen in der Regel die Objektvariablen initialisiert werden. Diese Initialisierung wird dazu in den Konstruktor gesetzt, um sicherzustellen, dass das neue Objekt einen sinnvollen Anfangszustand aufweist.

Aus der API-Dokumentation von `Point` sind drei Konstruktoren abzulesen:

```
class java.awt.Point
extends Point2D
```

- `Point()`
Legt einen Punkt mit den Koordinaten (0, 0) an.
- `Point(int x, int y)`
Legt einen neuen Punkt an und initialisiert ihn mit den Werten aus x und y.
- `Point(Point p)`
Legt einen neuen Punkt an und initialisiert ihn mit den gleichen Koordinaten, die der übergebene Punkt hat.

Ein Konstruktor ohne Argumente ist der *Standard-Konstruktor* (auch *Default-Konstruktor*, selten auch *No-Arg-Konstruktor* genannt). Jede Klasse kann höchstens einen Standard-Konstruktor besitzen, es kann aber auch sein, dass eine Klasse keinen Standard-Konstruktor deklariert, sondern nur Konstruktoren mit Parametern.

Beispiel

zB

Die Tabelle zeigt drei Varianten, ein `Point`-Objekt mit denselben Koordinaten (1/2) anzulegen (`java.awt.Point` ist mit `Point` abgekürzt):

<code>Point p =</code>	<code>Point q =</code>	<code>Point r =</code>
<code> new Point();</code>	<code> new Point(1,2);</code>	<code> new Point(q);</code>
<code>p.setLocation(1,2);</code>		

Der erste Konstruktor ist der Standard-Konstruktor, der zweite und der dritte sind parametisierte Konstruktoren.

3.5 ZZZZZnake

Ein Klassiker aus dem Genre der Computerspiele ist *Snake*. Auf dem Bildschirm gibt es den Spieler, eine Schlange, Gold und eine Tür. Die Tür und das Gold sind fest, den Spieler können wir bewegen, und die Schlange bewegt sich selbstständig auf den Spieler zu. Wir müssen versuchen, die Spielfigur zum Gold zu bewegen und dann zur Tür. Wenn die Schlange uns vorher erwischt, haben wir Pech gehabt und das Spiel ist verloren.

Vielleicht hört sich das auf den ersten Blick komplex an, aber wir haben alle Bausteine zusammen, um dieses Spiel zu programmieren.

- Spieler, Schlange, Gold und Tür sind `Point`-Objekte, die mit Koordinaten vorkonfiguriert sind.
- Eine Schleife läuft alle Koordinaten ab. Ist ein Spieler, die Tür, die Schlange oder Gold »getroffen«, gibt es eine symbolische Darstellung der Figuren.
- Wie testen drei Bedingungen für den Spielstatus:
 - a) Hat der Spieler das Gold eingesammelt und steht auf der Tür? (Das Spiel ist zu Ende.)
 - b) Beißt die Schlange den Spieler? (Das Spiel ist verloren.)
 - c) Sammelt der Spieler Gold ein?
- Mit dem Scanner können wir auf Tastendrücke reagieren und den Spieler auf dem Spielbrett bewegen.
- Die Schlange muss sich in Richtung des Spielers bewegen. Während der Spieler sich nur entweder horizontal oder vertikal bewegen kann, erlauben wir der Schlange, sich diagonal zu bewegen.

In Quellcode sieh das so aus:

Listing 3.5: ZZZZZnake.java

```
import java.awt.Point;

public class ZZZZZnake
{
    public static void main( String[] args )
    {
        java.awt.Point playerPosition = new java.awt.Point( 10, 9 );
        java.awt.Point snakePosition  = new java.awt.Point( 30, 2 );
        java.awt.Point goldPosition   = new java.awt.Point( 6, 6 );
```

```
java.awt.Point doorPosition = new java.awt.Point( 0, 5 );
boolean rich = false;

while ( true )
{
    // Raster mit Figuren zeichnen

    for ( int y = 0; y < 10; y++ )
    {
        for ( int x = 0; x < 40; x++ )
        {
            Point p = new Point( x, y );
            if ( playerPosition.equals( p ) )
                System.out.print( '&' );
            else if ( snakePosition.equals( p ) )
                System.out.print( 'S' );
            else if ( goldPosition.equals( p ) )
                System.out.print( '$' );
            else if ( doorPosition.equals( p ) )
                System.out.print( '#' );
            else System.out.print( '.' );
        }
        System.out.println();
    }

    // Status feststellen

    if ( rich && playerPosition.equals( doorPosition ) )
    {
        System.out.println( "Gewonnen!" );
        break;
    }
    if ( playerPosition.equals( snakePosition ) )
    {
        System.out.println( "ZZZZZZZ. Die Schlange hat dich!" );
        break;
    }
}
```

```

        }

        if ( playerPosition.equals( goldPosition ) )
        {
            rich = true;
            goldPosition.setLocation( -1, -1 );
        }

        // Konsoleneingabe und Spielerposition verändern

        switch ( new java.util.Scanner( System.in ).next().charAt( 0 ) )
        {
            case 'h' : playerPosition.y = Math.max( 0, playerPosition.y - 1 ); break;
            case 't' : playerPosition.y = Math.min( 9, playerPosition.y + 1 ); break;
            case 'l' : playerPosition.x = Math.max( 0, playerPosition.x - 1 ); break;
            case 'r' : playerPosition.x = Math.min( 39, playerPosition.x + 1 ); break;
        }

        // Schlange bewegt sich in Richtung Spieler

        if ( playerPosition.x < snakePosition.x )
            snakePosition.x--;
        else if ( playerPosition.x > snakePosition.x )
            snakePosition.x++;
        if ( playerPosition.y < snakePosition.y )
            snakePosition.y--;
        else if ( playerPosition.y > snakePosition.y )
            snakePosition.y++;

    } // end while
}

}

```

Die Point-Eigenschaften, die wir nutzen, sind:

- x, y: Der Spieler und die Schlange werden bewegt, und die Koordinaten müssen neu gesetzt werden.
- setLocation(): Ist das Gold aufgesammelt, setzen wie die Koordinaten so, dass die Koordinate vom Gold nicht mehr auf unserem Raster liegt.

- `equals()`: Testet, ob ein Punkt auf einem anderen Punkt steht. Der gleiche Test könnte mit `p.distance(q) == 0` durchgeführt werden, was sogar noch besser wäre, da die Point-Klasse intern mit dem Datentyp `double` statt `int` arbeitet und exakte Vergleiche immer etwas problematisch sind. Mit `distance()` können wir eine kleine Abweichung erlauben. Der Test `p.equals(q)` wäre dann etwa `p.distance(q) < 0.0001`.

Erweiterung

Wer Lust hat, an der Aufgabe noch ein wenig weiter zu programmieren, der kann Folgendes tun:

- Statt nur einem Stück Gold soll es zwei Stücke geben.
- Statt einer Schlange soll es zwei Schlangen geben.
- Mit zwei Schlangen und zwei Stücken Gold kann es etwas eng für den Spieler werden. Er soll daher 5 Züge machen können, ohne dass die Schlangen sich bewegen.
- Wenn der Spieler ein Goldstück einsammelt, soll die Länge der Schlange um eins schrumpfen.
- Für Vorarbeiter: Das Programm, das bisher nur eine Methode ist, soll in verschiedene Untermethoden aufgespalten werden.

3.6 Kompilationseinheiten, Imports und Pakete schnüren

Ein *Paket* ist eine Gruppe thematisch zusammengehöriger Typen. Pakete könnten Unterpakete besitzen, die in der Angabe durch einen Punkt getrennt werden. Die Gruppierung lässt sich sehr gut an der Java-Bibliothek beobachten, wo zum Beispiel eine Klasse `File` und `RandomAccessFile` dem Paket `java.io` angehören, denn Dateien und Möglichkeiten zum wahlfreien Zugriff auf Dateien gehören eben zur Ein-/Ausgabe. Ein Punkt und ein `Polygon`, repräsentiert durch die Klassen `Point` und `Polygon`, gehören in das Paket für grafische Oberflächen, und das ist das Paket `java.awt`.

Die Paketnamen `java`, `javax`

Die Klassen der Standardbibliothek sitzen in Paketen, die mit `java` und `javax` beginnen. So befindet sich `java.awt.Point` in einem Paket der Standardbibliothek, was durch den Teil `java` zu erkennen ist. Wenn jemand eigene Klassen in Pakete mit dem Präfix `java` setzen würde, etwa `java.ui`, würde er damit Verwirrung stiften, da nicht mehr nachvoll-

ziehbar ist, ob das Paket Bestandteil jeder Distribution ist. Klassen, die mit `javax` beginnen, müssen nicht zwingend zur Java SE gehören, aber dazu folgt mehr in Abschnitt 11.1.1, »Übersicht über die Pakete der Standardbibliothek«.

3.6.1 Volle Qualifizierung und import-Deklaration

Um die Klasse `Point`, die im Paket `java.awt` liegt, außerhalb des Pakets `java.awt` zu nutzen – und das ist für uns Nutzer immer der Fall –, muss sie dem Compiler mit der gesamten Paketangabe bekannt gemacht werden. Hierzu reicht der Klassename allein nicht aus, denn es kann ja sein, dass der Klassename mehrdeutig ist und eine Klassendeklaration in unterschiedlichen Paketen existiert. (In der Java-Bibliothek gibt es dazu einige Beispiele, etwa `java.util.Date` und `java.sql.Date`.)

Um dem Compiler die präzise Zuordnung einer Klasse zu einem Paket zu ermöglichen, gibt es zwei Möglichkeiten: Zum einen lassen sich die Typen voll qualifizieren, wie wir das bisher getan haben. Eine alternative und praktischere Möglichkeit besteht darin, den Compiler mit einer `import`-Deklaration auf die Typen im Paket aufmerksam zu machen:

Listing 3.6: AwtWithoutImport.java

```
class AwtWithoutImport
{
    public static void main( String[] args )
    {
        java.awt.Point p = new java.awt.Point();
        java.awt.Polygon t = new java.awt.Polygon();
        t.addPoint( 10, 10 );
        t.addPoint( 10, 20 );
        t.addPoint( 20, 10 );

        System.out.println( p );
        System.out.println( t.contains(15, 15) );
    }
}
```

Listing 3.7: AwtWithImport.java

```
import java.awt.Point;
import java.awt.Polygon;

class AwtWithImport
{
    public static void main( String[] args )
    {
        Point p = new Point();
        Polygon t = new Polygon();
        t.addPoint( 10, 10 );
        t.addPoint( 10, 20 );
        t.addPoint( 20, 10 );

        System.out.println( p );
        System.out.println( t.contains(15, 15) );
    }
}
```

Tabelle 3.3: Programm ohne und mit import-Deklaration

Während der Quellcode auf der linken Seite die volle Qualifizierung verwendet und jeder Verweis auf einen Typ mehr Schreibarbeit kostet, ist im rechten Fall beim `import` nur der Klassenname genannt und die Paketangabe in ein `import` »ausgelagert«. Kommt der Compiler zu einer Anweisung wie `Point p = new Point();`, findet er die Deklaration einer Klasse `Point` im Paket `java.awt` und kennt damit die für ihn unabkömmliche absolute Qualifizierung.

Hinweis

Die Typen aus `java.lang` sind automatisch importiert, sodass zum Beispiel ein `import java.lang.String;` nicht nötig ist.

3.6.2 Mit `import p1.p2.*` alle Typen eines Pakets erreichen

Greift eine Java-Klasse auf mehrere andere Typen des gleichen Pakets zurück, kann die Anzahl der `import`-Deklarationen groß werden. In unserem Beispiel nutzen wir mit `Point` und `Polygon` nur zwei Klassen aus `java.awt`, aber es lässt sich schnell ausmalen, was passt, wenn aus dem Paket für grafische Oberflächen zusätzlich Fenster, Beschriftungen, Schaltflächen, Schieberegler und so weiter eingebunden werden. Die Lösung in diesem Fall ist ein `*`, das das letzte Glied in einer `import`-Deklaration sein darf:

```
import java.awt.*;
import java.io.*;
```

Mit dieser Syntax kennt der Compiler alle Typen im Paket `java.awt` und `java.io`, sodass eine Klasse `Point` und `Polygon` genau bekannt ist, wie auch die Klasse `File`.

Hinweis

Das `*` ist nur in der letzten Hierarchie erlaubt und gilt immer für alle Typen in diesem Paket. Syntaktisch falsch sind:

```
import *;           // ☹ Syntax error on token "*", Identifier expected
import java.awt.Po*; // ☹ Syntax error on token "*", delete this token
```

Eine Anweisung wie `import java.*;` ist zwar syntaktisch korrekt, aber dennoch ohne Wirkung, denn direkt im Paket `java` gibt es keine Typendeklarationen, sondern nur Unterpakete.



Hinweis (Forts.)

Die `import`-Deklaration bezieht sich nur auf ein Verzeichnis (in der Annahme, dass die Pakete auf das Dateisystem abgebildet werden) und schließt die Unterverzeichnisse nicht mit ein.

Das `*` verkürzt zwar die Anzahl der individuellen `import`-Deklarationen, es ist aber gut, zwei Dinge im Kopf zu behalten:

- Falls zwei unterschiedliche Pakete einen gleichlautenden Typ beherbergen, etwa `Date` in `java.util` und `java.sql`, so kommt es bei der Verwendung des Typs zu einem Übersetzungsfehler. Hier muss voll qualifiziert werden.
- Die Anzahl der `import`-Deklarationen sagt etwas über den Grad der Komplexität aus. Je mehr `import`-Deklarationen es gibt, desto größer werden die Abhängigkeiten zu anderen Klassen, was im Allgemeinen ein Alarmzeichen ist. Zwar zeigen grafische Tools die Abhängigkeiten genau an, doch ein `import *` kann diese erst einmal verstecken.

3.6.3 Hierarchische Strukturen über Pakete

Ein Java-Paket ist eine logische Gruppierung von Klassen. Pakete lassen sich in Hierarchien ordnen, sodass in einem Paket wieder ein anderes Paket liegen kann; das ist genauso wie bei der Verzeichnisstruktur des Dateisystems. In der Standardbibliothek ist das Paket `java` ein Hauptzweig, aber das gilt auch für `javax`. Unter dem Paket `java` liegen dann zum Beispiel die Pakete `awt` und `util`, und unter `javax` liegen dann `swing` und sonstige Unterpakete.

Die zu einem Paket gehörenden Klassen befinden sich normalerweise¹² im gleichen Verzeichnis. Der Name des Pakets ist dann gleich dem Namen des Verzeichnisses (und natürlich umgekehrt). Statt des Verzeichnistrenners (etwa `»/«` oder `»\«`) steht ein Punkt.

Nehmen wir folgende Verzeichnisstruktur mit einer Hilfsklasse an:

```
com/tutego/
com/tutego/DatePrinter.class
```

¹² Ich schreibe »normalerweise«, da die Paketstruktur nicht zwingend auf Verzeichnisse abgebildet werden muss. Pakete könnten beispielsweise vom Klassenlader aus einer Datenbank gelesen werden. Im Folgenden wollen wir aber immer von Verzeichnissen ausgehen.

Hier ist der Paketname `com.tutego` und somit der Verzeichnisname `com/tutego/`. Umlaute und Sonderzeichen sollten vermieden werden, da sie auf dem Dateisystem immer wieder für Ärger sorgen. Aber Bezeichner sollten ja sowieso immer auf Englisch sein.

Der Aufbau von Paketnamen

Prinzipiell kann ein Paketname beliebig sein, doch Hierarchien bestehen in der Regel aus umgedrehten Domänennamen. Aus der Domäne zur Webseite `http://tutego.com` wird also `com.tutego`. Diese Namensgebung gewährleistet, dass Klassen auch weltweit eindeutig bleiben. Ein Paketname wird in aller Regel komplett kleingeschrieben.

3.6.4 Die package-Deklaration

Um die Klasse `DatePrinter` in ein Paket `com.tutego` zu setzen, müssen zwei Dinge gelten:

- Sie muss sich physikalisch in einem Verzeichnis befinden, also in `com/tutego/`.
- Der Quellcode enthält zuoberst eine package-Deklaration.

Die package-Deklaration muss ganz am Anfang stehen, sonst gibt es einen Übersetzungsfehler (selbstverständlich lassen sich Kommentare vor die package-Deklaration setzen):

Listing 3.8: com/tutego/DatePrinter.java

```
package com.tutego;

import java.util.Date;

public class DatePrinter
{
    public static void printCurrentDate()
    {
        System.out.printf( "%tD%n", new Date() );
    }
}
```

Hinter die package-Deklaration kommen wie gewohnt `import`-Anweisungen und die Typdeklarationen.

Um die Klasse zu nutzen, bieten sich wie bekannt zwei Möglichkeiten: einmal über die volle Qualifizierung und einmal über die `import`-Deklaration. Die erste Variante:

Listing 3.9: DatePrinterUser1.java

```
public class DatePrinterUser1
{
    public static void main( String[] args )
    {
        com.tutego.DatePrinter.printCurrentDate();      // 05/31/11
    }
}
```

Und die Variante mit der `import`-Deklaration:

Listing 3.10: DatePrinterUser2.java

```
import com.tutego.DatePrinter;

public class DatePrinterUser2
{
    public static void main( String[] args )
    {
        DatePrinter.printCurrentDate();      // 05/31/11
    }
}
```

3.6.5 Unbenanntes Paket (default package)

Falls eine Klasse ohne Paket-Angabe implementiert wird, befindet sie sich standardmäßig im *unbenannten Paket* (engl. *unnamed package*) oder *Default-Paket*. Es ist eine gute Idee, eigene Klassen immer in Paketen zu organisieren. Das erlaubt auch feinere Sichtbarkeiten, und Konflikte mit anderen Unternehmen und Autoren werden vermieden. Es wäre ein großes Problem, wenn a) jedes Unternehmen unübersichtlich alle Klassen in das unbenannte Paket setzt und dann b) versucht, die Bibliotheken auszutauschen: Konflikte wären vorprogrammiert.

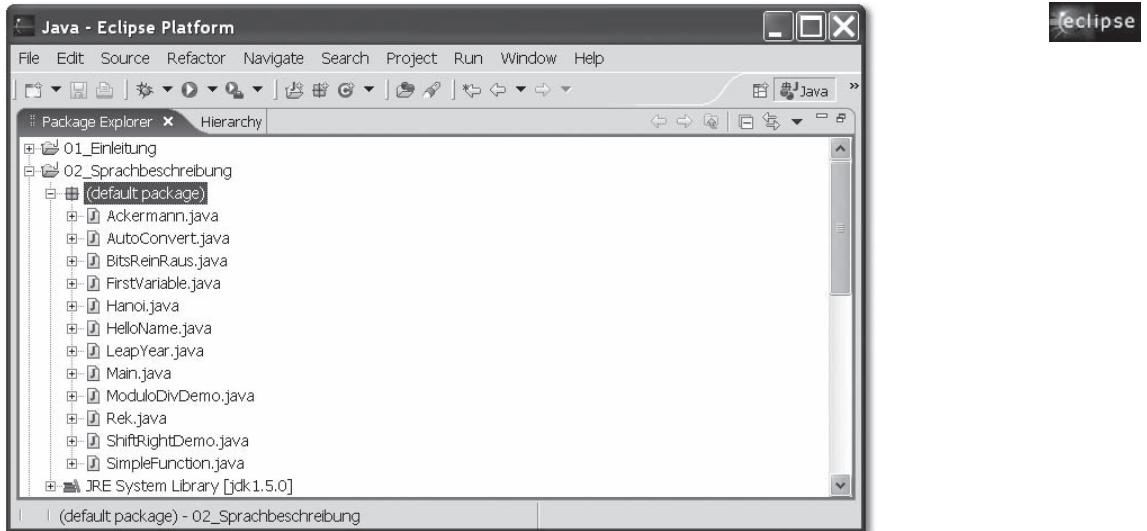


Abbildung 3.7: Das Verzeichnis »default package« steht in Eclipse für das unbenannte Paket.

Eine im Paket befindliche Klasse kann jede andere sichtbare Klasse aus anderen Paketen importieren, aber keine Klassen aus dem unbenannten Paket. Nehmen wir Chocolate im Paket com.tutego und Sugar im unbenannten Paket an:

*Sugar.class
com/tutego/Chocolate.class*

Die Klasse Chocolate kann Sugar nicht nutzen, da Klassen aus dem unbenannten Paket nicht für Unterpakete sichtbar sind. Nur andere Klassen im unbenannten Paket können Klassen im unbenannten Paket nutzen.

Stände nun Sugar in einem Paket – was auch ein Oberpaket sein kann! –, so wäre das wiederum möglich, und Chocolate könnte Sugar importieren.

*com/Sugar.class
com/tutego/Chocolate.class*

3.6.6 Klassen mit gleichen Namen in unterschiedlichen Paketen *

Ein Problem gibt es bei mehreren gleich benannten Klassen in unterschiedlichen Paketen. Hier ist eine volle Qualifizierung nötig. So gibt es in den Paketen java.awt und java.util eine Liste. Ein einfaches `import java.awt.*` und `java.util.*` hilft da nicht, weil der Compiler nicht weiß, ob die GUI-Komponente oder die Datenstruktur gemeint ist.

Auch sagt ein `import` nichts darüber aus, ob die Klassen in der importierenden Datei jemals gebraucht werden. Das Gleiche gilt für die Klasse `Date`, die einmal in `java.util` und einmal in `java.sql` zu finden ist. Lustigerweise erweitert `java.sql.Date` die Klasse `java.util.Date`. Dass der Compiler hier nicht durcheinanderkommt, ist ganz einfach dadurch zu erklären, dass er die Klassen nicht nur anhand ihres Namens unterscheidet, sondern vielmehr auch anhand ihrer Pakete. Der Compiler betrachtet intern immer eine volle Qualifizierung.

3.6.7 Compilationseinheit (Compilation Unit)

Die `package`- und `import`-Deklarationen gehören nicht wirklich zu der Typdeklaration, die nur ein `class C { }` oder verwandte Typdeklarationen umfasst. Genau genommen sind dies alles Bestandteile einer *Compilationseinheit (Compilation Unit)*. So besteht eine Compilationseinheit aus höchstens einer Paketdeklaration, beliebig vielen `import`-Deklarationen und beliebig vielen Typdeklarationen. Ein Paket ist letztendlich eine Sammlung aus Compilationseinheiten.

3.6.8 Statischer Import *

Das `import` hat in Java die Bedeutung, den Compiler über die Pakete zu informieren, so dass eine Klasse nicht mehr voll qualifiziert werden muss, wenn sie im `import`-Teil explizit aufgeführt wird oder wenn das Paket der Klasse genannt ist.

Falls eine Klasse statische Methoden oder Konstanten vorschreibt, werden ihre Eigenschaften immer über den Klassennamen angesprochen. Es gibt nun mit dem *statischen Import* die Möglichkeit, die Klasseneigenschaften wie eigene statische Methoden oder Variablen ohne Klassennamen sofort zu nutzen.

Praktisch ist das zum Beispiel für die Bildschirmausgabe, wenn die statische Variable `out` aus `System` eingebunden wird:

```
import static java.lang.System.out;
```

Bei der sonst üblichen Ausgabe über `System.out.printXXX()` kann nach dem statischen Import der Klassenname entfallen, und es bleibt beim `out.printXXX()`:

Listing 3.11: StaticImport.java

```
import static java.lang.System.out;  
import static javax.swing.JOptionPane.showInputDialog;
```

```

import static java.lang.Integer.parseInt;
import static java.lang.Math.max;
import static java.lang.Math.min;

class StaticImport
{
    public static void main( String[] args )
    {
        int i = parseInt( showInputDialog( "Erste Zahl" ) );
        int j = parseInt( showInputDialog( "Zweite Zahl" ) );
        out.printf( "%d ist größer oder gleich %d.%n",
                    max(i, j), min(i, j) );
    }
}

```

Mehrere Typen statisch importieren

Der statische Import

```

import static java.lang.Math.max;
import static java.lang.Math.min;

```

bindet die statische `max()/min()`-Methode ein. Besteht Bedarf an weiteren statischen Methoden, gibt es neben der individuellen Aufzählung eine Wildcard-Variante:

```
import static java.lang.Math.*;
```

Auch wenn Java seit Version 5 diese Möglichkeit bietet, sollte der Einsatz maßvoll erfolgen. Die Möglichkeit der statischen Importe wird dann nützlicher, wenn Klassen Konstanten nutzen wollen.

Hinweis

Eine Objektmethode aus der eigenen Klasse überdeckt statische importierte Methoden, was im Fall der `toString()`-Methode auffällt, die statisch aus der Utility-Klasse `Arrays` eingebunden werden kann. Der Compiler interpretiert `toString()` als Aufruf einer Objektmethode (auch dann, wenn die aufrufende Methode selbst statisch ist).



3.6.9 Eine Verzeichnisstruktur für eigene Projekte *

Neben der Einteilung in Pakete für das eigene Programm ist es auch sinnvoll, die gesamte Applikation in verschiedenen Verzeichnissen aufzubauen. Im Allgemeinen finden sich drei wichtige Hauptverzeichnisse: *src* für die Quellen, *lib* für externe Bibliotheken, auf die das Programm aufbaut, und *bin* (oder *build*) für die erzeugten Klassen-Dateien. Das Verzeichnis *src* lässt sich noch weiter unterteilen, etwa für Quellen, die Testfälle implementieren, oder für Beispiele:

```
src/
core/
examples/
test/
lib/
bin/
```

Mehr Anregungen zur Verzeichnisstruktur gibt die Webseite <http://java.sun.com/blueprints/code/projectconventions.html>.

3.7 Mit Referenzen arbeiten, Identität und Gleichheit

In Java gibt es mit `null` eine sehr spezielle Referenz, die Auslöser vieler Probleme ist. Doch ohne sie geht es nicht, und warum das so ist, wird der folgende Abschnitt zeigen. Anschließend wollen wir sehen, wie Objektvergleiche funktionieren und was der Unterschied zwischen Identität und Gleichheit ist.

3.7.1 Die null-Referenz

In Java gibt es drei spezielle Referenzen: `null`, `this` und `super`. (Wir verschieben `this` und `super` auf Kapitel 5, »Eigene Klassen schreiben«.) Das spezielle Literal `null` lässt sich zur Initialisierung von Referenzvariablen verwenden. Die `null`-Referenz ist typenlos, kann also jeder Referenzvariablen zugewiesen und jeder Methode übergeben werden, die ein Objekt erwartet.¹³

¹³ `null` verhält sich also so, als ob es ein Untertyp jedes anderen Typs wäre.

zB

3

Beispiel

Deklaration und Initialisierung zweier Objektvariablen mit null:

```
Point p = null;
String s = null;
System.out.println( null );
```

Die Konsolenausgabe über die letzte Zeile liefert kurz »null«.

Da es nur ein null gibt, ist zum Beispiel (Point) null == (String) null. Der Wert ist ausschließlich für Referenzen vorgesehen und kann in keinen primitiven Typ wie die Ganzzahl 0 umgewandelt werden.¹⁴

Mit null lässt sich eine ganze Menge machen. Der Haupteinsatzzweck sieht vor, damit uninitializede Referenzvariablen zu kennzeichnen, also auszudrücken, dass eine Referenzvariable auf kein Objekt verweist. In Listen oder Bäumen kennzeichnet null zum Beispiel das Fehlen eines gültigen Nachfolgers; null ist dann ein gültiger Indikator und kein Fehlerfall.

Auf null geht nix, nur die NullPointerException

Da sich hinter null kein Objekt verbirgt, ist es auch nicht möglich, eine Methode aufzurufen. Der Compiler kennt zwar den Typ jedes Objekts, aber erst die Laufzeitumgebung (JVM) weiß, was referenziert wird. Wird versucht, über die null-Referenz auf eine Eigenschaft eines Objekts zuzugreifen, löst eine JVM eine NullPointerException¹⁵ aus:

Listing 3.12: NullPointer.java

```
/* 1 */public class NullPointer
/* 2 */{
/* 3 */    public static void main( String[] args )
/* 4 */    {
/* 5 */        java.awt.Point p = null;
/* 6 */        String         s = null;
```

¹⁴ Hier unterscheiden sich C(++) und Java.

¹⁵ Der Name zeigt das Überbleibsel von Zeigern. Zwar haben wir es in Java nicht mit Zeigern zu tun, sondern mit Referenzen, doch heißt es NullPointerException und nicht NullReferenceException. Das erinnert daran, dass eine Referenz ein Objekt identifiziert und eine Referenz auf ein Objekt ein Pointer ist. Das .NET Framework ist hier konsequenter und nennt die Ausnahme NullReferenceException.

```

/* 7 */
/* 8 */     p.setLocation( 1, 2 );
/* 9 */     s.length();
/* 10 */
/* 11 */

```

Wir beobachten eine NullPointerException, denn das Programm bricht bei `p.setLocation()` mit folgender Ausgabe ab:

```

java.lang.NullPointerException
    at NullPointer.main(NullPointer.java:8)
Exception in thread "main"

```

Die Laufzeitumgebung teilt uns in der Fehlermeldung mit, dass sich der Fehler, die NullPointerException, in Zeile 8 befindet.

3.7.2 null-Referenzen testen

Mit dem Vergleichsoperator `==` oder dem Test auf Ungleichheit mit `!=` lässt sich leicht herausfinden, ob eine Referenzvariable wirklich ein Objekt referenziert oder nicht:

```

if ( object == null )
    // Variable referenziert nichts, ist aber gültig mit null initialisiert
else
    // Variable referenziert ein Objekt

```

null-Test und Kurzschluss-Operatoren

Wir wollen an dieser Stelle noch einmal auf die üblichen logischen Kurzschluss-Operatoren und den logischen, nicht kurzschließenden Operator zu sprechen kommen. Erstere werten Operanden nur so lange von links nach rechts aus, bis das Ergebnis der Operation feststeht. Auf den ersten Blick scheint es nicht viel auszumachen, ob alle Teilausdrücke ausgewertet werden oder nicht, in einigen Ausdrücken ist dies aber wichtig, wie das folgende Beispiel für die Variable `s` vom Typ `String` zeigt:

Listing 3.13: NullCheck.java

```

public class NullCheck
{
    public static void main( String[] args )

```

```

{
    String s = javax.swing.JOptionPane.showInputDialog( "Eingabe" );
    if ( s != null && ! s.isEmpty() )
        System.out.println( "Eingabe: " + s );
    else
        System.out.println( "Abbruch oder keine Eingabe" );
}
}

```

3

Die Rückgabe von `showInputDialog()` ist `null`, wenn der Benutzer den Dialog abbricht. Das soll unser Programm berücksichtigen. Daher testet die `if`-Bedingung, ob `s` überhaupt auf ein Objekt verweist, und prüft gleichzeitig, ob die Länge größer 0 ist. Dann folgt eine Ausgabe.

Diese Schreibweise tritt häufig auf, und der Und-Operator zur Verknüpfung muss ein Kurzschluss-Operator sein, da es in diesem Fall ausdrücklich darauf ankommt, dass die Länge nur dann bestimmt wird, wenn die Variable `s` überhaupt auf ein `String`-Objekt verweist und nicht `null` ist. Andernfalls bekämen wir bei `s.length()` eine `NullPointerException`, wenn jeder Teilausdruck ausgewertet würde und `s` gleich `null` wäre.

»null« in anderen Programmiersprachen



Ist Java eine pure objektorientierte Programmiersprache? Nein, da Java einen Unterschied zwischen primitiven Typen und Referenztypen macht. Nehmen wir für einen Moment an, dass es primitive Typen nicht gibt. Wäre Java dann eine reine objektorientierte Programmiersprache, bei der jede Referenz ein pures Objekt referenziert? Die Antwort ist immer noch nein, da es mit `null` etwas gibt, mit dem Referenzvariablen initialisiert werden können, aber was kein Objekt repräsentiert und keine Methoden besitzt. Und das kann bei der Dereferenzierung eine `NullPointerException` geben. Andere Programmiersprachen haben andere Lösungsansätze, und `null`-Referenzierungen sind nicht möglich. In der Sprache Ruby zum Beispiel ist immer alles ein Objekt. Wo Java mit `null` ein »nicht belegt« ausdrückt, macht das Ruby mit `nil`. Der feine Unterschied ist, dass `nil` ein Exemplar der Klasse `NilClass` ist, genau genommen ein Singleton, was es im System nur einmal gibt. `nil` hat auch ein paar öffentliche Methoden wie `to_s` (wie Javas `toString()`), was dann einen leeren String liefert. Mit `nil` gibt es keine `NullPointerException` mehr, aber natürlich immer noch einen Fehler, wenn auf diesem Objekt vom Typ `NilClass` eine Methode aufgerufen wird, die es nicht gibt.



»null« in anderen Programmiersprachen (Forts.)

In Objective-C, der Standardsprache für iPhone-Programme, gibt es das Null-Objekt `nil`. Üblicherweise passiert nichts, wenn eine Nachricht an das `nil`-Objekt gesendet wird; die Nachricht wird einfach ignoriert.¹⁶

3.7.3 Zuweisungen bei Referenzen

Eine Referenz erlaubt den Zugriff auf das referenzierte Objekt. Es kann durchaus mehrere Kopien dieser Referenz geben, die in Variablen mit unterschiedlichen Namen abgelegt sind – so wie eine Person von den Mitarbeitern als »Chefin« angesprochen wird, aber von ihrem Mann als »Schnuckiputzi«. Dies nennt sich auch *Alias*.

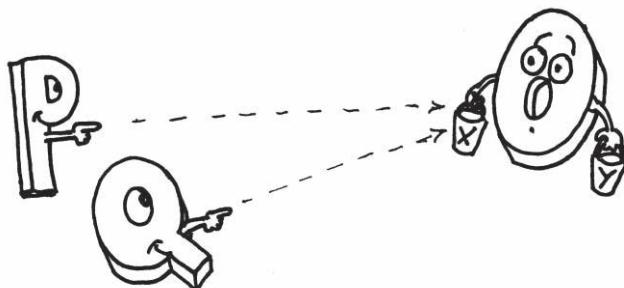
zB

Beispiel

Ein Punkt-Objekt wollen wir unter einem alternativen Variablenamen ansprechen:

```
Point p = new Point();
Point q = p;
```

Ein Punkt-Objekt wird erzeugt und mit der Variablen `p` referenziert. Die zweite Zeile speichert nun dieselbe Referenz in der Variablen `q`. Danach verweisen `p` und `q` auf das-selbe Objekt.



Verweisen zwei Objektvariablen auf das gleiche Objekt, hat das natürlich zur Konsequenz, dass über zwei Wege Objektzustände ausgelesen und modifiziert werden kön-

¹⁶ Es gibt auch Compiler wie den GCC, der mit der Option `-fno-nil-receivers` dieses Verhalten abschaltet, um schnelleren Maschinencode zu erzeugen. Denn letztendlich muss in Maschinencode immer ein Test stehen, der auf 0 prüft.

nen. Heißt die gleiche Person in der Firma »Chefin« und zu Hause »Schnuckiputzi«, wird der Mann sich freuen, wenn die Frau in der Firma keinen Stress hat.

Wir können das Beispiel auch gut bei Punkt-Objekten nachverfolgen. Zeigen `p` und `q` auf dasselbe Punkt-Objekt, können Änderungen über `p` auch über die Variable `q` beobachtet werden:

Listing 3.14: ItsTheSame.java

```
import java.awt.Point;

public class ItsTheSame
{
    public static void main( String[] args )
    {
        Point p = new Point();
        Point q = p;
        p.x = 10;
        System.out.println( q.x ); // 10
        q.y = 5;
        System.out.println( p.y ); // 5
    }
}
```

3.7.4 Methoden mit nicht-primitiven Parametern

Dass sich das gleiche Objekt unter zwei Namen (über zwei verschiedene Variablen) ansprechen lässt, können wir gut bei Methoden beobachten. Eine Methode, die über den Parameter eine Objektreferenz erhält, kann auf das übergebene Objekt zugreifen. Das bedeutet, die Methode kann dieses Objekt mit den angebotenen Methoden ändern oder auf die Attribute zugreifen.

Im folgenden Beispiel deklarieren wir zwei Methoden. Die erste Methode, `initializeToken(Point)`, soll einen Punkt mit Zufallskoordinaten initialisieren. Übergeben werden wir dann zwei `Point`-Objekte: einmal für einen Spieler und einmal für eine Schlange. Die zweite Methode, `printScreen(Point, Point)`, gibt das Spielfeld auf dem Bildschirm aus, und gibt dann, wenn die Koordinate einen Spieler trifft, ein »&« aus und bei der Schlange ein »S«.

Listing 3.15: DrawPlayerAndSnake.java

```
import java.awt.Point;

public class DrawPlayerAndSnake
{
    static void initializeToken( Point p )
    {
        int randomX = (int)(Math.random() * 40); // 0 <= x < 40
        int randomY = (int)(Math.random() * 10); // 0 <= y < 10
        p.setLocation( randomX, randomY );
    }

    static void printScreen( Point playerPosition,
                           Point snakePosition )
    {
        for ( int y = 0; y < 10; y++ )
        {
            for ( int x = 0; x < 40; x++ )
            {
                if ( playerPosition.distanceSq( x, y ) == 0 )
                    System.out.print( '&' );
                else if ( snakePosition.distanceSq( x, y ) == 0 )
                    System.out.print( 'S' );
                else System.out.print( '.' );
            }
            System.out.println();
        }
    }

    public static void main( String[] args )
    {
        Point playerPosition = new Point();
        Point snakePosition = new Point();
        System.out.println( playerPosition );
        System.out.println( snakePosition );
        initializeToken( playerPosition );
    }
}
```

```
    initializeToken( snakePosition );
    System.out.println( playerPosition );
    System.out.println( snakePosition );
    printScreen( playerPosition, snakePosition );
}
}
```

3

Die Ausgabe kann so aussehen:

In dem Moment, in dem `main()` die statische Methode `initializeToken()` aufruft, gibt es sozusagen zwei Namen für das Point-Objekt: `playerPosition` und `p`. Allerdings ist das nur innerhalb der virtuellen Maschine so, denn `initializeToken()` kennt das Objekt nur unter `p`, aber kennt die Variable `playerPosition` nicht. Bei `main()` ist es umgekehrt: Nur der Variablenname `playerPosition` ist in `main()` bekannt, er hat aber vom Namen `p` keine Ahnung.

Hinweis

Der Name einer Parametervariablen darf durchaus mit dem Namen der Argument-Variablen übereinstimmen, was die Semantik nicht verändert. Die Namensräume sind völlig getrennt, und Missverständnisse gibt es nicht, da beide – die aufrufende Methode und die aufgerufene Methode – komplett getrennte lokale Variablen haben.

Wertübergabe und Referenzübergabe per »Call by Value«

Primitive Variablen werden immer per Wert kopiert (engl. *Call by Value*). Das Gleiche gilt für Referenzen. Daher hat auch die folgende statische Methode keine Nebenwirkungen:

Listing 3.16: JavaIsAlwaysCallByValue.java

```
import java.awt.Point;

public class JavaIsAlwaysCallByValue
{
    static void clear( Point p )
    {
        p = new Point();
    }

    public static void main( String[] args )
    {
        Point p = new Point( 10, 20 );
        clear( p );
        System.out.println( p ); // java.awt.Point[x=10,y=20]
    }
}
```

Nach der Zuweisung in der `clear()`-Methode referenziert die Parameter-Variable `p` ein anderes Punkt-Objekt, und der der Methode übergebene Verweis geht damit verloren. Diese Änderung wird nach außen hin natürlich nicht sichtbar, denn das Methoden-`p` ist ja nur ein temporärer alternativer Name für das `main-p`; eine Neuzuweisung an das Methoden-`p` ändert nicht das `main-p`, was bedeutet, dass der Aufrufer von `clear()` kein neues Objekt unter sich hat.



»Call by Reference« gibt es in Java nicht – ein Blick auf C und C++

In C++ gibt es eine weitere Argumentübergabe, die sich *Call by Reference* nennt. Würde eine Methode wie `clear()` mit Referenzsemantik deklariert, würde die Variable `p` ein Synonym darstellen, also einen anderen Namen für eine Variable – in unserem Fall `q`. Damit würde die Zuweisung im Rumpf den Zeiger auf ein neues Objekt legen. Die `swap()`-Funktion ist ein gutes Beispiel für die Nützlichkeit von Call by Reference:

```
void swap( int& a, int& b ) { int tmp = a; a = b; b = tmp; }
```

!

3

»Call by Reference« gibt es in Java nicht – ein Blick auf C und C++ (Forts.)

Zeiger und Referenzen sind in C++ etwas anderes, was Spracheinsteiger leicht irritiert. Denn in C++ und auch in C hätte eine vergleichbare swap()-Funktion auch mit Zeigern implementiert werden können:

```
void swap( int *a, int *b ) { int tmp = *a; *a = *b; *b = tmp; }
```

Die Implementierung gibt in C(++) einen Verweis auf das Argument.

Final deklarierte Referenz-Parameter und das fehlende const

Wir haben gesehen, dass finale Variablen dem Programmierer vorgeben, dass er Variablen nicht beschreiben darf. Das heißt, Zuweisungen sind tabu. Dabei ist es egal, ob die Parametervariable vom primitiven Typ oder vom Referenztyp ist. Bei einer Methodendeklaration der folgenden Art wäre also eine Zuweisung an i und auch an s verboten:

```
public void foo( final int i, final String s )
```

Ist die Parametervariable ein Referenztyp (und nicht final), so würden wir mit einer Zuweisung den Verweis auf das ursprüngliche Objekt verlieren, und das wäre wenig sinnvoll.

```
public void foo( String s )
{
    s = "Keine Feier ohne Geier.";
}
```

Halten wir fest: Ist ein Parameter mit final deklariert, sind keine Zuweisungen möglich. final verbietet aber keine Änderungen an Objekten – und so könnte final im Sinne der Übersetzung »endgültig« verstanden werden. Mit der Referenz des Objekts können wir sehr wohl den Zustand verändern, so wie wir es auch im letzten Beispielprogramm taten.

final erfüllt demnach nicht die Aufgabe, schreibende Objektzugriffe zu verhindern. Eine Methode mit übergebenen Referenzen kann also Objektveränderungen vornehmen, wenn es etwa setXXX()-Methoden oder Variablen gibt, auf die zugegriffen werden kann. Die Dokumentation muss also immer ausdrücklich beschreiben, wann die Methode den Zustand eines Objekts modifiziert.

In C++ gibt es für Parameter den Zusatz const, an dem der Compiler erkennen kann, dass Objektzustände nicht verändert werden sollen. Ein Programm nennt sich *const-korrekt*,

wenn es niemals ein konstantes Objekt verändert. Dieses `const` ist in C++ eine Erweiterung des Objekttyps, die es in Java nicht gibt. Zwar haben die Java-Entwickler das Schlüsselwort `const` reserviert, doch genutzt wird es bisher nicht.

Finale Parameter in der Vererbung *

In der Vererbung von Methoden spielen finale Parametervariablen keine Rolle. Wir können es als zusätzliche Information für die jeweilige Methode betrachten. Eine Unterklasse kann demnach beliebig das `final` hinzufügen oder auch wegnehmen. Alte Bibliotheken lassen sich so leicht weiterverwenden.

3.7.5 Identität von Objekten

Der Vergleichsoperator `==` ist für alle Datentypen so definiert, dass er die vollständige Übereinstimmung zweier Werte testet. Bei primitiven Datentypen ist das einfach einzu-sehen und bei Referenztypen im Prinzip genauso. Der Operator `==` testet bei Referenzen, ob diese übereinstimmen, also auf das gleiche Objekt verweisen. Demnach sagt der Test etwas über die Identität der referenzierten Objekte aus, aber nichts darüber, ob zwei verschiedene Objekte möglicherweise den gleichen Inhalt haben. Der Inhalt der Objekte spielt bei `==` keine Rolle.

zB Beispiel

Zwei Objekte mit drei unterschiedlichen Punktvariablen `p`, `q`, `r` und die Bedeutung von

`==`:

```
Point p = new Point( 10, 10 );
Point q = p;
Point r = new Point( 10, 10 );
System.out.println( p == q ); // true, da p und q dasselbe Objekt referenzieren
System.out.println( p == r ); // false, da p und r zwei verschiedene Punkt-
                           // Objekte referenzieren, die zufällig dieselben
                           // Koordinaten haben
```

Da `p` und `q` auf dasselbe Objekt verweisen, ergibt der Vergleich `true`. `p` und `r` referenzieren unterschiedliche Objekte, die aber zufälligerweise den gleichen Inhalt haben. Doch woher soll der Compiler wissen, wann zwei Punkt-Objekte inhaltlich gleich sind? Weil sich ein Punkt durch die Attribute `x` und `y` auszeichnet? Die Laufzeitumgebung könnte voreilig die Belegung jeder Objektvariablen vergleichen, doch das entspricht nicht

immer einem korrekten Vergleich, so wie wir ihn uns wünschen. Ein Punkt-Objekt könnte etwa zusätzlich die Anzahl der Zugriffe zählen, die jedoch für einen Vergleich, der auf der Lage zweier Punkte basiert, nicht berücksichtigt werden darf.

3.7.6 Gleichheit und die Methode equals()

Die allgemein gültige Lösung besteht darin, die Klasse festlegen zu lassen, wann Objekte gleich sind. Dazu kann jede Klasse eine Methode `equals()` implementieren, die Exemplare dieser Klasse mit beliebigen anderen Objekten vergleichen kann. Die Klassen entscheiden immer nach Anwendungsfall, welche Attribute sie für einen Gleichheitstest heranziehen, und `equals()` liefert `true`, wenn die gewünschten Zustände (Objektvariablen) übereinstimmen.

Beispiel

zB

Zwei inhaltlich gleiche Punkt-Objekte, verglichen mit `==` und `equals()`:

```
Point p = new Point( 10, 10 );
Point q = new Point( 10, 10 );
System.out.println( p == q );      // false
System.out.println( p.equals(q) ); // true. Da symmetrisch auch q.equals(p)
```

Nur `equals()` testet in diesem Fall die inhaltliche Gleichheit.

Bei den unterschiedlichen Bedeutungen müssen wir demnach die Begriffe *Identität* und *Gleichheit* von Objekten sorgfältig unterscheiden. Daher noch einmal eine Zusammenfassung:

	Getestet mit	Implementierung
Identität der Referenzen	<code>==</code>	nichts zu tun
Gleichheit der Zustände	<code>equals()</code>	abhängig von der Klasse

Tabelle 3.4: Identität und Gleichheit von Objekten

Es gibt immer ein `equals()` *

Die Klasse `Point` deklariert `equals()`, wie die API-Dokumentation zeigt. Werfen wir einen Blick auf die Implementierung, um eine Vorstellung von der Arbeitsweise zu bekommen:

Listing 3.17: java.awt/Point.java, Ausschnitt

```

public class Point ...
{
    public int x;
    public int y;

    public boolean equals( Object obj )
    {
        if ( obj instanceof Point ) {
            Point pt = (Point) obj;
            return (x == pt.x) && (y == pt.y);      // (*)
        }
        return super.equals( obj );
    }
    ...
}

```

Obwohl bei diesem Beispiel für uns einiges neu ist, erkennen wir den Vergleich in der Zeile (*). Hier vergleicht das `Point`-Objekt seine eigenen Attribute mit den Attributen des Objekts, das als Argument an `equals()` übergeben wurde.

Es gibt immer ein `equals()`: die Oberklasse `Object` und ihr `equals()` *

Glücklicherweise müssen wir als Programmierer nicht lange darüber nachdenken, ob eine Klasse eine `equals()`-Methode anbieten soll oder nicht. Jede Klasse besitzt sie, da die universelle Oberklasse `Object` sie vererbt. Wir greifen hier auf Kapitel 5, »Eigene Klassen schreiben«, vor; der Abschnitt kann aber übersprungen werden. Wenn eine Klasse also keine eigene `equals()`-Methode angibt, dann erbt sie eine Implementierung aus der Klasse `Object`. Die sieht wie folgt aus:

Listing 3.18: java/lang/Object.java, Ausschnitt

```

public class Object
{
    public boolean equals( Object obj )
    {
        return ( this == obj );
    }
}

```

```
}
```

```
...
```

```
}
```

3

Wir erkennen, dass hier die Gleichheit auf die Gleichheit der Referenzen abgebildet wird. Ein inhaltlicher Vergleich findet nicht statt. Das ist das einzige, was die vorgegebene Implementierung machen kann, denn sind die Referenzen identisch, sind die Objekte logischerweise auch gleich. Nur über Zustände »weiß« die Basisklasse `Object` nichts.

3.8 Arrays

Ein *Array* (auch *Feld* oder *Reihung* genannt) ist ein spezieller Datentyp, der mehrere Werte zu einer Einheit zusammenfasst. Er ist mit einem Setzkasten vergleichbar, in dem die Plätze durchnummieriert sind. Angesprochen werden die Elemente über einen ganzzahligen Index. Jeder Platz (etwa für Schlämpfe) nimmt immer Werte des gleichen Typs auf (nur Schlämpfe und keine Pokémons). Normalerweise liegen die Plätze eines Arrays (seine Elemente) im Speicher hintereinander, doch ist dies ein für Programmierer nicht sichtbares Implementierungsdetail der virtuellen Maschine.

Jedes Array beinhaltet Werte nur eines bestimmten Datentyps bzw. Grundtyps. Dies können sein:

- elementare Datentypen wie `int`, `byte`, `long` und so weiter
- Referenztypen
- Referenztypen anderer Arrays, um mehrdimensionale Arrays zu realisieren

3.8.1 Grundbestandteile

Für das Arbeiten mit Arrays müssen wir drei neue Dinge kennenlernen:

1. das Deklarieren von Array-Variablen
2. das Initialisieren von Array-Variablen, Platzbeschaffung
3. den Zugriff auf Arrays. Das umfasst den Lesenden Zugriff ebenso wie den schreibenden.

zB Beispiel

1. Deklariere eine Variable `randoms`, die ein Array referenziert:

```
double[] randoms;
```

2. Initialisiere die Variable mit einem Array-Objekt der Größe 10:

```
randoms = new double[ 10 ];
```

3. Belege das erste Element mit einer Zufallszahl und das zweite Element mit dem Doppelten des ersten Elements:

```
randoms[ 0 ] = Math.random();
randoms[ 1 ] = randoms[ 0 ] * 2;
```

Die drei Punkte schauen wir uns nun detaillierter an.

3.8.2 Deklaration von Arrays

Eine Array-Variablen-deklaration ähnelt einer gewöhnlichen Deklaration, nur dass nach dem Datentyp die Zeichen »[« und »] « gesetzt werden.

zB Beispiel

Deklariere zwei Feld-Variablen:

```
int[] primes;
Point[] points;
```

Eine Variable wie `primes` hat jetzt den Typ »ist Feld« und »speichert int-Elemente«, also eigentlich zwei Typen.

 **Hinweis**

Die eckigen Klammern lassen sich bei der Deklaration einer Array-Variablen auch hinter den Namen setzen, doch ganz ohne Unterschied ist die Deklaration nicht. Das zeigt sich spätestens dann, wenn mehr als eine Variable deklariert wird:

```
int []primes,
matrix[], threeDimMatrix[][][];
```

**Hinweis (Forts.)**

Das entspricht dieser Deklaration :

```
int primes[], matrix[][], threeDimMatrix[][][];
```

Damit Irrtümer dieser Art ausgeschlossen werden, sollten Sie in jeder Zeile nur eine Deklaration eines Typs schreiben. Nach reiner Java-Lehre gehören die Klammern jedenfalls hinter den Typbezeichner, so hat es der Java-Schöpfer James Gosling gewollt.

3.8.3 Arrays mit Inhalt

Die bisherigen Deklarationen von Array-Variablen erzeugen noch lange kein Array-Objekt, das die einzelnen Array-Elemente aufnehmen kann. Wenn allerdings die Einträge direkt mit Werten belegt werden sollen, gibt es in Java eine Abkürzung, die ein Array-Objekt anlegt und zugleich mit Werten belegt.

Beispiel

zB

Wertebefüllung eines Felds:

```
int[] primes = { 2, 3, 5, 7, 7 + 4 };
String[] strings = {
    "Haus", "Maus",
    "dog".toUpperCase(), // DOG
    new java.awt.Point().toString(),
};
```

In diesem Fall wird ein Feld mit passender Größe angelegt, und die Elemente, die in der Aufzählung genannt sind, werden in das Feld kopiert. Innerhalb der Aufzählung kann abschließend ein Komma stehen, wie die Aufzählung bei strings demonstriert.

3.8.4 Die Länge eines Arrays über das Attribut length auslesen

Die Anzahl der Elemente, die ein Array aufnehmen kann, wird *Größe* beziehungsweise *Länge* genannt und ist für jedes Array-Objekt in der frei zugänglichen Objektvariablen `length` gespeichert. `length` ist eine `public final int`-Variable, deren Wert entweder positiv oder null ist. Die Größe lässt sich später nicht mehr ändern.

zB Beispiel

Ein Feld und die Ausgabe der Länge:

```
int[] primes = { 2, 3, 5, 7, 7 + 4 };
System.out.println( primes.length );           // 5
```

Feldlängen sind final

Das Attribut `length` eines Felds ist nicht nur öffentlich (`public`) und vom Typ `int`, sondern natürlich auch `final`. Schreibzugriffe sind nicht gestattet. (Was sollten sie bewirken? Eine dynamische Vergrößerung des Felds?) Ein Schreibzugriff führt zu einem Übersetzungsfehler.

3.8.5 Zugriff auf die Elemente über den Index

Der Zugriff auf die Elemente eines Felds erfolgt mithilfe der eckigen Klammern `[]`, die hinter die Referenz an das Array-Objekt gesetzt werden. In Java beginnt ein Array beim Index 0 (und nicht bei einer frei wählbaren Untergrenze wie in Pascal). Da die Elemente eines Arrays ab 0 nummeriert werden, ist der letzte gültige Index um 1 kleiner als die Länge des Felds. Bei einem Array `a` der Länge `n` ist der gültige Bereich somit `a[0]` bis `a[n - 1]`.

zB Beispiel

Greife auf das erste und letzte Zeichen aus dem Feld zu:

```
char[] name = { 'C', 'h', 'r', 'i', 's' };
char first = name[ 0 ];                      // C
char last = name[ name.length - 1 ];          // s
```

Da der Zugriff auf die Variablen über einen Index erfolgt, werden diese Variablen auch *indexierte Variablen* genannt.

zB Beispiel

Laufe das Feld der ersten Primzahlen komplett ab:

```
int[] primes = { 2, 3, 5, 7, 11 };
for ( int i = 0; i < primes.length; i++ )    // Index: 0 <= i < 5 = primes.length
    System.out.println( primes[ i ] );
```

Anstatt ein Feld einfach nur so abzulaufen und die Werte auszugeben, soll unser nächstes Programm den Mittelwert einer Zahlenfolge berechnen und ausgeben:

Listing 3.19: PrintTheAverage.java

```
public class PrintTheAverage
{
    public static void main( String[] args )
    {
        double[] numbers = { 1.9, 7.8, 2.4, 9.3 };

        double sum = 0;

        for ( int i = 0; i < numbers.length; i++ )
            sum += numbers[ i ];

        double avg = sum / numbers.length;

        System.out.println( avg ); // 5.35
    }
}
```

Das Feld muss mindestens ein Element besitzen, sonst gibt es bei der Division durch 0 eine Ausnahme.

Über den Typ des Index *

Innerhalb der eckigen Klammern steht ein positiver Ganzzahl-Ausdruck vom Typ `int`, der sich zur Laufzeit berechnen lassen muss. `long`-Werte, `boolean`, Gleitkommazahlen oder Referenzen sind nicht möglich; durch `int` verbleiben aber mehr als zwei Milliarden Elemente. Bei Gleitkommazahlen bliebe die Frage nach der Zugriffstechnik. Hier müssten wir den Wert auf ein Intervall herunterrechnen.

Strings sind keine Arrays *

Ein Array von `char`-Zeichen hat einen ganz anderen Typ als ein `String`-Objekt. Während bei Feldern eckige Klammern erlaubt sind, bietet die `String`-Klasse (bisher) keinen Zugriff auf Zeichen über `[]`. Die Klasse `String` bietet jedoch einen Konstruktor an, sodass aus einem Feld mit Zeichen ein `String`-Objekt erzeugt werden kann. Alle Zeichen des Felds werden kopiert, sodass anschließend Feld und `String` keine Verbindung mehr

besitzen. Dies bedeutet: Wenn sich das Feld ändert, ändert sich der String nicht automatisch mit. Das kann er auch nicht, da Strings unveränderlich sind.

3.8.6 Array-Objekte mit new erzeugen

Ein Array muss mit dem `new`-Operator unter Angabe einer festen Größe erzeugt werden. Das Anlegen der Variablen allein erzeugt noch kein Feld mit einer bestimmten Länge. In Java ist das Anlegen des Felds genauso dynamisch wie die Objekterzeugung. Dies drückt auch der `new`-Operator aus.¹⁷ Die Länge des Felds wird in eckigen Klammern angegeben. Hier kann ein beliebiger Integer-Wert stehen, auch eine Variable. Selbst 0 ist möglich.

zB

Beispiel

Erzeuge ein Feld für zehn Elemente:

```
int[] values;
values = new int[ 10 ];
```

Die Feld-Deklaration ist auch zusammen mit der Initialisierung möglich:

```
double[] values = new double[ 10 ];
```

Die Felder mit den primitiven Werten sind mit 0, 0.0 oder false und bei Verweisen mit null initialisiert.

Dass Arrays Objekte sind, zeigen einige Indizien:

- Eine spezielle Form des `new`-Operators erzeugt ein Exemplar der Array-Klasse; `new` erinnert uns immer daran, dass ein Objekt zur Laufzeit aufgebaut wird.
- Ein Array-Objekt kennt das Attribut `length`, und auf dem Array-Objekt sind Methoden – wie `clone()` und alles, was `java.lang.Object` hat – definiert.
- Die Operatoren `==` und `!=` haben ihre Objekt-Bedeutung: Sie vergleichen lediglich, ob zwei Variablen auf das gleiche Array-Objekt verweisen, aber auf keinen Fall die Inhalte der Arrays (das kann aber `Arrays.equals()`).

Der Zugriff auf die Array-Elemente über die eckigen Klammern `[]` lässt sich als versteckter Aufruf über geheime Methoden wie `array.get(index)` verstehen. Der `[]`-Operator wird bei anderen Objekten nicht angeboten.

¹⁷ Programmiersprachen wie C(++) bieten bei der Felderzeugung Abkürzungen wie `int array[100]`. Das führt in Java zu einem Compilerfehler.

Der Index vom Typ char ist auch ein int *

Der Index eines Felds muss von einem Typ sein, der ohne Verlust in `int` konvertierbar ist. Dazu gehören `byte`, `short` und `char`.

Beispiel

zB

Günstig ist ein Index vom Typ `char`, zum Beispiel als Laufvariable, wenn Felder von Zeichen generiert werden:

```
char[] alphabet = new char[ 'z' - 'a' + 1 ]; // 'a' entspricht 97 und 'z' 122
for ( char c = 'a'; c <= 'z'; c++ )
    alphabet[ c - 'a' ] = c;           // alphabet[0]='a', alphabet[1]='b', usw.
```

Genau genommen haben wir es auch hier mit Indexwerten vom Typ `int` zu tun, weil mit den `char`-Werten vorher noch gerechnet wird.

3.8.7 Typische Feldfehler

Beim Zugriff auf ein Array-Element können Fehler auftreten. Zunächst einmal kann das Array-Objekt fehlen, sodass die Referenzierung fehlschlägt.

Beispiel

zB

Der Compiler bemerkt den folgenden Fehler nicht, und die Strafe ist eine `NullPointerException` zur Laufzeit.¹⁸

```
int[] array = null;
array[ 1 ] = 1;      // ☹ NullPointerException
```

Weitere Fehler können im Index begründet sein. Ist der Index negativ¹⁹ oder zu groß, dann gibt es eine `IndexOutOfBoundsException`. Jeder Zugriff auf das Feld wird zur Laufzeit getestet, auch wenn der Compiler durchaus einige Fehler finden könnte.

¹⁸ Obwohl er sich bei nicht initialisierten lokalen Variablen auch beschwert.

¹⁹ Ganz anders verhalten sich da Python oder Perl. Dort wird ein negativer Index dazu verwendet, ein Feldelement relativ zum letzten Array-Eintrag anzusprechen. Und auch bei C ist ein negativer Index durchaus möglich und praktisch.

zB Beispiel

Bei folgenden Zugriffen könnte der Compiler theoretisch Alarm schlagen, was aber zu mindest der Standard-Compiler nicht tut. Der Grund ist, dass der Zugriff auf die Elemente auch mit einem ungültigen Index syntaktisch völlig in Ordnung ist.

```
int[] array = new int[ 100 ];
array[ -10 ] = 1;    // ☹ Fehler zur Laufzeit, nicht zur Compilierzeit
array[ 100 ] = 1;    // ☹ Fehler zur Laufzeit, nicht zur Compilierzeit
```

Wird die `IndexOutOfBoundsException` nicht abgefangen, bricht das Laufzeitsystem das Programm mit einer Fehlermeldung ab. Dass die Feldgrenzen überprüft werden, ist Teil von Javas Sicherheitskonzept und lässt sich nicht abstellen. Es ist aber heute kein großes Performance-Problem mehr, da die Laufzeitumgebung nicht jeden Index prüfen muss, um sicherzustellen, dass ein Block mit Feldzugriff korrekt ist.

Spielerei: Der Index und das Inkrement *

Wir haben beim Inkrement schon ein Phänomen wie `i = i++` betrachtet. Ebenso ist auch die Anweisung bei einem Feldzugriff zu behandeln:

```
array[ i ] = i++;
```

Bei der Position `array[i]` wird `i` gesichert und anschließend die Zuweisung vorgenommen. Wenn wir eine Schleife darum konstruieren, erweitern wir dies zu einer Initialisierung:

```
int[] array = new int[ 4 ];
int i = 0;
while ( i < array.length )
    array[ i ] = i++;
```

Die Ausgabe ergibt 0, 1, 2 und 3. Von der Anwendung ist wegen mangelnder Übersicht abzuraten.

3.8.8 Feld-Objekte als Parametertyp

Verweise auf Felder lassen sich bei Methoden genauso übergeben wie Verweise auf ganz normale Objekte. In der Deklaration heißt es dann zum Beispiel `foo(int[] val)` statt `foo(String val)`.

Wir hatten vorher schon den Mittelwert einer Zahlenreihe ermittelt. Die Logik dafür ist perfekt in eine Methode ausgelagert:

Listing 3.20: Avg1.java

```
public class Avg1
{
    static double avg( double[] array )
    {
        double sum = 0;

        for ( int i = 0; i < array.length; i++ )
            sum += array[ i ];

        return sum / array.length;
    }

    public static void main( String[] args )
    {
        double[] numbers = new double[]{ 2, 3, 4 };
        System.out.println( avg( numbers ) );           // 3.0
    }
}
```

3.8.9 Vorinitialisierte Arrays

Wenn wir in Java ein Array-Objekt erzeugen und gleich mit Werten initialisieren wollen, dann schreiben wir etwa:

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

Sollen die Feldinhalte erst nach der Variablen Deklaration initialisiert oder soll das Feld auch ohne Variable genutzt werden, so erlaubt Java dies nicht, und ein Versuch wie der folgende schlägt mit der Compilermeldung »Array constants can only be used in initializers« fehl:

```
primes = { 2, 5, 7, 11, 13 };           // ☹ Compilerfehler
avg( { 1.23, 4.94, 9.33, 3.91, 6.34 } ); // ☹ Compilerfehler
```

Zur Lösung gibt es zwei Ansätze. Der erste ist die Einführung einer neuen Variablen, hier `tmpprimes`:

```
int[] primes;
int[] tmpprimes = { 2, 5, 7, 11, 13 };
primes = tmpprimes;
```

Als zweiten Ansatz gibt es eine Variante des `new`-Operators, der durch ein Paar eckiger Klammern erweitert wird. Es folgen in geschweiften Klammern die Initialwerte des Arrays. Die Größe des Arrays entspricht genau der Anzahl der Werte. Für die oberen Beispiele ergibt sich folgende Schreibweise:

```
int[] primes;
primes = new int[]{ 2, 5, 7, 11, 13 };
```

Diese Notation ist auch bei Methodenaufrufen sehr praktisch, wenn Felder übergeben werden:

```
avg( new double[]{ 1.23, 4.94, 9.33, 3.91, 6.34 } );
```

Da hier ein initialisiertes Feld mit Werten gleich an die Methode übergeben und keine zusätzliche Variable benutzt wird, heißt diese Art der Arrays »anonyme Arrays«. Eigentlich gibt es auch sonst anonyme Arrays, wie `new int[2000].length` zeigt, doch wird in diesem Fall das Feld nicht mit Werten initialisiert.

3.8.10 Die erweiterte for-Schleife

for-Schleifen laufen oft Felder oder Datenstrukturen ab. Bei der Berechnung des Mittelwertes konnten wir das ablesen:

```
double sum = 0;
for ( int i = 0; i < array.length; i++ )
    sum += array[ i ];
double avg = sum / array.length;
```

Die Schleifenvariable `i` hat lediglich als Index ihre Berechtigung; nur damit lässt sich das Element an einer bestimmten Stelle im Feld ansprechen.

Weil das komplette Durchlaufen von Feldern häufig ist, wurde in Java 5 eine Abkürzung für solche Iterationen in die Sprache eingeführt:

```
for ( Typ Bezeichner : Feld )
    ...

```

Die erweiterte Form der `for`-Schleife löst sich vom Index und erfragt jedes Element des Felds. Das können Sie sich als Durchlauf einer Menge vorstellen, denn der Doppelpunkt liest sich als »in«. Rechts vom Doppelpunkt steht immer ein Feld oder, wie wir später sehen werden, etwas vom Typ `Iterable`, wie eine Datenstruktur. Links wird eine neue lokale Variable deklariert, die später beim Ablauf jedes Element der Sammlung annehmen wird.

Die Berechnung des Durchschnitts lässt sich nun umschreiben. Die statische Methode `avg()` soll mit dem erweiterten `for` über die Schleife laufen, anstatt den Index selbst hochzuzählen. Eine Ausnahme zeigt an, ob der Feldverweis `null` ist oder das Feld keine Elemente enthält:

Listing 3.21: Avg2.java, avg()

```
static double avg( double[] array )
{
    if ( array == null || array.length == 0 )
        throw new IllegalArgumentException( "Array null oder leer" );

    double sum = 0;

    for ( double n : array )
        sum += n;

    return sum / array.length;
}
```

Zu lesen ist die `for`-Zeile demzufolge als »Für jedes Element `n` vom Typ `double` in `array` tue ...«. Eine Variable für den Schleifenindex ist nicht mehr nötig.

Beispiel

zB

Rechts vom Doppelpunkt lässt sich auf die Schnelle ein Feld aufbauen, über welches das erweiterte `for` dann laufen kann.

```
for ( int prime : new int[]{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 } )
    System.out.println( prime );
```

Umsetzung und Einschränkung

Intern setzt der Compiler diese erweiterte `for`-Schleife ganz klassisch um, sodass der Bytecode unter beiden Varianten gleich ist. Nachteile dieser Variante sind jedoch:

- Das erweiterte `for` läuft immer das ganze Feld ab. Ein Anfangs- und ein Ende-Index können nicht ausdrücklich gesetzt werden.
- Die Ordnung ist immer von vorn nach hinten.
- Der Index ist nicht sichtbar.
- Die Schleife liefert ein Element, kann aber nicht in das Feld schreiben.

Abbrechen lässt sich die Schleife mit einem `break`. Bestehen andere Anforderungen, kann weiterhin nur eine klassische `for`-Schleife helfen.

3.8.11 Arrays mit nicht-primitiven Elementen

Der Datentyp der Array-Elemente muss nicht zwingend ein primitiver sein. Auch ein Array von Objektreferenzen kann deklariert werden. Dieses Array besteht dann nur aus Referenzen auf die eigentlichen Objekte, die in dem Array abgelegt werden sollen. Die Größe des Arrays im Speicher errechnet sich demnach aus der Länge des Felds, multipliziert mit dem Speicherbedarf einer Referenzvariablen. Nur das Array-Objekt selbst wird angelegt, nicht aber die Objekte, die das Array aufnehmen soll. Dies lässt sich einfach damit begründen, dass der Compiler auch gar nicht wüsste, welchen Konstruktor er aufrufen sollte.

Felder mit Strings durchsuchen

In unserem ersten Beispiel soll ein nicht-primitives Feld `Strings` referenzieren und später schauen, ob eine Benutzereingabe im Feld ist. String-Vergleiche lassen sich mit `equals()` realisieren:

Listing 3.22: UserInputInStringArray.java

```
import java.util.Scanner;

public class UserInputInStringArray
{
    public static void main( String[] args )
    {
        String[] validInputs = { "Banane", "Apfel", "Kirsche" };
    }
}
```

```

userInputLoop:
while ( true )
{
    String input = new Scanner( System.in ).nextLine();

    for ( String s : validInputs )
        if ( s.equals( input ) )
            break userInputLoop;

}

System.out.println( "Gültiges Früchtchen eingegeben" );
}
}

```

Zur Initialisierung des Feldes nutzt das Programm eine kompakte Variante, die drei Dinge vereint: den Aufbau eines Feld-Objektes (mit Platz für drei Referenzen), die Initialisierung des Feld-Objektes mit den drei Objektreferenzen und schlussendlich die Initialisierung der Variablen `validInputs` mit dem neuen Feld – alles in einer Anweisung.

Für die Suche kommt das erweiterte `for` zum Einsatz, das in einer Endlosschleife eingebettet ist. Erst wenn es einen Fund gibt, verlässt das `break` die Endlosschleife. Wir müssen hier zu Sprungmarken greifen, denn ein `break` ohne Sprungmarke würde die erweiterte `for`-Schleife beenden, was wir aber nicht möchten.

Zufällige Spielerpositionen erzeugen

Im zweiten Beispiel sollen fünf zufällig initialisierte Punkte in einem Feld abgelegt werden. Die Punkte sollten Spieler repräsentieren.

Zunächst benötigen wir ein Feld:

```
Point[] players = new Point[ 5 ];
```

Die Deklaration schafft Platz für fünf Verweise auf Punkt-Objekte, aber kein einziges `Point`-Objekt ist angelegt. Standardmäßig werden die Array-Elemente mit der `null`-Referenz initialisiert, sodass `System.out.println(players[0])` die Ausgabe »null« auf den Bildschirm bringen würde. Bei `null` wollen wir es nicht belassen, daher müssen die einzelnen Feldplätze etwa mit `players[0] = new Point()` initialisiert werden.

Zufallszahlen erzeugt die mathematische Methode `Math.random()`. Da die statische Methode jedoch Fließkommazahlen zwischen 0 (inklusiv) und 1 (exklusiv) liefert, werden die Zahlen zunächst durch Multiplikation frisiert und dann abgeschnitten.

Im letzten Schritt geben wir ein Raster auf dem Bildschirm aus, in dem zwei ineinander verschachtelte Schleifen alle x/y-Koordinaten des gewählten Bereichs ablaufen und dann ein »&« setzen, wenn der Punkt einen Spieler trifft.

Das Programm als Ganzes:

Listing 3.23: FivePlayers.java

```
import java.awt.Point;
import java.util.Arrays;

public class FivePlayers
{
    public static void main( String[] args )
    {
        Point[] players = new Point[ 5 ];

        for ( int i = 0; i < players.length; i++ )
            players[ i ] = new Point( (int)(Math.random() * 40),
                                      (int)(Math.random() * 10) );

        for ( int y = 0; y < 10; y++ )
        {
            for ( int x = 0; x < 40; x++ )
                if ( Arrays.asList( players ).contains( new Point(x,y) ) )
                    System.out.print( "&" );
                else
                    System.out.print( "." );
            System.out.println();
        }
    }
}
```

Der Ausdruck `Arrays.asList(players).contains(new Point(x,y))` testet, ob irgendein Punkt im Feld `players` gleich dem Punkt mit den x/y-Koordinaten ist.

Die Ausgabe erzeugt zum Beispiel Folgendes:

```
.....  
.....&.....  
8.....  
.....  
.....  
.....&.....  
.....  
....&.....&.....  
.....  
.....
```

Während die erweiterte `for`-Schleife gut das Feld ablaufen kann, funktioniert das zur Initialisierung nicht, denn das erweiterte `for` ist nur zum Lesen gut. Elementinitialisierungen funktionieren bei Feldern nur mit `players[i]=...`, und dazu ist eben eine klassische `for`-Schleife mit dem Index nötig.

3.8.12 Mehrdimensionale Arrays *

Java realisiert mehrdimensionale Arrays durch Arrays von Arrays. Sie können etwa für die Darstellung von mathematischen Matrizen oder Rasterbildern Verwendung finden. Dieser Abschnitt lehrt, wie Objekte für mehrdimensionale Felder initialisiert, aufgebaut und abgegrast werden.

Feld-Objekte mit new aufbauen

Die folgende Zeile deklariert ein zweidimensionales Feld mit Platz für 32 Zellen, die in vier Zeilen und acht Spalten angeordnet sind:

```
int[][] A = new int[ 4 ][ 8 ];
```

Obwohl mehrdimensionale Arrays im Prinzip Arrays mit Arrays als Elementen sind, lassen sie sich leicht deklarieren. Zwei alternative Deklarationen (die Position der eckigen Klammern ist verschoben) sind:

```
int A[][] = new int[ 4 ][ 8 ];  
int[] A[] = new int[ 4 ][ 8 ];
```

Anlegen und Initialisieren in einem Schritt

Ebenso wie bei eindimensionalen Feldern lassen sich mehrdimensionale Felder gleich beim Anlegen initialisieren:

```
int[][] A3x2 = { {1, 2}, {2, 3}, {3, 4} };
int[][] B     = { {1, 2}, {2, 3, 4}, {5} };
```

Der zweite Fall lässt erkennen, dass das Feld nicht unbedingt rechteckig sein muss. Dazu gleich mehr.

Zugriff auf Elemente

Einzelne Elemente spricht der Ausdruck $A[i][j]$ an.²⁰ Der Zugriff erfolgt mit so vielen Klammerpaaren, wie die Dimension des Arrays angibt.

zB

Beispiel

Der Aufbau von zweidimensionalen Feldern (und der Zugriff auf sie) ist mit einer Matrix beziehungsweise Tabelle vergleichbar. Dann lässt sich der Eintrag im Feld $a[x][y]$ in folgender Tabelle ablesen:

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$	$a[0][4]$	$a[0][5]$...
$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$	$a[1][4]$	$a[1][5]$	
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$	$a[2][4]$	$a[2][5]$	
...						

length() bei mehrdimensionalen Feldern

Nehmen wir eine Buchstabendefinition wie die folgende:

```
char[][] letter = { { ' ', '#', ' ' },
                    { '#', ' ', '#' },
                    { '#', ' ', '#' },
                    { '#', ' ', '#' },
                    { ' ', '#', ' ' } };
```

²⁰ Die in Pascal übliche Notation $A[i,j]$ wird in Java nicht unterstützt. Die Notation wäre im Prinzip möglich, da Java im Gegensatz zu C(++) den Komma-Operator nur in for-Schleifen zulässt.

Dann können wir `length()` auf zwei verschiedene Weisen anwenden:

- `letter.length()` ergibt 5, denn es gibt 5 Zeilen.
- `letter[0].length()` ergibt 3 – genauso wie `letter[1].length()` usw. –, weil jedes Unterfeld die Größe 3 hat.

Zweidimensionale Felder mit ineinandergeschachtelten Schleifen ablaufen

Um den Buchstaben unseres Beispiels auf dem Bildschirm auszugeben, nutzen wir zwei ineinandergeschachtelte Schleifen:

```
for ( int line = 0; line < letter.length; line++ )  
{  
    for ( int colum = 0; colum < letter[line].length; colum++ )  
        System.out.print( letter[line][colum] );  
    System.out.println();  
}
```

Fassen wir das Wissen zu einem Programm zusammen, das vom Benutzer eine Zahl erfragt und diese Zahl in Binärdarstellung ausgibt. Wir drehen die Buchstaben um 90 Grad im Uhrzeigersinn, damit wir uns nicht damit beschäftigen müssen, die Buchstaben horizontal nebeneinander zu legen.

Listing 3.24: BinaryBanner.java

```
import java.util.Scanner;  
  
public class BinaryBanner  
{  
    static void printLetter( char[][] letter )  
    {  
        for ( int colum = 0; colum < letter[0].length; colum++ )  
        {  
            for ( int line = letter.length - 1; line >= 0; line-- )  
                System.out.print( letter[line][colum] );  
            System.out.println();  
        }  
        System.out.println();  
    }  
}
```

```

static void printZero()
{
    char[][] zero = { { ' ', '#', ' ' },
                      { '#', ' ', '#' },
                      { '#', ' ', '#' },
                      { '#', ' ', '#' },
                      { ' ', '#', ' ' } };
    printLetter( zero );
}

static void printOne()
{
    char[][] one = { { ' ', '#' },
                     { '#', '#' },
                     { ' ', '#' },
                     { ' ', '#' },
                     { ' ', '#' } };
    printLetter( one );
}

public static void main( String[] args )
{
    int input = new Scanner( System.in ).nextInt();
    String bin = Integer.toBinaryString( input );
    System.out.printf( "Banner für %s (binär %s):%n", input, bin );
    for ( int i = 0; i < bin.length(); i++ )
        switch ( bin.charAt( i ) )
        {
            case '0': printZero(); break;
            case '1': printOne(); break;
        }
}

```

Die Methode `printLetter()` bekommt als Argument das `char[][]`-Feld und läuft es anders ab als im ersten Fall, um die Rotation zu realisieren. Mit der Eingabe 2 gibt es folgende Ausgabe:

Banner für 2 (binär 10):

```
#  
#####  
  
###  
# #  
###
```

Nützlicher Kommaoperator

Nur in `for`-Schleifen erlaubt Java die Nutzung des Kommaoperators. Für mehrdimensionale Felder ist das eine nützliches Spracheigenschaft, wie das folgende Programm zeigt: Es testet, ob ein zweidimensionales Feld ein gültiges Sudoku-Puzzle ist (die Zahlen müssen im Bereich 1 bis 9 liegen; das überprüft das Programm allerdings nicht).

Listing 3.25: SudokuValidddator.java

```
public class SudokuValidddator  
{  
    public static boolean isValidSudoku( int[][] puzzle )  
    {  
        for ( int sum = 0, prod = 1, row = 0; row < 9; sum = 0, prod = 1, row++ )  
        {  
            for ( int col = 0; col < 9;  
                  sum += puzzle[row][col], prod *= puzzle[row][col], col++ ) {}  
            if ( sum != 45 || prod != 362880 )  
                return false;  
        }  
  
        for ( int sum = 0, prod = 1, col = 0; col < 9; sum = 0, prod = 1, col++ )  
        {  
            for ( int row = 0; row < 9;  
                  sum += puzzle[row][col], prod *= puzzle[row][col], row++ ) {}  
            if ( sum != 45 || prod != 362880 )  
                return false;  
        }  
        return true;  
    }  
}
```

```

public static void main( String[] args )
{
    int[][] puzzle = { {7,3,1,8,5,2,6,9,4},
                       {2,5,4,9,7,6,8,3,1},
                       {9,6,8,3,4,1,5,2,7},
                       {8,4,5,1,2,7,3,6,9},
                       {6,1,2,4,9,3,7,8,5},
                       {3,9,7,5,6,8,4,1,2},
                       {1,7,6,2,8,4,9,5,3},
                       {4,2,9,6,3,5,1,7,8},
                       {5,8,3,7,1,9,2,4,6}};

    System.out.println( isValidSudoku(puzzle) ); // true

    puzzle[0][0] = 8;
    System.out.println( isValidSudoku(puzzle) ); // false
}
}

```

Die Idee bei der Lösung ist, die Summe und das Produkt von jeder Zeile und Spalte zu berechnen und zu testen, ob sie 45 und 362880 ist.

3.8.13 Nichtrechteckige Arrays *

Da in Java mehrdimensionale Arrays als Arrays von Arrays implementiert sind, müssen diese nicht zwingend rechteckig sein. Jede Zeile im Feld kann eine eigene Größe haben.

zB Beispiel

Ein dreieckiges Array mit Zeilen der Länge 1, 2 und 3:

```

int[][] a = new int[ 3 ][];
for ( int i = 0; i < 3; i++ )
    a[ i ] = new int[ i + 1 ];

```

Initialisierung der Unterfelder

Wenn wir ein mehrdimensionales Feld deklarieren, erzeugen versteckte Schleifen automatisch die inneren Felder. Bei

```
int[][] a = new int[ 3 ][ 4 ];
```

erzeugt die Laufzeitumgebung die passenden Unterfelder automatisch. Dies ist bei

```
int[][] a = new int[ 3 ][];
```

nicht so. Hier müssen wir selbst die Unterfelder initialisieren, bevor wir auf die Elemente zugreifen:

```
for ( int i = 0; i < a.length; i++ )
    a[ i ] = new int[ 4 ];
```

PS: `int[][] m = new int[][][4];` funktioniert natürlich nicht!

Beispiel

zB

Es gibt verschiedene Möglichkeiten, ein mehrdimensionales Array zu initialisieren:

```
int[][] A3x2 = { {1,2}, {2,3}, {3,4} };
```

beziehungsweise

```
int[][] A3x2 = new int[][]{ {1,2}, {2,3}, {3,4} };
```

beziehungsweise

```
int[][] A3x2 = new int[][]{ new int[]{1,2}, new int[]{2,3}, new int[]{3,4} };
```

Das pascalsche Dreieck

Das folgende Beispiel zeigt eine weitere Anwendung von nichtrechteckigen Arrays, in der das pascalsche Dreieck nachgebildet wird. Das Dreieck ist so aufgebaut, dass die Elemente unter einer Zahl genau die Summe der beiden direkt darüberstehenden Zahlen bilden. Die Ränder sind mit Einsen belegt.

Listing 3.26: Das pascalsche Dreieck

```

      1
     1   1
    1   2   1
   1   3   3   1
  1   4   6   4   1
 1   5   10  10   5   1
1   6   15  20  15   6   1
```

In der Implementierung wird zu jeder Ebene dynamisch ein Feld mit der passenden Länge angefordert. Die Ausgabe tätigt `printf()` mit einigen Tricks mit dem Formatspezifizierer, da wir auf diese Weise ein führendes Leerzeichen bekommen:

Listing 3.27: PascalsTriangle.java

```
class PascalsTriangle
{
    public static void main( String[] args )
    {
        int[][] triangle = new int[7][];
        for ( int row = 0; row < triangle.length; row++ )
        {
            System.out.printf( "%." + (14 - row*2) +"s", " " );
            triangle[row] = new int[row + 1];
            for ( int col = 0; col <= row; col++ )
            {
                if ( (col == 0) || (col == row) )
                    triangle[row][col] = 1;
                else
                    triangle[row][col] = triangle[row - 1][col - 1] +
                        triangle[row - 1][col];
                System.out.printf( "%3d ", triangle[row][col] );
            }
            System.out.println();
        }
    }
}
```

Die Anweisung `System.out.printf("%." + (14 - row*2) +"s", " ")` produziert Einrückungen. Ohne die Konkatenation liest es sich einfacher:

- `System.out.printf("%.14s", " ")` führt zu "
- `System.out.printf("%.12s", " ")` führt zu "
- `System.out.printf("%.10s", " ")` führt zu "
- usw.

Andere Anwendungen

Mit zweidimensionalen Feldern ist die Verwaltung von symmetrischen Matrizen einfach, da eine solche Matrix symmetrisch zur Diagonalen gleiche Elemente enthält. Daher kann entweder die obere oder die untere Dreiecksmatrix entfallen. Besonders nützlich ist der Einsatz dieser effizienten Speicherform für Adjazenzmatrizen²¹ bei ungerichteten Graphen.

3.8.14 Die Wahrheit über die Array-Initialisierung *

So schön die kompakte Initialisierung der Feldelemente ist, so laufzeit- und speicherintensiv ist sie auch. Da Java eine dynamische Sprache ist, passt das Konzept der Array-Initialisierung nicht ganz in das Bild. Daher wird die Initialisierung auch erst zur Laufzeit durchgeführt. Unser Primzahlfeld

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

wird vom Java-Compiler umgeformt und analog zu Folgendem behandelt:

```
int[] primes = new int[ 6 ];
primes[ 0 ] = 2;
primes[ 1 ] = 3;
primes[ 2 ] = 5;
primes[ 3 ] = 7;
primes[ 4 ] = 11;
primes[ 5 ] = 13;
```

²¹ Eine Adjazenzmatrix stellt eine einfache Art dar, Graphen zu speichern. Sie besteht aus einem zweidimensionalen Array, das die Informationen über vorhandene Kanten im (gerichteten) Graphen enthält. Existiert eine Kante von einem Knoten zum anderen, so befindet sich in der Zelle ein Eintrag: entweder true/false für »Ja, die beiden sind verbunden« oder ein Ganzahlwert für eine Gewichtung (Kantengewicht).

Erst nach kurzem Überlegen wird das Ausmaß der Umsetzung sichtbar: Zunächst ist es der Speicherbedarf für die Methoden. Ist das Feld `primes` in einer Methode deklariert und mit Werten initialisiert, kostet die Zuweisung Laufzeit, da wir viele Zugriffe haben, die auch alle schön durch die Index-Überprüfung gesichert sind. Da zudem der Bytecode für eine einzelne Methode wegen diverser Beschränkungen in der JVM nur beschränkt lang sein darf, kann dieser Platz für richtig große Arrays schnell erschöpft sein. Daher ist davon abzuraten, etwa Bilder oder große Tabellen im Programmcode zu speichern. Unter C war es populär, ein Programm einzusetzen, das eine Datei in eine Folge von Array-Deklarationen verwandelte. Ist dies in Java wirklich nötig, sollten wir Folgendes in Betracht ziehen:

- Wir verwenden ein statisches Feld (eine Klassenvariable), sodass das Array nur einmal während des Programmlaufs initialisiert werden muss.
- Liegen die Werte im Byte-Bereich, können wir sie in einen String konvertieren und später den String in ein Feld umwandeln. Das ist eine sehr clevere Methode, um Binärdaten einfach unterzubringen.

3.8.15 Mehrere Rückgabewerte *

Wenn wir in Java Methoden schreiben, dann haben sie über `return` höchstens einen Rückgabewert. Wollen wir aber mehr als einen Wert zurückgeben, müssen wir eine andere Lösung suchen. Zwei Ideen lassen sich verwirklichen:

- Behälter wie Arrays oder andere Sammlungen fassen Werte zusammen und liefern sie als Rückgabe.
- Spezielle Behälter werden übergeben, in denen die Methode Rückgabewerte platziert; eine `return`-Anweisung ist nicht mehr nötig.

Betrachten wir eine statische Methode, die für zwei Zahlen die Summe und das Produkt als Array liefert:

Listing 3.28: MultipleReturnValues.java

```
public class MultipleReturnValues
{
    static int[] productAndSum( int a, int b )
    {
        return new int[]{ a * b, a + b };
    }
}
```

```
public static void main( String[] args )  
{  
    System.out.println( productAndSum(9, 3)[ 1 ] );  
}  
}
```

Hinweis

Eine ungewöhnliche Syntax in Java erlaubt es, bei Feldrückgaben das Paar eckiger Klammern auch hinter den Methodenkopf zu stellen, also statt

```
static int[] productAndSum( int a, int b )
```

alternativ Folgendes zu schreiben:

```
static int productAndSum( int a, int b )[]
```

Das ist nicht empfohlen.

3.8.16 Methode mit variabler Argumentanzahl (Vararg)

Bei vielen Methoden ist es klar, wie viele Argumente sie haben; eine Sinus-Methode bekommt ohnehin nur ein Argument. Es gibt jedoch Methoden, bei denen die Zahl mehr oder weniger frei ist, etwa bei der Methode `max()`. Die Klasse `java.lang.Math` sieht eine statische `max()`-Methode mit zwei Argumenten vor, doch grundsätzlich könnte die Methode auch ein Feld entgegennehmen und von diesen Elementen das Maximum bilden. Java 5 sieht eine weitere Möglichkeit vor: Methoden mit *variabler Argumentanzahl*, auch *Varargs* genannt.

Eine Methode mit variabler Argumentanzahl nutzt die Ellipse (»...«) zur Verdeutlichung, dass eine beliebige Anzahl Argumente angegeben werden darf. Der Typ fällt dabei aber nicht unter den Tisch; er wird ebenfalls angegeben:

```
static int max( int... array )  
{  
}
```

Die Methode `max()` behandelt den Parameter `array` wie ein Feld. Da wir Argumente vom Typ `int` fordern, ist `array` vom Typ `int[]` und kann so zum Beispiel mit dem erweiterten `for` durchlaufen werden:

```
for ( int e : array )
```

```
...
```

Werden variable Argumentlisten in der Signatur definiert, so dürfen sie nur den letzten Parameter bilden; andernfalls könnte der Compiler bei den Parametern nicht unbedingt zuordnen, was nun ein Vararg und was schon der nächste gefüllte Parameter ist:

Listing 3.29: MaxVarArgs.java

```
public class MaxVarArgs
{
    static int max( int... array )
    {
        if ( array == null || array.length == 0 )
            throw new IllegalArgumentException( "Array null oder leer" );

        int currentMax = Integer.MIN_VALUE;
        for ( int e : array )
            if ( e > currentMax )
                currentMax = e;
        return currentMax;
    }

    public static void main( String[] args )
    {
        System.out.println( max(1, 2, 9, 3) );      // 9
    }
}
```

Der Nutzer kann jetzt die Methode aufrufen, ohne ein Feld für die Argumente explizit zu definieren. Er bekommt auch gar nicht mit, dass der Compiler im Hintergrund ein Feld mit vier Elementen angelegt hat. So übergibt der Compiler:

```
System.out.println( max( new int[] { 1, 2, 9, 3 } ) );
```

An der Schreibweise lässt sich gut ablesen, dass wir ein Feld auch von Hand übergeben können:

```
int[] feld = { 1, 2, 9, 3 };
System.out.println( max(feld) );
```

Hinweis

Da Varargs als Felder umgesetzt werden, sind überladene Varianten wie `max(int... array)` und `max(int[] array)`, also einmal mit einem Vararg und einmal mit einem Feld, nicht möglich. Besser ist es hier, immer eine Variante mit Varargs zu nehmen, da diese mächtiger ist.

3.8.17 Klonen kann sich lohnen – Arrays vermehren *

Wollen wir eine Kopie eines Arrays mit gleicher Größe und gleichem Elementtyp schaffen, so nutzen wir dazu die Objektmethode `clone()`.²² Sie klonen – in unserem Fall kopiert – die Elemente des Array-Objekts in ein neues.

Listing 3.30: CloneDemo.java, main() Teil 1

```
int[] sourceArray = new int[ 6 ];
sourceArray[ 0 ] = 4711;
int[] targetArray = sourceArray.clone();
System.out.println( targetArray.length ); // 6
System.out.println( targetArray[ 0 ] ); // 4711
```

Im Fall von geklonten Objekt-Feldern ist es wichtig, zu verstehen, dass die Kopie flach ist. Die Verweise aus dem ersten Feld kopiert `clone()` in das neue Feld, es klonen aber die referenzierten Objekte selbst nicht. Bei mehrdimensionalen Arrays wird also nur die erste Dimension kopiert, Unter-Arrays werden somit gemeinsam genutzt:

Listing 3.31: CloneDemo.java, main() Teil 2

```
Point[] pointArray1 = { new Point(1, 2), new Point(2, 3) };
Point[] pointArray2 = pointArray1.clone();
System.out.println( pointArray1[ 0 ] == pointArray2[ 0 ] ); // true
```

Die letzte Zeile zeigt anschaulich, dass die beiden Felder dasselbe Point-Objekt referenzieren; die Kopie ist flach, aber nicht tief.

²² Das ist gültig, da Arrays intern die Schnittstelle `Cloneable` implementieren. `System.out.println(new int[0] instanceof Cloneable);` gibt `true` zurück.

3.8.18 Feldinhalte kopieren *

Eine weitere nützliche statische Methode ist `System.arraycopy()`. Sie kann auf zwei Arten arbeiten:

- *Auf zwei schon existierenden Feldern.* Ein Teil eines Feldes wird in ein anderes Feld kopiert. `arraycopy()` eignet sich dazu, sich vergrößernde Felder zu implementieren, indem zunächst ein neues größeres Feld angelegt wird und anschließend die alten Feldinhalte in das neue Feld kopiert werden.
- *Auf dem gleichen Feld.* So lässt sich die Methode dazu verwenden, Elemente eines Felds um bestimmte Positionen zu verschieben. Die Bereiche können sich durchaus überlappen.

zB

Beispiel

Um zu zeigen, dass `arraycopy()` auch innerhalb des eigenen Feldes kopiert, sollen alle Elemente bis auf eines im Feld `f` nach links und nach rechts bewegt werden:

```
System.arraycopy( f, 1, f, 0, f.length - 1 );      // links
System.arraycopy( f, 0, f, 1, f.length - 1 );      // rechts
```

Hier bleibt jedoch ein Element doppelt!

```
final class java.lang.System
```

- static void `arraycopy(Object src, int srcPos,
Object dest, int destPos, int length)`

Kopiert `length` viele Einträge des Arrays `src` ab der Position `srcPos` in ein Array `dest` ab der Stelle `destPos`. Der Typ des Feldes ist egal, es muss nur in beiden Fällen der gleiche Typ sein. Die Methode arbeitet für große Felder schneller als eine eigene Kopierschleife.

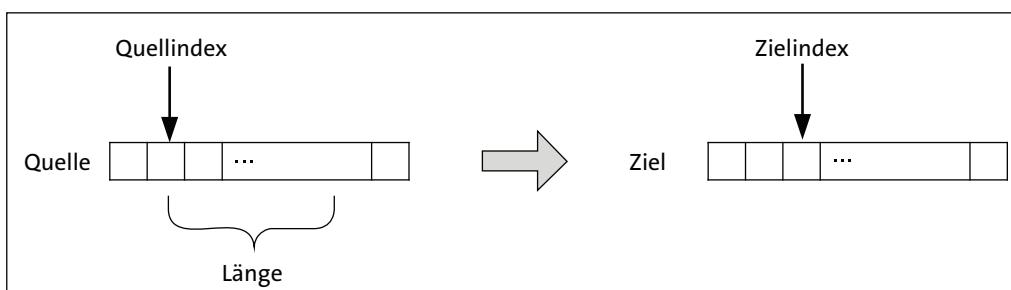


Abbildung 3.8: Kopieren der Elemente von einem Feld in ein anderes

3.8.19 Die Klasse Arrays zum Vergleichen, Füllen, Suchen, Sortieren nutzen

Die Klasse `java.util.Arrays` deklariert nützliche statische Methoden im Umgang mit Arrays. So bietet sie Möglichkeiten zum Vergleichen, Sortieren und Füllen von Feldern sowie zur binären Suche.



Abbildung 3.9: Das UML-Diagramm zeigt viele Methoden in der Klasse Arrays

String-Präsentation eines Feldes

Nehmen wir an, wir haben es mit einem Feld von Hundenamen zu tun, das wir auf dem Bildschirm ausgeben wollen:

Listing 3.32: DogArrayToString, main()

```
String[] dogs = {
    "Flocky Fluke", "Frizzi Faro", "Fanny Favorit", "Frosty Filius",
    "Face Flash", "Fame Frisco"
};
```

Soll der Feldinhalt zum Testen auf den Bildschirm gebracht werden, so kommt eine Ausgabe mit `System.out.println(dogs)` nicht infrage, denn `toString()` ist auf dem Objekttyp Array nicht sinnvoll definiert:

```
System.out.println( dogs ); // [Ljava.lang.String;@10b62c9
```

Die statische Methode `Arrays.toString(array)` liefert für unterschiedliche Feldtypen die gewünschte String-Präsentation des Feldes:

```
System.out.println( Arrays.toString(dogs) ); // [Flocky Fluke, ...]
```

Das spart nicht unbedingt eine `for`-Schleife, die durch das Feld läuft und auf jedem Element `print()` aufruft, denn die Ausgabe ist immer von einem bestimmten Format, das mit »[« beginnt, jedes Element mit »« trennt und mit »]« abschließt.

Die Klasse `Arrays` deklariert die `toString()`-Methode für unterschiedliche Feldtypen:

- | |
|------------------------|
| class java.util.Arrays |
|------------------------|
- static String `toString(XXX[] a)`
Liefert eine String-Präsentation des Feldes. Der Typ `XXX` steht stellvertretend für `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`.
 - static String `toString(Object[] a)`
Liefert eine String-Präsentation des Feldes. Im Fall des Objekttyps ruft die Methode auf jedem Objekt im Feld `toString()` auf.
 - static String `deepToString(Object[] a)`
Ruft auch auf jedem Unterfeld `Arrays.toString()` auf und nicht nur `toString()` wie bei jedem anderen Objekt.

Sortieren

Diverse statische `Arrays.sort()`-Methoden ermöglichen das Sortieren von Elementen im Feld. Bei primitiven Elementen (kein `boolean`) gibt es keine Probleme, da sie eine natürliche Ordnung haben.

Beispiel

zB

Sortiere zwei Felder:

```
int[] numbers = { -1, 3, -10, 9, 3 };
String[] names = { "Xanten", "Alpen", "Wesel" };
Arrays.sort( numbers );
Arrays.sort( names );
System.out.println( Arrays.toString( numbers ) ); // [-10, -1, 3, 3, 9]
System.out.println( Arrays.toString( names ) ); // [Alpen, Wesel, Xanten]
```

Besteht das Feld aus Objektreferenzen, müssen die Objekte vergleichbar sein. Das gelingt entweder mit einem extra `Comparator`, oder die Klassen implementieren die Schnittstelle `Comparable`, wie zum Beispiel `Strings`. Kapitel 8, »Besondere Klassen der Java SE«, beschreibt diese Möglichkeiten präzise.

class `java.util.Arrays`

- `static void sort(XXX[] a)`
Sortiert die gesamte Liste vom Typ `XXX` (wobei `XXX` für `byte`, `char`, `short`, `int`, `long`, `float`, `double` steht) oder einen ausgewählten Teil. Bei angegebenen Grenzen ist `fromIndex` wieder inklusiv und `toIndex` exklusiv. Sind die Grenzen fehlerhaft, löst die Methode eine `IllegalArgumentException` (im Fall `fromIndex > toIndex`) beziehungsweise eine `ArrayIndexOutOfBoundsException` (`fromIndex < 0` oder `toIndex > a.length`) aus.
- `static void sort(Object[] a)`
Sortiert ein Feld von Objekten. Die Elemente müssen `Comparable` implementieren. Bei der Methode gibt es keinen generischen Typ-Parameter.
- `static <T> void sort(T[] a, Comparator<? super T> c)`
Sortiert ein Feld von Objekten mit gegebenem `Comparator`.
- `static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)`
Sortiert ein Feld von Objekten mit gegebenem `Comparator`.

Felder mit Arrays.equals() und Arrays.deepEquals() vergleichen *

Die statische Methode `Arrays.equals()` vergleicht, ob zwei Felder die gleichen Inhalte besitzen; dazu ist die überladene Methode für alle wichtigen Typen definiert. Wenn zwei Felder tatsächlich die gleichen Inhalte besitzen, ist die Rückgabe der Methode `true`, sonst `false`. Natürlich müssen beide Arrays schon die gleiche Anzahl von Elementen besitzen, sonst ist der Test sofort vorbei und das Ergebnis `false`. Im Fall von Objektfeldern nutzt `Arrays.equals()` nicht die Identitätsprüfung per `=`, sondern die Gleichheit per `equals()`.

zB Beispiel

Vergleiche zwei Felder:

```
int[] array1 = { 1, 2, 3, 4 };
int[] array2 = { 1, 2, 3, 4 };
System.out.println( Arrays.equals( array1, array2 ) );      // true
```

Ein Vergleich von Teifeldern ist leider auch nach mehr als zehn Jahren Java-Bibliothek einfach nicht vorgesehen.

Bei unterreferenzierten Feldern betrachtet `Arrays.equals()` das innere Feld als einen Objektverweis und vergleicht es auch mit `equals()` – was jedoch bedeutet, dass nicht identische, aber mit gleichen Elementen referenzierte innere Felder als ungleich betrachtet werden. Die statische Methode `deepEquals()` bezieht auch unterreferenzierte Felder in den Vergleich mit ein.

zB Beispiel

Unterschied zwischen `equals()` und `deepEquals()`:

```
int[][] a1 = { { 0, 1 }, { 1, 0 } };
int[][] a2 = { { 0, 1 }, { 1, 0 } };
System.out.println( Arrays.equals( a1, a2 ) );      // false
System.out.println( Arrays.deepEquals( a1, a2 ) ); // true
System.out.println( a1[0] );                      // zum Beispiel [I@10b62c9
System.out.println( a2[0] );                      // zum Beispiel [I@82ba41
```

Dass die Methoden unterschiedlich arbeiten, zeigen die beiden letzten Konsolenausgaben: Die von `a1` und `a2` unterreferenzierten Felder enthalten die gleichen Elemente, sind aber zwei unterschiedliche Objekte, also nicht identisch.

Hinweis

`deepEquals()` vergleicht auch eindimensionale Felder:

```
Object[] b1 = { "1", "2", "3" };
Object[] b2 = { "1", "2", "3" };
System.out.println( Arrays.deepEquals( b1, b2 ) ); // true
```

class java.util.Arrays

- `static boolean equals(XXX[] a, XXX[] a2)`

Vergleicht zwei Felder gleichen Typs und liefert `true`, wenn die Felder gleich groß und Elemente paarweise gleich sind. `XXX` steht stellvertretend für `boolean, byte, char, int, short, long, double, float`.

- `static boolean equals(Object[] a, Object[] a2)`

Vergleicht zwei Felder mit Objektverweisen. Ein Objekt-Feld darf `null` enthalten; dann gilt für die Gleichheit `e1==null ? e2==null : e1.equals(e2)`.

- `static boolean deepEquals(Object[] a1, Object[] a2)`

Liefert `true`, wenn die beiden Felder ebenso wie alle Unterfelder – rekursiv im Fall von Unter-Objekt-Feldern – gleich sind.

Füllen von Feldern *

`Arrays.fill()` füllt ein Feld mit einem festen Wert. Der Start-Endbereich lässt sich optional angeben.

Beispiel

zB

Fülle ein char-Feld mit Sternchen:

```
char[] chars = new char[ 4 ];
Arrays.fill( chars, '*' );
System.out.println( Arrays.toString( chars ) ); // [*, *, *, *]
```

class java.util.Arrays

- `static void fill(XXX[] a, XXX val)`

- `static void fill(XXX[] a, int fromIndex, int toIndex, XXX val)`

Setzt das Element `val` in das Feld. Mögliche Typen für `XXX` sind `boolean, byte, char, int, short, long, double, float`.

short, int, long, double, float oder mit Object beliebige Objekte. Beim Bereich ist fromIndex inklusiv und toIndex exklusiv.

Feldabschnitte kopieren *

Die Klasse Arrays bietet eine Reihe von copyOf()- bzw. copyOfRange()-Methoden, die gegenüber clone() den Vorteil haben, dass sie auch Bereichsangaben erlauben und das neue Feld größer machen können; im letzten Fall füllen die Methoden das Feld je nach Typ mit null, false oder 0.

zB

Beispiel

```
String[] snow = { "Neuschnee", "Altschnee", "Harsch", "Firn" };
String[] snow1 = Arrays.copyOf( snow, 2 ); // [Neuschnee, Altschnee]
String[] snow2 = Arrays.copyOf( snow, 5 ); // [Neuschnee, Altschnee, Harsch,
                                            // Firn, null]
String[] snow3 = Arrays.copyOfRange( snow, 2, 4 ); // [Harsch, Firn]
String[] snow4 = Arrays.copyOfRange( snow, 2, 5 ); // [Harsch, Firn, null]
```

class java.util.Arrays

- static boolean[] copyOf(boolean[] original, int newLength)
- static byte[] copyOf(byte[] original, int newLength)
- static char[] copyOf(char[] original, int newLength)
- static double[] copyOf(double[] original, int newLength)
- static float[] copyOf(float[] original, int newLength)
- static int[] copyOf(int[] original, int newLength)
- static long[] copyOf(long[] original, int newLength)
- static short[] copyOf(short[] original, int newLength)
- static <T> T[] copyOf(T[] original, int newLength)
- static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType)
- static boolean[] copyOfRange(boolean[] original, int from, int to)
- static byte[] copyOfRange(byte[] original, int from, int to)
- static char[] copyOfRange(char[] original, int from, int to)
- static double[] copyOfRange(double[] original, int from, int to)

- static float[] copyOfRange(float[] original, int from, int to)
- static int[] copyOfRange(int[] original, int from, int to)
- static long[] copyOfRange(long[] original, int from, int to)
- static short[] copyOfRange(short[] original, int from, int to)
- static <T> T[] copyOfRange(T[] original, int from, int to)
- static <T,U> T[] copyOfRange(
U[] original, int from, int to, Class<? extends T[]> newType)

Erzeugt ein neues Feld mit der gewünschten Größe beziehungsweise dem angegebenen Bereich aus einem existierenden Feld. Wie üblich ist der Index `from` inklusiv und `to` exklusiv.

Beispiel *

zB

Hänge zwei Arrays aneinander. Das ist ein gutes Beispiel für `copyOf()`, wenn das Zielfeld größer ist:

```
public static <T> T[] concat( T[] first, T[] second )
{
    T[] result = Arrays.copyOf( first, first.length + second.length );
    System.arraycopy( second, 0, result, first.length, second.length );

    return result;
}
```

Hinweis: Das Beispiel nutzt Generics, um den Typ flexibel zu halten. Zum einfacheren Verständnis können wir uns `T` als Typ `Object` vorstellen und das, was in spitzen Klammern ist, löschen.

Halbierungssuche *

Ist das Feld sortiert, lässt sich mit `Arrays.binarySearch()` eine binäre Suche (Halbierungssuche) durchführen. Ist das Feld nicht sortiert, ist das Ergebnis unvorhersehbar. Findet `binarySearch()` das Element, ist der Rückgabewert der Index der Fundstelle, andernfalls ist die Rückgabe negativ.

zB Beispiel

Der beste Fall ist, dass sich das Element im Feld befindet:

```
int[] numbers = { 1, 10, 100, 1000 };
System.out.println( Arrays.binarySearch( numbers, 100 ) ); // 2
```

Ist das Feld nicht sortiert, ist ein falsches Ergebnis die Folge.

```
int[] wrong = { 10, 100, 1000, 1 };
System.out.println( Arrays.binarySearch( wrong, 100 ) ); // 1
```

`binarySearch()` liefert in dem Fall, in dem das Element nicht im Feld ist, eine kodierte Position zurück, an der das Element eingefügt werden könnte. Damit die Zahl nicht mit einer normalen Position kollidiert – die immer ≥ 0 ist –, ist die Rückgabe negativ und als *-Einfügeposition - 1* kodiert.

zB Beispiel

Das Element 101 ist nicht im Feld:

```
int[] numbers = { 1, 10, 100, 1000 };
System.out.println( Arrays.binarySearch( numbers, 101 ) ); // -4
```

Die Rückgabe ist -4 , denn $-4 = -3 - 1$, was eine mögliche Einfügeposition von 3 ergibt.

Das ist korrekt, denn 101 käme wie folgt ins Feld: 1 (Position 0), 10 (Position 1), 100 (Position 2), 101 (Position 3).

**Tipp**

Da das Feld bei `Arrays.binarySearch()` zwingend sortiert sein muss, kann ein vorangehendes `Arrays.sort()` dies vorbereiten.

```
int[] numbers = { 10, 100, 1000, 1 };
Arrays.sort( numbers );
System.out.println( Arrays.toString( numbers ) ); // [1, 10, 100, 1000]
System.out.println( Arrays.binarySearch( numbers, 100 ) ); // 2
```

Die Sortierung ist nur einmal nötig und sollte nicht unnötigerweise wiederholt werden.

class java.util.Arrays

- static int binarySearch(XXX[] a, XXX key)
Sucht mit der Halbierungssuche nach einem Schlüssel. XXX steht stellvertretend für byte, char, int, long, float, double.
- static int binarySearch(Object[] a, Object key)
Sucht mit der Halbierungssuche nach key. Die Objekte müssen die Schnittstelle Comparable implementieren; das bedeutet im Allgemeinen, dass die Elemente vom gleichen Typ sein müssen – also nicht Strings und Hüpfburg-Objekte gemischt.
- static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
Sucht mit der Halbierungssuche ein Element im Objektfeld. Die Vergleiche übernimmt ein spezielles Vergleichsobjekt c.
- static <T> int binarySearch(
 T[] a, int fromIndex, int toIndex, T key, Comparator<? super T> c)
Schränkt die Binärsuche auf Bereiche ein.

Die API-Dokumentation von `binarySearch()` ist durch Verwendung der Generics (mehr darüber folgt in Kapitel 9, »Generics<T>«) etwas schwieriger. Wir werden in Kapitel 13, »Einführung in Datenstrukturen und Algorithmen«, auch noch einmal auf die statische Methode `binarySearch()` für beliebige Listen zurückkommen und insbesondere die Bedeutung der Schnittstellen `Comparator` und `Comparable` in Kapitel 8, »Besondere Klassen der Java SE«, genau klären.

Felder zu Listen mit `Arrays.asList()` – praktisch für die Suche und zum Vergleichen *

Ist das Feld unsortiert, funktioniert `binarySearch()` nicht. Die Klasse `Arrays` hat für diesen Fall keine Methode im Angebot – eine eigene Schleife muss her. Es gibt aber noch eine Möglichkeit: Die statische Methode `Arrays.asList()` dekoriert das Array als Liste vom Typ `java.util.List`, die dann praktische Methoden wie `contains()`, `equals()` oder `sublist()` anbietet. Mit den Methoden sind Dinge auf Feldern möglich, für die `Arrays` bisher keine Methoden definierte.

Beispiel

zB

Teste, ob auf der Kommandozeile der Schalter `-?` gesetzt ist. Die auf der Kommandozeile übergebenen Argumente übergibt die Laufzeitumgebung als String-Feld an die `main(String[] args)`-Methode:

```
if ( Arrays.asList( args ).contains( "-?" ) )
    ...
```

zB Beispiel

Teste, ob Teile zweier Felder gleich sind:

```
// Index      0          1          2
String[] a = { "Asus",      "Elitegroup", "MSI" };
String[] b = { "Elitegroup", "MSI",       "Shuttle" };

System.out.println( Arrays.asList( a ).subList( 1, 3 ) .
                      equals( Arrays.asList( b ).subList( 0, 2 ) ) ); // true
```

Im Fall von `subList()` ist der Start-Index inklusiv und der End-Index exklusiv (das ist die Standardnotation von Bereichen in Java, etwa auch bei `substring()` oder `fill()`). Somit werden im obigen Beispiel die Einträge 1 bis 2 aus a mit den Einträgen 0 bis 1 aus b verglichen.

class java.util.Arrays

- static <T> List<T> asList(T... a)
Liefert eine Liste vom Typ T bei einem Feld vom Typ T.

Die statische Methode `asList()` nimmt über das Vararg entweder ein Feld von Objekten (kein primitives Feld!) an oder aufgezählte Elemente.

→ Hinweis

Im Fall der aufgezählten Elemente ist auch kein oder genau ein Element erlaubt:

```
System.out.println( Arrays.asList() );           // []
System.out.println( Arrays.asList("Chris") );    // [Chris]
```

→ Hinweis

Ein an `Arrays.asList()` übergebene primitives Feld liefert keine Liste von primitiven Elementen (es gibt keine List, die mit primitiven Werten gefüllt ist):

```
int[] nums = { 1, 2 };
System.out.println( Arrays.asList(nums).toString() ); // [[I@82ba41]
System.out.println( Arrays.toString(nums) );         // [1, 2]
```

Hinweis (Forts.)

Der Grund ist einfach: `Arrays.asList()` erkennt `nums` nicht als Feld von Objekten, sondern als genau ein Element einer Aufzählung. So setzt die statische Methode das Feld mit Primitiven als ein Element in die Liste, und die Objektmethode `toString()` eines `java.util.List`-Objekts ruft lediglich auf dem Feld-Objekt `toString()` auf, was die kryptische Ausgabe zeigt.

3.8.20 Eine lange Schlange

Das neu erworbene Wissen über Felder wollen wir für unser Schlangenspiel nutzen, indem die Schlange länger wird. Bisher hatte die Schlange keine Länge, sondern nur eine Position auf dem Spielbrett; das wollen wir ändern, indem sich ein Programm im Feld immer die letzten Positionen merken soll. Folgende Änderungen sind nötig:

- Anstatt die Position in einem `Point`-Objekt zu speichern, liegen die letzten fünf Positionen in einem `Point[]` `snakePositions`.
- Trifft der Spieler einen dieser fünf Schlangen-Punkte, ist das Spiel verloren. Ist bei der Bildschirmausgabe eine Koordinate gleich einem der Schlangenpunkte, zeichnen wie ein »S«. Der Test, ob eine der Schlangenkoordinaten einen Punkt `p` trifft, wird mit `Arrays.asList(snakePositions).contains(p)` durchgeführt.

Ein weiterer Punkt ist, dass die Schlange sich bewegt, aber das Feld mit den 5 Punkten immer nur die letzten 5 Bewegungen speichert – die alten Positionen werden verworfen. Das Programm realisiert das mit einem Ringspeicher – zusätzlich zu den Positionen verwalten wir einen Index, der auf den Kopf zeigt. Jede Bewegung der Schlange setzt den Index eine Position weiter, bis am Ende des Feldes der Index wieder bei 0 steht.

Eine symbolische Darstellung mit möglichen Punkten verdeutlicht dies:

```
snakeIdx = 0
snakePositions = { [2,2], null, null, null, null }
snakeIdx = 1
snakePositions = { [2,2], [2,3], null, null, null }
...
snakeIdx = 4
snakePositions = { [2,2], [2,3], [3,3], [3,4], [4,4] }
snakeIdx = 0
snakePositions = { [5,5], [2,3], [3,3], [3,4], [4,4] }
```

Alte Positionen werden überschrieben und durch neue ersetzt.

Das ganze Spiel mit den Änderungen, die fett hervorgehoben sind:

Listing 3.33: LongerZZZZnake.java

```
import java.awt.Point;
import java.util.Arrays;

public class LongerZZZZnake
{
    public static void main( String[] args )
    {
        Point playerPosition = new Point( 10, 9 );
        Point goldPosition = new Point( 6, 6 );
        Point doorPosition = new Point( 0, 5 );
        Point[] snakePositions = new Point[5];
        int snakeIdx = 0;
        snakePositions[ snakeIdx ] = new Point( 30, 2 );
        boolean rich = false;

        while ( true )
        {
            if ( rich && playerPosition.equals( doorPosition ) )
            {
                System.out.println( "Gewonnen!" );
                break;
            }
            if ( Arrays.asList( snakePositions ).contains( playerPosition ) )
            {
                System.out.println( "ZZZZZZZ. Die Schlange hat dich!" );
                break;
            }
            if ( playerPosition.equals( goldPosition ) )
            {
                rich = true;
                goldPosition.setLocation( -1, -1 );
            }
        }
    }
}
```

```
// Raster mit Figuren zeichnen

for ( int y = 0; y < 10; y++ )
{
    for ( int x = 0; x < 40; x++ )
    {
        Point p = new Point( x, y );
        if ( playerPosition.equals( p ) )
            System.out.print( '&' );
        else if ( Arrays.asList( snakePositions ).contains( p ) )
            System.out.print( 'S' );
        else if ( goldPosition.equals( p ) )
            System.out.print( '$' );
        else if ( doorPosition.equals( p ) )
            System.out.print( '#' );
        else
            System.out.print( '.' );
    }
    System.out.println();
}

// Konsoleneingabe und Spielerposition verändern

switch ( new java.util.Scanner( System.in ).next().charAt( 0 ) )
{
    case 'h' : playerPosition.y = Math.max( 0, playerPosition.y - 1 ); break;
    case 't' : playerPosition.y = Math.min( 9, playerPosition.y + 1 ); break;
    case 'l' : playerPosition.x = Math.max( 0, playerPosition.x - 1 ); break;
    case 'r' : playerPosition.x = Math.min( 39, playerPosition.x + 1 ); break;
}

// Schlange bewegt sich in Richtung Spieler

Point snakeHead = new Point( snakePositions[snakeIdx].x,
                            snakePositions[snakeIdx].y );
```

```

        if ( playerPosition.x < snakeHead.x )
            snakeHead.x--;
        else if ( playerPosition.x > snakeHead.x )
            snakeHead.x++;
        if ( playerPosition.y < snakeHead.y )
            snakeHead.y--;
        else if ( playerPosition.y > snakeHead.y )
            snakeHead.y++;

        snakeIdx = (snakeIdx + 1) % snakePositions.length;
        snakePositions[snakeIdx] = snakeHead;
    } // end while
}
}

```

3.9 Der Einstiegspunkt für das Laufzeitsystem: main()

In Java-Klassen gibt es eine besondere statische Methode `main()`, die das Laufzeitsystem in der angegebenen Hauptklasse (oder Startklasse) des Programms aufruft.

3.9.1 Korrekte Deklaration der Startmethode

Damit die JVM ein Java-Programm starten kann, muss es eine besondere Methode `main()` geben. Da die Groß-/Kleinschreibung in Java relevant ist, muss sie `main()` lauten, und nicht `Main()` oder `MAIN()`. Die Sichtbarkeit ist auf `public` gesetzt, und die Methode muss statisch sein, da die JVM die Methode auch ohne Exemplar der Klasse aufrufen möchte. Als Parameter wird ein Array von `String`-Objekten angenommen. Darin sind die auf der Kommandozeile übergebenen Parameter abgelegt.

Zwei Varianten gibt es zur Deklaration:

- `public static void main(String[] args)`
- `public static void main(String... args)`

Die zweite funktioniert seit Java 5 und nutzt variable Argumentlisten, ist aber mit der ersten Version gleich.

Falsche Deklarationen

Nur eine Methode mit dem Kopf `public static void main(String[] args)` wird als Startmethode akzeptiert. Ein Methodenkopf wie `public static void Main(String[] args)` ist durchaus gültig, aber eben keiner, der die JVM zum Start ansteuern würde. Findet die JVM die Startmethode nicht, gibt sie eine Fehlermeldung aus:

Fehler: Hauptmethode in Klasse ABC nicht gefunden. Definieren Sie die Hauptmethode als:

```
public static void main(String[] args)
```

Hinweis

Im Gegensatz zu C(++) steht im ersten Element des Argument-Arrays mit Index 0 nicht der Programmname, also der Name der Hauptklasse, sondern bereits der erste Programmparameter der Kommandozeile.



3.9.2 Kommandozeilenargumente verarbeiten

Eine besondere Variable für die Anzahl der übergebenen Argumente der Kommandozeile ist nicht erforderlich, weil das String-Array-Objekt uns diese Information über `length` mitteilt. Um etwa alle übergebenen Argumente über die erweiterte `for`-Schleife auszugeben, schreiben wir:

Listing 3.34: LovesGoldenHamster.java, main()

```
public static void main( String[] args )
{
    if ( args.length == 0 )
        System.out.println( "Was!! Keiner liebt kleine Hamster?" );
    else
    {
        System.out.print( "Liebt kleine Hamster: " );

        for ( String s : args )
            System.out.format( "%s ", s );

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

Das Programm lässt sich auf der Kommandozeile wie folgt aufrufen:

```
$ java LovesGoldenHamster Raphael Perly Mirjam Paul
```



Bibliothek

Zum Parsen der Kommandozeilenargumente bietet sich zum Beispiel die Bibliothek *Jakarta Commons CLI* (<http://jakarta.apache.org/commons/cli/>) an.

3.9.3 Der Rückgabetyp von `main()` und `System.exit()` *

Der Rückgabetyp `void` der Startmethode `main()` ist sicherlich diskussionswürdig, da diejenigen, die die Sprache entworfen haben, auch hätten fordern können, dass ein Programm immer einen Statuscode an das aufrufende Programm zurückgibt. Für diese Lösung haben sie sich aber nicht entschieden, da Java-Programme in der Regel nur minimal mit dem umgebenden Betriebssystem interagieren sollen und echte Plattform-unabhängigkeit gefordert ist, etwa bei Java in Handys.

Für die Fälle, in denen ein Statuscode zurückgeliefert werden soll, steht die statische Methode `System.exit(status)` zur Verfügung; sie beendet eine Applikation. Das an `exit()` übergebene Argument nennt sich *Statuswert* (engl. *exit status*) und wird an die Kommandozeile zurückgegeben. Der Wert ist für Skriptprogramme wichtig, da sie über diesen Rückgabewert auf das Gelingen oder Misserfolg des Java-Programms reagieren können. Ein Wert von 0 zeigt per Definition das Gelingen an, ein Wert ungleich 0 einen Fehler. Der Wertebereich sollte sich zwischen 0 und 255 bewegen. Auf der Unix-Kommandozeile ist der Rückgabewert eines Programms unter `$?` verfügbar und in der Windows `cmd.exe` unter `%ERRORLEVEL%`, einer Art dynamischer Umgebungsvariable.

Dazu ein Beispiel. Ein Java-Programm liefert den Statuswert 42:

Listing 3.35: SystemExitDemo.java

```
public class SystemExitDemo
{
    public static void main( String[] args )
    {
        System.exit( 42 );
    }
}
```

Das folgende Shell-Programm gibt den Statuswert zunächst aus und zeigt zudem, welche Fallunterscheidung die Shell für Statuswerte bietet:

Listing 3.36: showreturn.bat

```
@echo off
java SystemExitDemo
echo %ERRORLEVEL%
if errorlevel 10 (
    echo Exit-Code ist über 10, genau %ERRORLEVEL%
)
)
```

Die JVM startet das Java-Programm und beendet es mit `System.exit()`, was zu einer Belegung der Variable `%ERRORLEVEL%` mit 42 führt. Das Skript gibt zunächst die Belegung der Variablen aus. Die Windows Shell besitzt mit `if errorlevel Wert` eine spezielle Variante für Fallunterscheidungen mit Exit-Codes, die genau dann greift, wenn der aktuelle Exit-Code größer oder gleich dem angegebenen Wert ist. Das heißt in unserem Beispiel: Es gibt eine Ausgabe, wenn der Exit-Code größer 10 ist, und mit 42 ist er das. Daher folgt die Ausgabe vom kleinen Skript:

```
>showreturn.bat
42
Error-Level ist über 10, genau 42
```

Es ist wichtig zu bedenken, dass `%ERRORLEVEL%` natürlich überschrieben wird, wenn Befehle folgen. So gibt Folgendes nur 0 aus, da `dir` erfolgreich abgeschlossen werden kann und `dir` nach der Durchführung den Exit-Code auf 0 setzt:

```
java SystemExitDemo
dir
echo %ERRORLEVEL%
```

Liegen zwischen dem Aufruf der JVM und der Auswertung der Variablen Aufrufe, die den Exit-Code verändern, ist es sinnvoll, den Inhalt von `%ERRORLEVEL%` zwischenspeichern, etwa so:

Listing 3.37: showreturn2.bat

```
@echo off
java SystemExitDemo
SET EXITCODE=%ERRORLEVEL%
dir > NUL:
```

```
echo %ERRORLEVEL%
echo %EXITCODE%
```

Die Ausgabe ist dann:

```
0
42
```

```
final class java.lang.System
```

- static void exit(int status)

Beendet die aktuelle JVM und gibt das Argument der Methode als Statuswert zurück. Ein Wert ungleich null zeigt einen Fehler an. Also ist der Rückgabewert beim normalen fehlerfreien Verlassen null. Eine SecurityException wird ausgelöst, falls der aktuelle SecurityManager dem aufrufenden Code nicht erlaubt, die JVM zu beenden. Das gilt insbesondere bei Applets in einem Webbrowser.

3.10 Annotationen und Generics

Bis zu diesen Punkt haben wir uns mit den Grundlagen der Objektorientierung beschäftigt und wissen, wie Objekte aufgebaut und Eigenschaften genutzt werden.

Mit zwei weiteren Eigenschaften der Programmiersprache wollen wir uns kurz beschäftigen: Annotationen und Generics. Beiden Punkten nähern wir uns aus der Nutzerperspektive und nicht aus der Sicht eines API-Designers, der neue Annotationen zur Verfügung stellen muss oder neue Methoden und Klassen deklariert und diese generisch parametrisierbar ausstatten möchte.

3.10.1 Generics

Java ist eine typisierte Programmiersprache, was bedeutet, dass jede Variable und jeder Ausdruck einen Typ hat, den der Compiler kennt und der sich zur Laufzeit nicht ändert. Eine Zählvariable ist zum Beispiel vom Typ `int`, ein Abstand zwischen zwei Punkten ist vom Typ `double`, und ein Koordinatenpaar ist vom Typ `Point`. Allerdings gibt es bei der Typisierung Lücken. Nehmen wir etwa eine Liste von Punkten:

```
List list;
```

Zwar ist die Variable `list` nun mit `List` typisiert, und das ist besser als nichts, jedoch bleibt unklar, was die Liste eigentlich genau für Objekte speichert. Sind es Punkte, Per-

sonen oder rostige Fähren? Es wäre sinnvoll, nicht nur die Liste selbst als Typ zu haben, sondern sozusagen rekursiv in die Liste reinzugehen und genauen hinzuschauen, was die Liste eigentlich referenziert. Genau das ist die Aufgabe von Generics. Eine Liste erwartet seit Java 5 eine Typangabe, was sie genau speichert. Dieser Typ erscheint in spitzen Klammern hinter dem eigentlichen »Haupttyp«.

```
List<Point> list;
```

Mit Generics haben API-Designer ein Werkzeug, um Typen noch genauer vorzuschreiben. Die Entwickler des Typs `List` können so vom Nutzer fordern, den Elementtyp anzugeben. So können Entwickler dem Compiler genauer sagen, was sie für Typen verwenden, und es dem Compiler ermöglichen, genauere Tests zu machen. Es ist erlaubt und möglich, diesen »Nebentyp« nicht anzugeben, doch das führt zu einer Compiler-Warnung und ist nicht empfehlenswert: Je genauer Typangaben sind, desto besser ist das für alle.

Vereinzelt kommen in den nächsten Kapiteln generische Typen vor, etwa `Comparable` (hilft Objekte zu vergleichen). An dieser Stelle reicht es zu verstehen, dass wir als Nutzer einen Typ in spitze Klammern eintragen müssen. Mit Generics selbst beschäftigen wir uns in Kapitel 9 genauer.

3.10.2 Annotationen

In diesem Kapitel haben wir schon unterschiedliche Modifizierer kennengelernt. Darunter waren zum Beispiel `static` oder `public`. Das Besondere an diesen Modifizierern ist, dass sie die Programmsteuerung nicht beeinflussen, aber dennoch wichtige Zusatzinformationen darstellen, also Semantik einbringen. Diese Informationen nennen sich *Metadaten*. Die Modifizierer `static`, `public` sind Metadaten für den Compiler, doch mit etwas Fantasie lassen sich auch Metadaten vorstellen, die nicht vom Compiler, sondern von einer Java-Bibliothek ausgewertet werden. So wie `public` zum Beispiel dem Compiler sagt, dass ein Element für jeden sichtbar ist, kann auch auf der anderen Seite zum Beispiel ein besonderes Metadatum an einem Element hängen, um auszudrücken, dass es nur bestimmte Wertebereiche annehmen kann.

3.10.3 Eigene Metadaten setzen

Seit Java 5 gibt es eine in die Programmiersprache eingebaute Fähigkeit für Metadaten: *Annotationen*. Die Annotationen lassen sich wie benutzerdefinierte Modifizierer erklären. Wir können zwar keine neue Sichtbarkeit erfinden, aber dennoch dem Compiler,

bestimmten Werkzeugen oder der Laufzeitumgebung durch die Annotationen Zusatzinformationen geben. Dazu ein paar Beispiele für Annotationen und Anwendungsfälle.

Annotation	Erklärung
<code>@WebService class Calculator {</code> <code> @WebMethod int add(int x, int y) ...</code>	Definiert einen Web-Service mit einer Web-Service-Methode.
<code>@Override public String toString() ...</code>	Überschreibt eine Methode der Oberklasse.
<code>@XmlRoot class Person { ...</code>	Ermöglicht die Abbildung eines Objekts auf eine XML-Datei.

Tabelle 3.5: Beispiele für Annotationen und Anwendungsfälle

Die Tabelle soll lediglich einen Überblick geben; genaue Anwendungen und Beispiele folgen.

Annotationen werden wie zusätzliche Modifizierer gebraucht, doch unterscheiden sie sich durch ein vorangestelltes @-Zeichen (das @-Zeichen, AT, ist auch eine gute Abkürzung für *Annotation Type*). Daher ist auch die Reihenfolge egal, sodass es zum Beispiel

- `@Override public String toString() oder`
- `public @Override String toString()`

lauten kann. Es ist aber üblich, die Annotationen an den Anfang zu setzen.

3.10.4 Annotationstypen `@Override`, `@Deprecated`, `@SuppressWarnings`

Das Paket `java.lang` deklariert vier Annotationstypen (einer davon ist neu in Java 7), wobei uns `@Override` ab Kapitel 5, »Eigene Klassen schreiben«, noch häufiger über den Weg laufen wird.

Annotationstyp	Wirkung
<code>@Override</code>	Die annotierte Methode überschreibt eine Methode aus der Oberklasse oder implementiert eine Methode einer Schnittstelle.
<code>@Deprecated</code>	Das markierte Element ist veraltet und sollte nicht mehr verwendet werden.

Tabelle 3.6: Annotationen aus dem Paket »java.lang«

Annotationstyp	Wirkung
@SuppressWarnings	Unterdrückt bestimmte Compiler-Warnungen.
@SafeVarargs	Besondere Markierung für Methoden mit variabler Argumentanzahl und generischem Argumenttyp

Tabelle 3.6: Annotationen aus dem Paket »java.lang« (Forts.)

Die vier Annotationen haben vom Compiler beziehungsweise Laufzeitsystem eine besondere Semantik. Java SE deklariert in anderen Paketen (wie dem javax.annotation-Paket) noch weitere allgemeine Annotationstypen, doch die sind an dieser Stelle nicht relevant. Dazu kommen spezielle technologiespezifische Annotationstypen wie für die XML-Objekt-Abbildung oder Web-Service-Deklarationen.

Die Begriffe »Annotation« und »Annotationstyp«

Die *Annotationstypen* sind die Deklarationen, wie etwa ein Klassentyp. Werden sie an ein Element gehängt, ist es eine konkrete *Annotation*. Während also `Override` selbst der Annotationstyp ist, ist `@Override` vor `toString()` die konkrete Annotation.

@Deprecated

Die Annotation `@Deprecated` übernimmt die gleiche Aufgabe wie das JavaDoc-Tag `@deprecated`: Die markierten Elemente werden als veraltet markiert und drücken damit aus, dass der Entwickler Alternativen nutzen soll.

Beispiel

zB

Die Methode `fubar()`²³ soll als veraltet markiert werden:

`@Deprecated`

```
public void fubar() { ... }
```

Ruft irgendein Programmstück `fubar()` auf, gibt der Compiler eine einfache Meldung aus.

Die Übersetzung mit dem Schalter `-Xlint:deprecation` liefert die genauen Warnungen; im Moment ist das mit `-deprecation` gleich.

²³ Im US-Militär-Slang steht das für: » Fucked up beyond any recognition « – »vollkommen ruinert«.

Auch über ein JavaDoc-Tag kann ein Element als veraltet markiert werden. Ein Unterschied bleibt: Das JavaDoc-Tag kann nur von JavaDoc (beziehungsweise einem anderen Doclet) ausgewertet werden, während Annotationen auch andere Tools auswerten können.

Annotationen mit zusätzlichen Informationen

Die Annotationen `@Override` und `@Deprecated` gehören zur Klasse der Marker-Annotationen, weil keine zusätzlichen Angaben nötig (und erlaubt) sind. Zusätzlich gibt es die *Single-Value Annotation*, die genau eine zusätzliche Information bekommt, und eine volle Annotation mit beliebigen Schlüssel/Werte-Paaren.

Schreibweise der Annotation	Funktion
<code>@Annotationstyp</code>	(Marker-)Annotation
<code>@Annotationstyp(Wert)</code>	Annotation mit genau einem Wert
<code>@Annotationstyp(Schlüssel1=Wert1, Schlüssel2=Wert2, ...)</code>	Annotation mit Schlüssel/Werte-Paaren

Tabelle 3.7: Annotationen mit und ohne zusätzliche Informationen

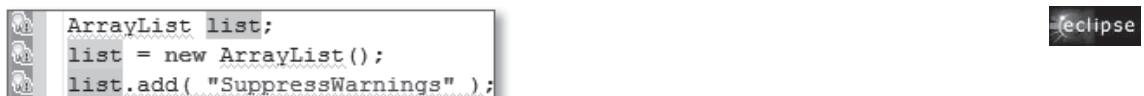
Klammern sind bei einer Marker-Annotation optional.

`@SuppressWarnings`

Die Annotation `@SuppressWarnings` steuert Compiler-Warnungen. Unterschiedliche Werte bestimmen genauer, welche Hinweise unterdrückt werden. Nützlich ist die Annotation bei der Umstellung von Quellcode, der vor Java 5 entwickelt wurde. Mit Java 5 zogen Generics ein, eine Möglichkeit, dem Compiler noch mehr Informationen über Typen zu geben. Die Java API-Designer haben daraufhin die Deklaration der Datenstrukturen überarbeitet und Generics eingeführt, was dazu führt, dass vor Java 5 entwickelter Quellcode mit einem Java 5-Compiler eine Vielzahl von Warnungen ausgibt. Nehmen wir folgenden Programmcode:

```
ArrayList list;
list = new ArrayList();
list.add( "SuppressWarnings" );
```

Eclipse zeigt die Meldungen direkt an, NetBeans dagegen standardmäßig nicht.



The screenshot shows a Java code editor in Eclipse. The code is:

```
ArrayList list;
list = new ArrayList();
list.add( "SuppressWarnings" );
```

Two warning icons are shown next to the first two lines of code. The status bar at the bottom right says "eclipse".

Abbildung 3.10: Warnungen in Eclipse

Der Compiler `javac` meldet über die Kommandozeile recht unspezifisch:

```
Note: ABC.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Mit dem gesetzten Schalter `-Xlint` heißt es dann genauer:

```
warning: [rawtypes] found raw type: ArrayList
    ArrayList list1;
    ^
missing type arguments for generic class ArrayList<E>
where E is a type-variable:
    E extends Object declared in class ArrayList
```

```
warning: [rawtypes] found raw type: ArrayList
    list1 = new ArrayList();
    ^
missing type arguments for generic class ArrayList<E>
where E is a type-variable:
    E extends Object declared in class ArrayList
```

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type ArrayList
    list1.add("SuppressWarnings");
    ^
where E is a type-variable:
    E extends Object declared in class ArrayList
```

Zwei unterschiedliche Arten von Warnungen treten auf:

- Da die Klasse `ArrayList` als generischer Typ deklariert ist, melden die ersten beiden Zeilen »found raw type: `ArrayList`« (`javac`) bzw. »`ArrayList` is a raw type. References to generic type `ArrayList<E>` should be parameterized« (Eclipse).
- Die dritte Zeile nutzt mit `add()` eine Methode, die über Generics einen genaueren Typparameter bekommen könnte. Da wir einen Typ aber nicht angegeben haben, folgt

die Warnung: »unchecked call to add(E) as a member of the raw type ArrayList« (javac) bzw. »Type safety: The method add(Object) belongs to the raw type ArrayList. References to generic type ArrayList<E> should be parameterized« (Eclipse).

Warnungen lassen sich über die Annotation `@SuppressWarnings` ausschalten. Als spezieller Modifizierer lässt sich die Annotation an der Variablen Deklaration anbringen, an der Methodendeklaration oder an der Klassendeklaration. Die Reichweite ist aufsteigend. Wer bei altem Programmcode kurz und schmerzlos alle Warnungen abschalten möchte, der setzt ein `@SuppressWarnings("all")` an die Klassendeklaration.

zB Beispiel

Der Compiler soll keine Meldungen für die Klasse geben:

```
@SuppressWarnings( "all" )
public class SuppressAllWarnings
{
    public static void main( String[] args )
    {
        java.util.ArrayList list1 = new java.util.ArrayList();
        list1.add( "SuppressWarnings" );

        java.util.ArrayList list2 = new java.util.ArrayList();
    }
}
```

Anstatt jede Warnung zu unterdrücken, ist es eine bessere Strategie, selektiv vorzugehen. Eclipse unterstützt uns mit einem Quick-Fix und schlägt für unser Beispiel Folgendes vor:

- `@SuppressWarnings("rawtypes")` für `ArrayList list` und `list = new ArrayList()`
- `@SuppressWarnings("unchecked")` für `list.add(...)`

Da zwei gleiche Modifizierer nicht erlaubt sind – und auch zweimal `@SuppressWarnings` nicht –, wird eine besondere Array-Schreibweise gewählt.

zB Beispiel

Der Compiler soll für die ungenerisch verwendete Liste und deren Methoden keine Meldungen geben:

zB

3

Beispiel (Forts.)

```
@SuppressWarnings( { "rawtypes", "unchecked" } )
public static void main( String[] args )
{
    ArrayList list = new ArrayList();
    list.add( "SuppressWarnings" );
}
```

Kurz kam bereits zur Sprache, dass die `@SuppressWarnings`-Annotation auch an der Variablen Deklaration möglich ist. Für unser Beispiel hilft das allerdings wenig, wenn etwa bei der Deklaration der Liste alle Warnungen abgeschaltet werden:

```
@SuppressWarnings( "all" ) ArrayList list;
list = new ArrayList();           // Warnung: ArrayList is a raw type...
list.add( "SuppressWarnings" );   // Warnung: Type safety ...
```

Das `@SuppressWarnings("all")` gilt nur für die eine Deklaration `ArrayList list` und nicht für folgende Anweisungen, die etwas mit der `list` machen. Zur Verdeutlichung setzt das Beispiel die Annotation daher in die gleiche Zeile.

Hinweis

Die Schreibweise `@SuppressWarnings("xyz")` ist nur eine Abkürzung von `@SuppressWarnings({ "xyz" })`, und das wiederum ist nur eine Abkürzung von `@SuppressWarnings(value= { "xyz" })`.

3.11 Zum Weiterlesen

In diesem Kapitel wurde das Thema Objektorientierung recht schnell eingeführt, was nicht bedeuten soll, dass OOP einfach ist. Der Weg zu gutem Design ist steinig und führt nur über viele Java-Projekte. Hilfreich sind das Lesen von fremden Programmen und die Beschäftigung mit Entwurfsmustern. Rund um UML ist ebenfalls eine Reihe von Produkten entstanden. Das Angebot beginnt bei einfachen Zeichenwerkzeugen, geht über UML-Tools mit Roundtrip-Fähigkeit und reicht bis zu kompletten CASE-Tools mit MDA-Fähigkeit.



Kapitel 4

Der Umgang mit Zeichenketten

4

»Ohne Unterschied macht Gleichheit keinen Spaß.«
– Dieter Hildebrandt (*1927)

4.1 Von ASCII über ISO-8859-1 zu Unicode

Die Übertragung von Daten spielte in der IT schon immer eine zentrale Rolle. Daher haben sich unterschiedliche Standards herausgebildet. Sie sind Gegenstand der nächsten Abschnitte.

4.1.1 ASCII

Um Dokumente austauschen zu können, führte die *American Standards Association* im Jahr 1963 eine 7-Bit-Kodierung ein, die *ASCII* (von *American Standard Code for Information Interchange*) genannt wird. ASCII gibt jedem der 128 Zeichen (mehr Zeichen passen in 7 Bit nicht hinein) eine eindeutige Position, die *Codepoint* (*Codeposition*) genannt wird. Es gibt 94 druckbare Zeichen (Buchstaben, Ziffern, Interpunktionszeichen), 33 nicht druckbare Kontrollzeichen (etwa den Tabulator und viele andere Zeichen, die bei Fernschreibern nützlich waren, aber heute uninteressant sind), und das Leerzeichen, das nicht als Kontrollzeichen gezählt wird. Am Anfang des ASCII-Alphabets stehen an den Positionen 0–31 Kontrollzeichen, an Stelle 32 folgt das Leerzeichen, und anschließend kommen alle druckbaren Zeichen. An der letzten Position, 127, wird ASCII von einem Kontrollzeichen abgeschlossen.

Die folgende Tabelle stammt aus dem Originalstandard von 1968 und gibt einen Überblick über die Position der Zeichen.

USASCII code chart							
b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	Column
Row		0	0	0	1	0	1
b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	Column
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	NUL
0	0	0	0	1	1	1	DLE
0	0	0	1	1	1	1	SP
0	0	1	0	2	2	2	!
0	0	1	0	2	2	2	SOH
0	0	1	0	2	2	2	DC1
0	0	1	1	3	3	3	STX
0	0	1	1	3	3	3	DC2
0	1	0	0	4	4	4	ETX
0	1	0	0	4	4	4	DC3
0	1	0	0	4	4	4	EOT
0	1	0	0	4	4	4	DC4
0	1	0	1	5	5	5	ENQ
0	1	0	1	5	5	5	NAK
0	1	1	0	6	6	6	%
0	1	1	0	6	6	6	ACK
0	1	1	0	6	6	6	SYN
0	1	1	1	7	7	7	BEL
0	1	1	1	7	7	7	ETB
1	0	0	0	8	8	8	BS
1	0	0	0	8	8	8	CAN
1	0	0	1	9	9	9	HT
1	0	0	1	9	9	9	EM
1	0	1	0	10	10	10	LF
1	0	1	0	10	10	10	SUB
1	0	1	1	11	11	11	VT
1	0	1	1	11	11	11	ESC
1	1	0	0	12	12	12	FF
1	1	0	0	12	12	12	FS
1	1	0	1	13	13	13	CR
1	1	0	1	13	13	13	GS
1	1	1	0	14	14	14	SO
1	1	1	0	14	14	14	RS
1	1	1	1	15	15	15	SI
1	1	1	1	15	15	15	US

Abbildung 4.1: ASCII-Tabelle

4.1.2 ISO/IEC 8859-1

An dem ASCII-Standard gab es zwischendurch Aktualisierungen, sodass einige Kontrollzeichen entfernt wurden, doch in 7 Bit konnten nie alle länderspezifischen Zeichen untergebracht werden. Wir in Deutschland haben Umlaute, die Russen haben ein kyrillisches Alphabet, die Griechen Alpha und Beta und so weiter. Die Lösung war, statt einer 7-Bit-Kodierung, die 128 Zeichen unterbringen kann, einfach 8 Bit zu nehmen, womit 256 Zeichen kodiert werden können. Da in weiten Teilen der Welt das lateinische Alphabet genutzt wird, sollte diese Kodierung natürlich alle die Buchstaben zusammen mit einem Großteil aller diakritischen Zeichen (das sind etwa ü, á, à, ó, â, Å, Æ) umfassen. So setzte sich ein Standardisierungsgremium zusammen und schuf 1985 den *ISO/IEC 8859-1*-Standard, der 191 Zeichen beschreibt. Die Zeichen aus dem ASCII-Alphabet behalten ihre Positionen. Wegen der lateinischen Buchstaben hat sich die informelle Bezeichnung *Latin-1* als Alternative zu ISO/IEC 8859-1 etabliert.

Alle Zeichen aus ISO/IEC 8859-1 sind druckbar, sodass alle Kontrollzeichen – etwa der Tabulator oder das Zeilenumbruchzeichen – *nicht* dazu gehören. Von den 256 möglichen Positionen bleiben 65 Stellen frei. Das sind die Stellen 0 bis 31 sowie 127 von den ASCII-Kontrollzeichen und zusätzlich 128 bis 159.

ISO 8859-1

Da es kaum sinnvoll ist, den Platz zu vergeuden, gibt es eine Erweiterung des ISO/IEC 8859-1-Standards, die unter dem Namen *ISO 8859-1* (also ohne IEC) geläufig ist. ISO 8859-1 enthält alle Zeichen aus ISO/IEC 8859-1 sowie die Kontrollzeichen aus dem ASCII-Standard an den Positionen 0–31 und 127. Somit steckt ASCII vollständig in ISO 8859-1, aber nur die druckbaren ASCII-Zeichen sind in ISO/IEC 8859-1. Auch die Stellen 128 bis 159 sind in ISO 8859-1 definiert, wobei es alles recht unbekannte Kontrollzeichen (wie Padding, Start einer Selektion, kein Umbruch) sind.

Windows-1252 *

Weil die Zeichen an der Stelle 128 bis 159 uninteressante Kontrollzeichen sind, belegt Windows sie mit Buchstaben und Interpunktionszeichen und nennt die Kodierung *Windows-1252*. An Stelle 128 liegt etwa das €-Zeichen, an 153 das ™-Symbol. Diese Neubelebung der Plätze 128 bis 159 hat sich mittlerweile auch in der Nicht-Windows-Welt etabliert, sodass das, was im Web als ISO-8859-1 deklariert ist, heute die Symbole aus den Codepoints 128 bis 159 enthalten kann und von Browsern so dargestellt wird.

4.1.3 Unicode

Obwohl Latin-1 für die »großen« Sprachen alle Zeichen mitbrachte, fehlen Details, wie Ó, ö, Ÿ für das Ungarische, das komplette griechische Alphabet, die kyrillischen Buchstaben, chinesische und japanische Zeichen, mathematische Zeichen und vieles mehr. Um das Problem zu lösen, hat sich das Unicode-Konsortium gebildet, um jedes Zeichen der Welt zu kodieren und ihm einen eindeutigen Codepoint zu geben. Unicode enthält alle Zeichen aus ISO 8859-1, was die Konvertierung von Dokumenten vereinfacht. So behält zum Beispiel »A« den Codepoint 65 von ISO 8859-1, den der Buchstabe wiederum von ASCII erbt. Unicode ist aber viel mächtiger als ASCII oder Latin-1: Die letzte Version des Unicode-Standards 6 beschreibt über 100.000 Zeichen.

Wegen der vielen Zeichen ist es unpraktisch, diese dezimal anzugeben, sodass sich die hexadezimale Angabe durchgesetzt hat. Der Unicode-Standard nutzt das Präfix »U+«, gefolgt von Hexadezimalzahlen. Der Buchstabe »A« ist dann U+0041.

Unicode-Tabellen unter Windows *

Unter Windows legt Microsoft das nützliche Programm *charmap.exe* für eine Zeichentabelle bei, mit der jede Schriftart auf ihre installierten Zeichen untersucht werden kann. Praktischerweise zeigt die Zeichentabelle auch gleich die Position in der Unicode-Tabelle an.

Unter ERWEITERTE ANSICHT lassen sich mit GRUPPIEREN NACH in einem neuen Dialog Unicode-Unterbereiche auswählen, wie etwa Währungszeichen oder unterschiedliche Sprachen. Im Unterbereich LATIN finden sich zum Beispiel die Zeichen aus der französischen Schrift (etwa »Ç« mit Cedille unter 00c7) und der spanischen Schrift (»ñ« mit Tilde unter 00f1), und bei ALLG. INTERPUNKTIONSZEICHEN findet sich das umgedrehte (invertierte) Fragezeichen bei O0BF.



Abbildung 4.2: Zeichentabelle unter Windows Vista

Anzeige der Unicode-Zeichen *

Die Darstellung der Zeichen – besonders auf der Konsole – ist auf einigen Plattformen noch ein Problem. Die Unterstützung für die Standardzeichen des ASCII-Alphabets ist dabei weniger ein Problem als die Sonderzeichen, die der Unicode-Standard definiert. Ein Versuch, zum Beispiel den Smiley auf der Standardausgabe auszugeben, scheitert oft an der Fähigkeit des Terminals beziehungsweise der Shell. Hier ist eine spezielle Shell notwendig, die aber bei den meisten Systemen noch in der Entwicklung ist. Und auch bei grafischen Oberflächen ist die Integration noch mangelhaft. Es wird Aufgabe der Betriebssystementwickler bleiben, dies zu ändern.¹

Hinweis

Obwohl Java intern alle Zeichenfolgen in Unicode kodiert, ist es ungünstig, Klassennamen zu wählen, die Unicode-Zeichen enthalten. Einige Dateisysteme speichern die Namen im alten 8-Bit-ASCII-Zeichensatz ab, sodass Teile des Unicode-Zeichens verloren gehen.

4.1.4 Unicode-Zeichenkodierung

Da es im aktuellen Unicode-Standard 6.0 mehr als 109.000 Zeichen gibt, werden zur Kodierung eines Zeichens 4 Byte beziehungsweise 32 Bit verwendet. Ein Dokument, das Unicode 5.1-Zeichen enthält, wird dann einen Speicherbedarf von $4 \times \text{Anzahl der Zeichen}$ besitzen. Wenn die Zeichen auf diese Weise kodiert werden, sprechen wir von einer *UTF-32-Kodierung*.

Für die meisten Texte ist UTF-32 reine Verschwendug, denn besteht der Text aus nur einfachen ASCII-Zeichen, sind 3 Byte gleich 0. Gesucht ist eine Kodierung, die die allermeisten Texte kompakt kodieren kann, aber dennoch jedes der Unicode-5.1-Zeichen zulässt. Zwei Kodierungen sind üblich: *UTF-8* und *UTF-16*. UTF-8 kodiert ein Zeichen entweder in 1, 2, 3 oder 4 Byte, UTF-16 in 2 Byte oder 4 Byte. Das folgende Beispiel² zeigt die Kodierung für die Buchstaben »A« und »ß«, für das chinesische Zeichen für Osten und für ein Zeichen aus dem *Deseret*, einem phonetischen Alphabet.

1 Mit veränderten Dateiströmen lässt sich dies etwas in den Griff bekommen. So kann man beispielsweise mit einem speziellen `OutputStream`-Objekt eine Konvertierung für die Windows-NT-Shell vornehmen, sodass auch dort die Sonderzeichen erscheinen.

2 <http://java.sun.com/developer/technicalArticles/Intl/Supplementary/>

Glyph	A	ß	東	ð
Unicode-Code-point	U+0041	U+00DF	U+6771	U+10400
UTF-32	00000041	000000DF	00006771	00 01 04 00
UTF-16	00 41	00 DF	6771	D801 DC00
UTF-8	41	C3 9F	E6 9D B1	F0 90 90 80

Tabelle 4.1: Zeichenkodierung in den verschiedenen Unicode-Versionen

Werden Texte ausgetauscht, sind diese üblicherweise UTF-8 kodiert. Bei Webseiten ist das ein guter Standard. UTF-16 ist für Dokumente seltener, wird aber häufiger als interne Textrepräsentation genutzt. So verwenden zum Beispiel die JVM und die .NET-Laufzeitumgebung intern UTF-16.

4.1.5 Escape-Sequenzen/Fluchtsymbole

Um spezielle Zeichen, etwa den Zeilenumbruch oder Tabulator, in einen String oder `char` setzen zu können, stehen Escape-Sequenzen³ zur Verfügung.

Zeichen	Bedeutung
\b	Rückschritt (Backspace)
\n	Zeilenschaltung (Newline)
\f	Seitenumbruch (Formfeed)
\r	Wagenrücklauf (Carriage return)
\t	Horizontaler Tabulator
\"	Doppeltes Anführungszeichen
'	Einfaches Anführungszeichen
\\"	Backslash

Tabelle 4.2: Escape-Sequenzen

³ Nicht alle aus C stammenden Escape-Sequenzen finden sich auch in Java wieder. Es gibt kein '\a' (Alert), kein '\v' (vertikaler Tabulator) und kein '\?' (Fragezeichen).

zB

4

Beispiel

Zeichenvariablen mit Initialwerten und Sonderzeichen:

```
char theLetterA = 'a',
    singlequote = '\',
    newline      = '\n';
```

Die Fluchtsymbole sind für Zeichenketten die gleichen. Auch dort können bestimmte Zeichen mit Escape-Sequenzen dargestellt werden:

```
String s      = "Er fragte: \"Wer lispelt wie Katja Burkard?\"";
String filename = "C:\\Dokumente\\Siemens\\Schweigegeld.doc";
```

4.1.6 Schreibweise für Unicode-Zeichen und Unicode-Escapes

Da ein Java-Compiler alle Eingaben als Unicode verarbeitet, kann er grundsätzlich Quellcode mit deutschen Umlauten, griechischen Symbolen und chinesischen Schriftzeichen verarbeiten. Allerdings ist es gut, zu überlegen, ob ein Programm direkt Unicode-Zeichen enthalten sollte, denn Editoren haben mit Unicode-Zeichen oft ihre Schwierigkeiten – genauso wie Dateisysteme.

Beliebige Unicode-Zeichen lassen sich für den Compiler über *Unicode-Escapes* schreiben. Im Quellcode steht dann \uxxx, wobei x eine hexadezimale Ziffer ist – also 0...9, A...F (beziehungsweise a...f). Diese sechs ASCII-Zeichen, die das Unicode-Zeichen beschreiben, lassen sich in jedem ASCII-Texteditor schreiben, sodass kein Unicode-fähiger Editor nötig ist. Unicode-Zeichen für deutsche Sonderzeichen sind folgende:

Zeichen	Unicode
Ä, ä	\u00c4, \u00e4
Ö, ö	\u00d6, \u00f6
Ü, ü	\u00dc, \u00fc
ß	\u00df

Tabelle 4.3: Deutsche Sonderzeichen in Unicode

zB Beispiel

Deklariere und initialisiere eine Variable π :

```
double \u03c0 = 3.141592653589793;
```

Statt der herkömmlichen Buchstaben und Ziffern kann natürlich alles gleich in der \u-Schreibweise in den Editor gehackt werden, doch das ist nur dann sinnvoll, wenn Quellcode versteckt werden soll. Beim Compiler kommt intern alles als Unicode-Zeichenstrom an, egal ob wir in eine Datei

```
\u0063\u006c\u0061\u0073\u0073\u0020\u0041\u0020\u007b\u007d
```

setzen oder

```
class A {}
```

Die Unicode-Escape-Sequenzen sind also an beliebiger Stelle erlaubt, aber wirklich nützlich ist das eher für Zeichenketten. Der beliebte Smiley ☺ ist als Unicode unter \u263A (WHITE SMILING FACE) beziehungsweise unter \u2639 (WHITE FROWNING FACE) ☹ definiert. Das Euro-Zeichen € ist unter \u20ac zu finden.

zB Beispiel

Zeige Pi und in einem GUI-Dialog einen Grinsemann:

```
System.out.println( "Pi: \u03c0" ); // Pi: π
javax.swing.JOptionPane.showMessageDialog( null, "\u263a" );
```

Das Unicode-Zeichen \uffff ist nicht definiert und kann bei Zeichenketten als Endesymbol verwendet werden.

**Tipp**

Sofern sich die Sonderzeichen und Umlaute auf der Tastatur befinden, sollten keine Unicode-Kodierungen Verwendung finden. Der Autor von Quelltext sollte seine Leser nicht zwingen, eine Unicode-Tabelle zur Hand zu haben. Die Alternativdarstellung lohnt sich daher nur, wenn der Programmtext bewusst unleserlich gemacht werden soll. Bezeichner sollten in der Regel aber so gut wie immer in Englisch geschrieben werden, sodass höchstens Unicode-Escapes bei Zeichenketten vorkommen.

Enkodierung vom Quellcode festlegen *

Enthält Quellcode Nicht-ASCII-Zeichnen (wie Umlaute), dann ist das natürlich gegenüber der \u-Schreibweise ein Nachteil, denn so spielt das Dateiformat eine große Rolle. Es kann passieren, dass Quellcode in einer Zeichenkodierung abgespeichert wurde (etwa UTF-8), aber ein anderer Rechner eine andere Zeichenkodierung nutzt (vielleicht Latin-1) und den Java-Quellcode nun einlesen möchte. Das kann ein Problem werden, dann standardmäßig liest der Compiler den Quellcode in der Kodierung ein, die er selbst hat, denn an einer Textdatei ist die Kodierung nicht abzulesen. Wenn der Quellcode in einem anderen Format ist, wird er unbeabsichtigt vom Compiler falsch in das interne Unicode-Format konvertiert. Um das Problem zu lösen, gibt es zwei Ansätze:

- Dem *javac*-Compiler kann mit dem Schalter `-encoding` die Kodierung mitgegeben werden, in der der Quellcode vorliegt. Liegt etwa der Quellcode in UTF-8 vor, ist aber auf dem System mit dem Compiler die Kodierung Latin-1 eingestellt, dann muss für den Compiler der Schalter `-encoding` UTF-8 gesetzt sein. Details dazu gibt es unter <http://download.oracle.com/javase/7/docs/technotes/tools/windows/javac.html>. Wenn allerdings unterschiedliche Dateien in unterschiedlichen Formaten vorliegen, dann hilft die globale Einstellung nichts.
- Das Kommandozeilenprogramm *native2ascii* konvertiert Texte von beliebigen Kodierungen in Latin-1 und setzt Zeichen, die nicht Latin-1 sind, in \u-Folgen um. Zu den Argumenten gibt es mehr Informationen unter <http://download.oracle.com/javase/7/docs/technotes/tools/windows/native2ascii.html>.

4.1.7 Unicode 4.0 und Java *

In den letzten Jahren hat sich der Unicode-Standard erweitert, und Java ist den Erweiterungen gefolgt. Die Java-Versionen von 1.0 bis 1.4 nutzen den Unicode-Standard 1.1 bis 3.0, der für jedes Zeichen 16 Bit reserviert. So legt Java jedes Zeichen in 2 Byte ab und ermöglicht die Kodierung von mehr als 65.000 Zeichen. Ab Java 5 ist der Unicode 4.0-Standard möglich, der 32 Bit für die Abbildung eines Zeichens nötig macht. Java 7 unterstützt Unicode 6.0. Die Entwickler haben allerdings beim Wechsel auf Unicode 4 für ein Java-Zeichen nicht die interne Länge angehoben, sondern es bleibt dabei, dass ein `char` 2 Byte groß ist. Das heißt aber auch, dass ein Zeichen, das größer als 65.536 ist, irgendwie anders kodiert werden muss. Dazu muss ein »großes« Unicode-Zeichen durch zwei `char` zusammengesetzt werden. Dieses Pärchen aus zwei 16-Bit-Zeichen wird *Surrogate-Paar* genannt. Sie bilden in der *UTF-16-Kodierung* ein Unicode 4.0-Zeichen. Diese Surrogate vergrößern den Bereich der *Basic Multilingual Plane (BMP)*.

Unter Java 5 gab es an den String-Klassen einige Änderungen, sodass etwa eine Methode, die nach einem Zeichen sucht, nun nicht nur mit einem `char` parametrisiert ist, sondern auch mit `int`, und der Methode damit auch ein Surrogate-Paar übergeben werden kann. In diesem Buch spielt das aber keine Rolle, da Unicode-Zeichen aus dem höheren Bereichen, etwa für die phönizische Schrift, die im Unicode-Block U+10900 bis U+1091F liegt – also kurz hinter 65536, was durch 2 Byte abbildbar ist –, nur für eine ganz kleine Gruppe von Interessenten wichtig sind.



Entwicklerfrust

Die Abbildung eines Zeichens auf eine Position übernimmt eine Tabelle, die sich *Codepage* nennt. Nur gibt es unterschiedliche Abbildungen der Zeichen auf Positionen, und das führt zu Problemen beim Dokumentenaustausch. Denn wenn eine Codepage die Tilde »~« auf Position 161 setzt und eine andere Codepage das »ß« auch auf Position 161 anordnet, dann führt das zwangsläufig zu Ärgernissen. Daher muss beim Austausch von Textdokumenten immer ein Hinweis mitgegeben werden, in welchem Format die Texte vorliegen. Leider unterstützen die Betriebssysteme aber solche Meta-Angaben nicht, und so werden sie etwa in XML- oder HTML-Dokumenten in den Text selbst geschrieben. Bei Unicode-UTF-16-Dokumenten gibt es eine andere Konvention, sodass sie mit den Hexwert 0xFEFF beginnen – das wird *BOM (Byte Order Mark)* genannt und dient gleichzeitig als Indikator für die Byte-Reihenfolge.

4.2 Die Character-Klasse

Die im Kernpaket `java.lang` angesiedelte Klasse `Character` bietet eine große Anzahl statischer Methoden, die im Umgang mit einzelnen Zeichen interessant sind. Dazu gehören Methoden zum Testen, etwa ob ein Zeichen eine Ziffer, ein Buchstabe oder ein Sonderzeichen ist.

4.2.1 Ist das so?

Allen Testmethoden ist gemeinsam, dass sie mit der Vorsilbe `is` beginnen und ein `boolean` liefern. Dazu gesellen sich Methoden zum Konvertieren, etwa in Groß-/Kleinschreibung.

Ein paar Beispiele:

Ausdruck	Ergebnis
Character.isDigit('0')	true
Character.isDigit('-')	false
Character.isLetter('ß')	true
Character.isLetter('0')	false
Character.isWhitespace(' ')	true
Character.isWhitespace('-')	false

Tabelle 4.4: Ergebnis einiger isXXX()-Methoden

Alle diese Methoden »wissen« über die Eigenschaften der einzelnen Unicode-Zeichen Bescheid. Und 0 bleibt ja immer eine Null, egal ob das Programm in Deutschland oder in der Mongolei ausgeführt wird, denn der Codepoint jedes Unicode-Zeichens ist immer der gleiche.

Testen, ob eine Zeichenkette nur aus Ziffern besteht

Im folgenden Beispiel wollen wir die Methode deklarieren, die einen String abläuft und testet, ob der String nur aus Ziffern besteht. Obwohl so eine Funktionalität in der Praxis nützlich ist, bietet Java SE dafür keine simple Methode.

Listing 4.1: IsNumeric.java

```
public class IsNumeric
{
    /**
     * Returns {@code true} if the String contains only Unicode digits.
     * An empty string or {@code null} leads to {@code false}.
     *
     * @param string Input String.
     * @return {@code true} if string is numeric, {@code false} otherwise.
     */
    public static boolean isNumeric( String string )
    {
        if ( string == null || string.length() == 0 )
            return false;
```

```

for ( int i = 0; i < string.length(); i++ )
    if ( ! Character.isDigit( string.charAt( i ) ) )
        return false;

    return true;
}

public static void main( String[] args )
{
    System.out.println( isNumeric( "1234" ) ); // true
    System.out.println( isNumeric( "12.4" ) ); // false
    System.out.println( isNumeric( "-123" ) ); // false
}
}

```

Es ist so definiert, dass `null` und ein leerer String nicht als numerisch angesehen werden, allerdings lässt sich auch definieren, dass `null` zu einer Ausnahme führen soll und der leere String durchaus numerisch ist. Konventionen wie diese liegen beim Autor der Bibliothek, und unterschiedliche Utility-Bibliotheken mit solchen Hilfsfunktionen haben dort unterschiedliche Vorstellungen.

Das Beispiel nutzt zwei String-Methoden: `length()` liefert die Länge eines Strings, und `charAt()` liefert das Zeichen an der gewünschten Stelle. Eine Schleife iteriert über den String und testet jedes Zeichen mit `isDigit()`; ist ein Zeichen keine Ziffer, verlässt `return false` automatisch die Schleife. Läuft die Schleife erfolgreich durch, kann ein `return true` vermelden, dass jedes Zeichen eine Ziffer war.

Die wichtigsten `isXXX()`-Methoden im Überblick

```

final class java.lang.Character
    implements Serializable, Comparable<Character>

```

- `static boolean isDigit(char ch)`
Handelt es sich um eine Ziffer zwischen 0 und 9?
- `static boolean isLetter(char ch)`
Handelt es sich um einen Buchstaben?
- `static boolean isLetterOrDigit(char ch)`
Handelt es sich um ein alphanumerisches Zeichen?

- static boolean isLowerCase(char ch)
- static boolean isUpperCase(char ch)

Ist es ein Klein- oder ein Großbuchstabe?
- static boolean isWhiteSpace(char ch)

Ist es ein Leerzeichen, Zeilenvorschub, Return oder Tabulator, also ein sogenannter *Weißraum*⁴ (engl. *white space*), auch *Leerraum* genannt?
- static boolean isJavaIdentifierStart(char ch)

Ist es ein Java-Buchstabe, mit dem Bezeichner beginnen dürfen?
- static boolean isJavaIdentifierPart(char ch)

Ist es ein Java-Buchstabe oder eine Ziffer, der beziehungsweise die in der Mitte eines Bezeichners vorkommen darf?
- static boolean isTitleCase(char ch)

Sind es spezielle Zwei-Buchstaben-Paare mit gemischter Groß- und Kleinschreibung? Dies kommt etwa im Spanischen vor, wo »lj« für einen einzigen Buchstaben steht. In Überschriften erscheint dieses Paar als »Lj« und wird von dieser Methode als Sonderfall erkannt. Unter <http://www.unicode.org/reports/tr21/tr21-5.html> schreibt der Unicode-Standard die Konvertierung vor.

4.2.2 Zeichen in Großbuchstaben/Kleinbuchstaben konvertieren

Zum Konvertieren eines Zeichens in Groß-/Kleinbuchstaben deklariert die Character Klasse die Methoden `toUpperCase()` und `toLowerCase()`. Die testenden `isXXX()`-Methoden finden oft Anwendung beim Ablaufen einer Zeichenkette.

Unser nächstes Beispiel fragt den Benutzer nach einem String. Gültige Buchstaben sollen in Großbuchstaben konvertiert werden, und jeder Weißraum soll durch einen Unterstrich ersetzt werden. Zum Ablaufen der Eingabe nutzen wir wieder die schon bekannten String-Methoden `length()` und `charAt()`:

Listing 4.2: UppercaseWriter.java

```
public class UppercaseWriter
{
    public static void main( String[] args )
    {
```

⁴ Es wird Weißraum genannt, weil das ausgegebene Zeichen den Raum in der Regel weiß lässt, aber die Position der Ausgabe dennoch fortschreitet.

```

String input = new java.util.Scanner( System.in ).nextLine();

for ( int i = 0; i < input.length(); i++ )
{
    char c = input.charAt( i );
    if ( Character.isWhitespace( c ) )
        System.out.print( ' ' );
    else if ( Character.isLetter( c ) )
        System.out.print( Character.toUpperCase( c ) );
}
}
}

```

Wenn die Eingabe etwa »honiara brotherhood guesthouse1« ist, ist die Ausgabe »HONIARA_BROTHERHOOD_GUESTHOUSE«. Die »1« verschwindet, weil sie weder ein Leerzeichen noch ein Buchstabe ist.

```

final class java.lang.Character
    implements Serializable, Comparable<Character>

```

- static char toUpperCase(char ch)
- static char toLowerCase(char ch)

Die statischen Methoden `toUpperCase()` und `toLowerCase()` liefern den passenden Groß- beziehungsweise Kleinbuchstaben zurück.



Hinweis

Die Methoden `toUpperCase()` und `toLowerCase()` gibt es zweimal: einmal als statische Methoden bei `Character` – dann nehmen sie genau ein `char` als Argument – und einmal als Objektmethoden auf `String`-Exemplaren.

Vorsicht ist bei `Character.toUpperCase('ß')` geboten, denn das Ergebnis ist »ß«, anders als bei der `String`-Methode `toUpperCase("ß")`, die das Ergebnis »SS« liefert; einen String, der um eins verlängert ist.

4.2.3 Ziffern einer Basis *

Die Character-Klasse besitzt ebenso eine Umwandlungsmethode für Ziffern bezüglich einer beliebigen Basis:

```
final class java.lang.Character
    implements Serializable, Comparable<Character>
```

- static int digit(char ch, int radix)

Liefert den numerischen Wert, den das Zeichen `ch` unter der Basis `radix` besitzt. Beispielsweise ist `Character.digit('f', 16)` gleich 15. Erlaubt ist jedes Zahlensystem mit einer Basis zwischen `Character.MIN_RADIX` (2) und `Character.MAX_RADIX` (36). Ist keine Umwandlung möglich, beträgt der Rückgabewert -1.

- static char forDigit(int digit, int radix)

Konvertiert einen numerischen Wert in ein Zeichen. Beispielsweise ist `Character.forDigit(6, 8)` gleich »6« und `Character.forDigit(12, 16)` ist »c«.

Es ist bedauerlich, dass der Radix immer mit angegeben werden muss, obwohl er in der Regel immer 10 ist. Eine überladene statische Methode wäre hier angebracht.

Beispiel

zB

Steht in einem Zeichen `c` zum Beispiel '3' und soll aus diesem die Ganzzahl 3 werden, so besteht die traditionelle Art darin, eine '0' abzuziehen. Die ASCII-Null '0' hat den char-Wert 48, '1' dann 49, bis '9' schließlich 57 erreicht. So ist logischerweise $'3' - '0' = 51 - 48 = 3$.) Die `digit()`-Methode ist dazu eine Alternative. Wir nutzen sie in einem kleinen Beispiel, um eine Zeichenkette mit Ziffern schließlich in eine Ganzzahl zu konvertieren.

```
char[] chars = { '3', '4', '0' };
int result = 0;
for ( char c : chars )
{
    result = result * 10 + Character.digit( c, 10 );
    System.out.println( result );
}
```

Die Ausgabe ist 3, 34 und 340.



Abbildung 4.3: UML-Diagramm der umfangreichen Klasse Character

4.3 Zeichenfolgen

Ein String ist eine Sammlung von Zeichen (Datentyp `char`), die die Laufzeitumgebung geordnet im Speicher ablegt. Die Zeichen sind einem Zeichensatz entnommen, der in Java dem 16-Bit-Unicode-Standard entspricht – mit einigen Umwegen ist auch Unicode 4 mit 32-Bit-Zeichen möglich.

Java sieht drei Klassen vor, die Zeichenfolgen verwalten. Sie unterscheiden sich in drei Punkten:

- Sind die Zeichenfolgen unveränderbar (*immutable*), können sie also später nicht mehr verändert werden?
- Sind die Zeichenfolgen später veränderbar (*mutable*)?
- Sind die Operationen auf den Zeichenketten gegen nebenläufige Zugriffe aus mehreren Threads abgesichert?

	Verwaltet immutable Zeichenketten	Verwaltet mutable Zeichenketten
Threadsicher	<code>String</code>	<code>StringBuffer</code>
Nicht threadsicher	-	<code>StringBuilder</code>

Tabelle 4.5: Die drei Klassen, die Zeichenfolgen verwalten

Die Klasse String

Die Klasse `String` repräsentiert nicht änderbare Zeichenketten. (Allgemein heißen Objekte, deren Zustand sich nicht verändert lässt, *immutable*.) Daher ist ein `String` immer threadsicher, denn eine Synchronisation ist nur dann nötig, wenn es Änderungen geben kann. Mit Objekten vom Typ `String` lässt sich nach Zeichen oder Teilzeichenketten suchen, und ein `String` lässt sich mit einem anderen `String` vergleichen, aber Zeichen im `String` können nicht verändert werden. Es gibt einige Methoden, die scheinbar Veränderungen an `String`s vornehmen, aber sie erzeugen in Wahrheit neue `String`-Objekte, die die veränderten Zeichenreihen repräsentieren. So entsteht beim Aneinanderhängen zweier `String`-Objekte als Ergebnis ein drittes `String`-Objekt für die zusammengefügte Zeichenreihe.

Die Klassen `StringBuilder`/`StringBuffer`

Die Klassen `StringBuilder` und `StringBuffer` repräsentieren im Gegensatz zu `String` dynamische, beliebig änderbare Zeichenreihen. Der Unterschied zwischen den API-gleichen Klassen ist lediglich, dass `StringBuffer` vor nebenläufigen Operationen geschützt ist, `StringBuilder` nicht. Die Unterscheidung ist bei `Strings` nicht nötig, denn wenn Objekte nachträglich nicht veränderbar sind, machen parallele Lesezugriffe keine Schwierigkeiten.

zB Beispiel

`String`-Objekte selbst lassen sich nicht verändern, aber natürlich lässt sich eine Referenz auf ein anderes `String`-Objekt setzen:

```
String s = "tutego";
s = "TUTEGO";
```

Mit »verändern« meinen wir hier, dass der Zustand eines Objekts verändert wird, etwa indem das erste Zeichen gelöscht wird. Die Referenz umzubiegen verändert keinen Zustand.

Der Basistyp `CharSequence` für Zeichenketten

`CharSequence` ist die gemeinsame Schnittstelle von `String`, `StringBuilder` und `StringBuffer` und wird in der Bibliothk mehrfach verwendet. Nehmen wir ein Beispiel: Die Klasse `String` deklariert eine Methode `contains(CharSequence s)`, die testet, ob der Teilstring `s` im `String` vorkommt. Von welchem Typ kann nun die Variable `s` sein? Wir können Exemplare etwa von `String`, `StringBuilder` oder `StringBuffer` übergeben, weil dies alles `CharSequences` sind. Wir steigen später noch etwas genauer in den Typ ein, an dieser Stelle reicht es zu wissen, dass wir uns überall, wo `CharSequence` steht, `String`, `StringBuilder` oder `StringBuffer` denken können.

Bestehen `Strings` aus `char`-Feldern?

Die drei Klassen `String`, `StringBuilder`, `StringBuffer` entsprechen der idealen Umsetzung der objektorientierten Idee (mit der wir uns in Kapitel 5, »Eigene Klassen schreiben«, intensiv auseinandersetzen): Wie genau die Zeichenfolgen gespeichert werden, dringt nicht nach außen. Zwar ist es so, dass die Zeichenfolgen intern als `char`-Arrays abgebildet werden, aber dieses Feld ist privat und somit sind keine Zugriffe von außen möglich. Selbst die Länge ist ein privates Attribut der Klassen, die nur über eine Methode zugängig

lich ist. Das Schöne ist also, dass die Klassen uns also die lästige Arbeit abnehmen, selbst Zeichenfolgen in Feldern verwalten zu müssen.

4.4 Die Klasse String und ihre Methoden

Die Entwickler von Java haben eine Symbiose zwischen `String` als Klasse und dem `String` als eingebautem Datentyp vorgenommen. Die Sonderbehandlung gegenüber anderen Objekten ist an zwei Punkten abzulesen:

- Die Sprache ermöglicht die direkte Konstruktion von String-Objekten aus String-Literalen (Zeichenketten in doppelten Anführungszeichen).
- Die Konkatenation (Aneinanderreihung von Strings mit `+`) von mehreren Strings ist erlaubt, aber Plus ist für keinen anderen Objekttyp erlaubt. Es lassen sich zum Beispiel nicht zwei `Point`-Objekte addieren. Mit dem Plus auf String-Objekten ist also ein besonderer Operator auf der Klasse `String` definiert, der nicht eigenständig auf anderen Klassen definiert werden kann. Java unterstützt keine überladenen Operatoren für Klassen, und dieses Plus ist ein Abweichler.

4.4.1 String-Literale als String-Objekte für konstante Zeichenketten

Damit wir Zeichenketten nutzen können, muss ein Objekt der Klasse `String` vorliegen. Das Schöne ist, dass alles in doppelten Anführungszeichen schon automatisch ein String-Objekt ist. Das bedeutet auch, dass hinter dem String-Literal gleich ein Punkt für den Methodenaufruf stehen kann.

Beispiel

zB

`"Hi Chris".length()` liefert die Länge der Zeichenkette. Das Ergebnis ist 8. Leerzeichen und Sonderzeichen zählen mit.

Nur Zeichenfolgen in doppelten Anführungszeichen sind String-Literale und somit schon gleich vorkonstruierte Objekte. Das gilt für `StringBuilder/StringBuffer` nicht – sie müssen von Hand mit `new` erzeugt werden. Nutzen wir String-Literale, so sollten wir ausdrücklich davon absehen, String-Objekte mit `new` zu erzeugen; ein `s = new("String")` ist unsinnig.

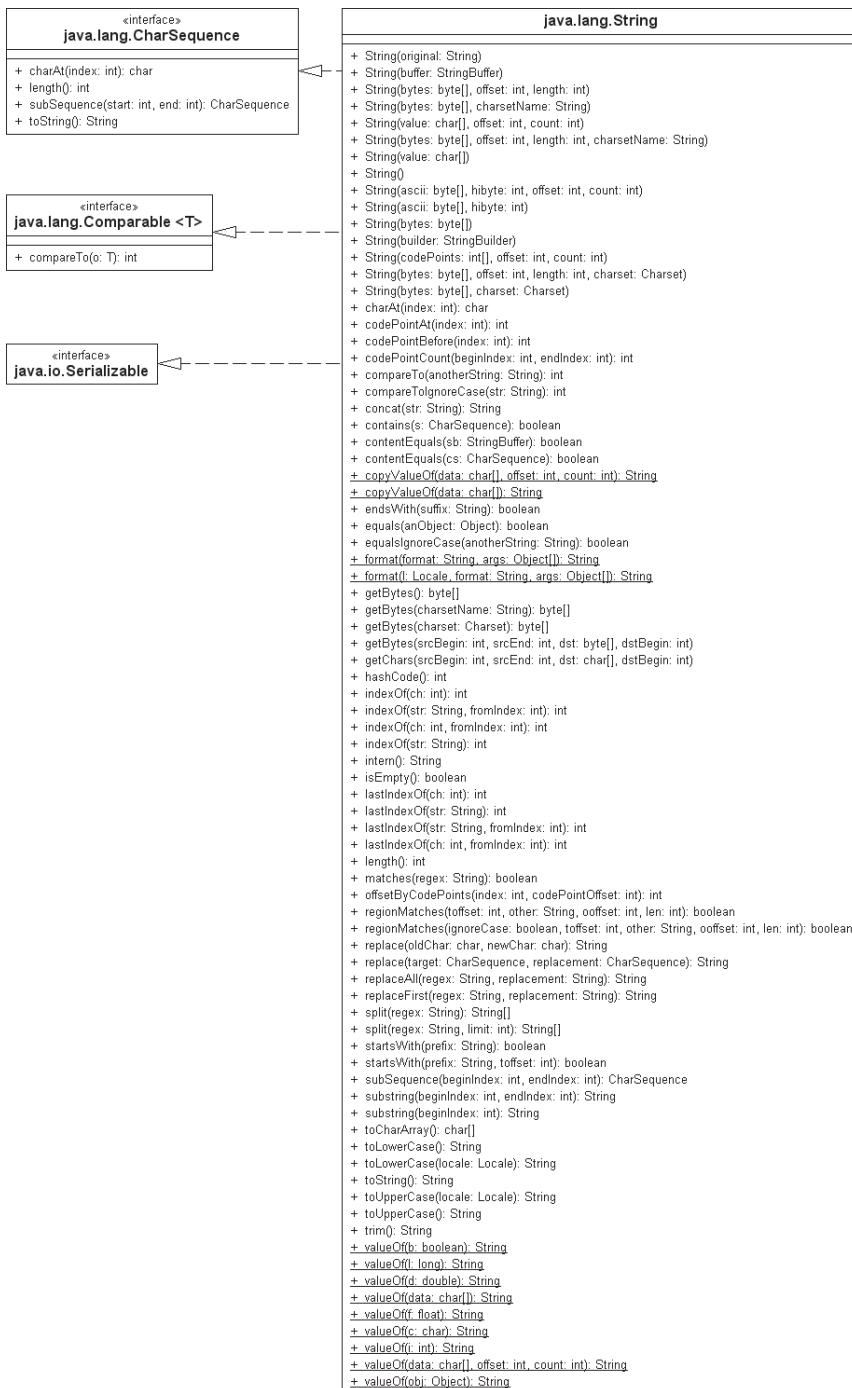


Abbildung 4.4: UML-Diagramm der umfangreichen Klasse String

4.4.2 Konkatenation mit +

Mehrere Beispiele haben schon gezeigt, dass Strings mit + konkateniert werden können.

Beispiel

zB

Haben die Glieder bei der Konkatenation unterschiedliche Datentypen, und einer ist String, werden diese automatisch auf String gebracht.

```
int    age    = 37;
double height = 1.83;
String s = "Alter: " + age + ", Größe: " + height;
System.out.println( s );      // Alter: 37, Größe: 1.83
```

Hinweis



Intern ist die Aneinanderreihung über Methoden der Klassen String, StringBuilder beziehungsweise StringBuffer realisiert. Die Konkatenation über + ist insbesondere in Schleifen nicht performant, und später werden wir im Abschnitt 4.6.4 bessere Lösungen kennenlernen.

4.4.3 String-Länge und Test auf Leerstring

String-Objekte verwalten intern die Zeichenreihe, die sie repräsentieren, und bieten eine Vielzahl von Methoden, um die Eigenschaften des Objekts preiszugeben. Eine Methode haben wir schon benutzt: `length()`. Für String-Objekte ist diese so implementiert, dass die Anzahl der Zeichen im String (die Länge des Strings) zurückgegeben wird. Um herauszufinden, ob der String keine Zeichen hat, lässt sich neben `length() == 0` auch die Methode `isEmpty()` nutzen:

Anweisung	Ergebnis
<code>"".length()</code>	0
<code>"".isEmpty()</code>	true
<code>" ".length()</code>	1
<code>" ".isEmpty()</code>	false
<code>String s = null; s.length();</code>	NullPointerException

Tabelle 4.6: Ergebnisse der Methoden `length()` und `isEmpty()` bei unterschiedlichen Strings

Eine praktische Hilfsmethode: `isNullOrEmpty()`

Während das .NET-Framework etwa die statische Member-Funktion `IsNullOrEmpty(String)` anbietet, um zu testen, ob die übergebene Referenz `null` oder die Zeichenkette leer ist, so muss das in Java getrennt getestet werden. Hier ist eine eigene statische Utility-Methode praktisch:

Listing 4.3: `LengthAndEmptyDemo.java`, `isNullOrEmpty()`

```
/*
 * Checks if a String is {@code null} or empty ({@code ""}).
 *
 * <pre>
 * StringUtils.isEmpty(null) == true
 * StringUtils.isEmpty("") == true
 * StringUtils.isEmpty(" ") == false
 * StringUtils.isEmpty("bob") == false
 * StringUtils.isEmpty(" bob ") == false
 * </pre>
 *
 * @param str The String to check, may be {@code null}.
 * @return {@code true} if the String is empty or {@code null}, {@code false}
 * otherwise.
 */
public static boolean isNullOrEmpty( String str )
{
    return str == null || str.length() == 0;
}
```

Ob der String nur aus Leerzeichen besteht, testet die Methode nicht.

4.4.4 Zugriff auf ein bestimmtes Zeichen mit `charAt()`

Die vielleicht wichtigste Methode der Klasse `String` ist `charAt(int index)`. Diese Methode liefert das entsprechende Zeichen an einer Stelle, die »Index« genannt wird. Dies bietet eine Möglichkeit, die Zeichen eines Strings (zusammen mit der Methode `length()`) zu durchlaufen. Ist der Index kleiner null oder größer beziehungsweise gleich der Anzahl

der Zeichen im String, so löst die Methode eine `StringIndexOutOfBoundsException`⁵ mit der Fehlerstelle aus.

Beispiel

zB

Liefere das erste und letzte Zeichen im String s:

```
String s = "Ich bin nicht dick! Ich habe nur weiche Formen.";
char first = s.charAt( 0 );                                // 'I'
char last  = s.charAt( s.length() - 1 );                  // .'
```

4

Wir müssen bedenken, dass die Zählung wieder bei null beginnt. Daher müssen wir von der Länge des Strings eine Stelle abziehen. Da der Vergleich auf den korrekten Bereich bei jedem Zugriff auf `charAt()` stattfindet, ist zu überlegen, ob der String bei mehrmaligem Zugriff nicht stattdessen einmalig in ein eigenes Zeichen-Array kopiert werden sollte.

4.4.5 Nach enthaltenen Zeichen und Zeichenfolgen suchen

Die Objektmethode `contains(CharSequence)` testet, ob ein *Teilstring* (engl. *substring*) in der Zeichenkette vorkommt, und liefert `true`, wenn das der Fall ist. Groß-/Kleinschreibung ist relevant. Im nächsten Programm wollen wir testen, ob eine E-Mail mögliche Spam-Wörter enthält.

Listing 4.4: EmailSpamChecker.java

```
public class EMailSpamChecker
{
    public static void main( String[] args )
    {
        String email1 = "Hallo Chris,...";
        System.out.println( containsSpam( email1 ) ); // false
        String email2 = "Kaufe Viagra! Noch billiga und macht noch härta!";
        System.out.println( containsSpam( email2 ) ); // true
    }
}
```

⁵ Mit 31 Zeichen gehört dieser Klassenname schon zu den längsten. Übertroffen wird er aber noch um fünf Zeichen von `TransformerFactoryConfigurationError`. Im Spring-Paket gibt es aber `JdbcUpdateAffectedIncorrectNumberOfRowsException` – auch nicht von schlechten Eltern.

```
public static boolean containsSpam( String text )
{
    return text.contains( "Viagra" ) || text.contains( "Ding-Dong-Verlängerung" );
}
```

Fundstelle mit indexOf() zurückgeben

Die Methode `contains()` ist *nicht* mit einem `char` überladen, kann also nicht nach einem einzelnen Zeichen suchen, es sei denn, der String bestünde nur aus einem Zeichen. Dazu ist `indexOf()` in der Lage: Die Methode liefert die Fundstelle eines Zeichens beziehungsweise Teilstrings. Findet `indexOf()` nichts, liefert sie `-1`.

zB Beispiel

Ein Zeichen mit `indexOf()` suchen:

```
String s = "Ernest Gräfenberg";
int index1 = s.indexOf( 'e' );                      // 3
int index2 = s.indexOf( 'e', index1 + 1 );          // 11
```

Die Belegung von `index1` ist 3, da an der Position 3 das erste Mal ein 'e' vorkommt. Die zweite Methode `indexOf()` sucht mit dem zweiten Ausdruck `index1 + 1` ab der Stelle 4 weiter. Das Resultat ist 11.

Wie `contains()` unterscheidet die Suche zwischen Groß- und Kleinschreibung. Die Zeichen in einem String sind wie Array-Elemente ab 0 durchnummieriert. Ist das Index-Argument kleiner 0, so wird dies ignoriert und der Index automatisch auf 0 gesetzt.

zB Beispiel

Beschreibt das Zeichen c ein Escape-Zeichen, etwa einen Tabulator oder ein Return, dann soll die Bearbeitung weitergeführt werden:

```
if ( "\b\t\n\f\r\"\\".indexOf(c) >= 0 ) {
    ...
}
```

`contains()` konnten wir nicht verwenden, da der ParameterTyp nur `CharSequence`, aber kein `char` ist.

Die `indexOf()`-Methode ist nicht nur mit `char` parametrisiert, sondern auch mit `String`⁶, um nach ganzen Zeichenfolgen zu suchen und die Startposition zurückzugeben.

Beispiel

zB

`indexOf()` mit der Suche nach einem Teilstring:

```
String str = "In Deutschland gibt es immer noch ein Ruhrgebiet, " +
            "obwohl es diese Krankheit schon lange nicht mehr geben soll.";
String s = "es";
int index = str.indexOf( s, str.indexOf(s) + 1 );           // 57
```

Die nächste Suchposition wird ausgehend von der alten Finderposition errechnet. Das Ergebnis ist 57, da dort zum zweiten Mal das Wort »es« auftaucht.

4

Vom Ende an suchen

Genauso wie am Anfang gesucht werden kann, ist es auch möglich, am Ende zu beginnen.

Beispiel

zB

Hierzu dient die Methode `lastIndexOf()`:

```
String str = "May the Force be with you.";
int index = str.lastIndexOf( 'o' );                  // 23
```

Genauso wie bei `indexOf()` existiert eine überladene Version, die rückwärts ab einer bestimmten Stelle nach dem nächsten Vorkommen von »o« sucht. Wir schreiben:

```
index = str.lastIndexOf( 'o', index - 1 );          // 9
```

Die Parameter der char-orientierten Methoden `indexOf()` und `lastIndexOf()` sind alle vom Typ `int` und nicht, wie man spontan erwarten könnte, vom Typ `char` und `int`. Das

⁶ Der Parametertyp `String` erlaubt natürlich nur Objekte vom Typ `String`, und Unterklassen von `String` gibt es nicht. Allerdings gibt es andere Klassen in Java, die Zeichenfolgen beschreiben, etwa `StringBuilder` oder `StringBuffer`. Diese Typen unterstützt die `indexOf()`-Methode nicht. Das ist schade, denn `indexOf()` hätte statt `String` durchaus einen allgemeineren Typ `CharSequence` erwarten können, um den `String` sowie `StringBuilder/StringBuffer` zu implementieren (zu dieser Schnittstelle in Abschnitt 4.6) mehr.

zu suchende Zeichen wird als erstes int-Argument übergeben. Die Umwandlung des char in ein int nimmt der Java-Compiler automatisch vor, sodass dies nicht weiter auffällt. Bedauerlicherweise kann es dadurch aber zu Verwechslungen bei der Reihenfolge der Argumente kommen: Bei `s.indexOf(start, c)` wird der erste Parameter `start` als Zeichen interpretiert und das gewünschte Zeichen `c` als Startposition der Suche.

Anzahl der Teilstrings einer Zeichenkette *

Bisher bietet die Java-Bibliothek keine direkte Methode, um die Anzahl der Teilstrings einer Zeichenkette herauszufinden. Eine solche Methode ist jedoch schnell geschrieben:

Listing 4.5: CountMatches.java

```
public class CountMatches
{
    public static int frequency( String source, String part )
    {
        if ( source == null || source.isEmpty() || part == null || part.isEmpty() )
            return 0;

        int count = 0;

        for ( int pos = 0; (pos = source.indexOf( part, pos )) != -1; count++ )
            pos += part.length();

        return count;
    }

    public static void main( String[] args )
    {
        System.out.println( frequency( "schlingelschlangel", "sch" ) ); // 2
        System.out.println( frequency( "schlingelschlangel", "ing" ) ); // 1
        System.out.println( frequency( "schlingelschlangel", "" ) ); // 0
    }
}
```

4.4.6 Das Hangman-Spiel

Geschätzte 90 Prozent aller Praxisaufgaben lösen die Methoden, die wir bisher schon kennengelernt haben:

- int charAt(int)
- int length()
- boolean equals(Object)
- boolean contains(CharSequence)
- int indexOf(char), int indexOf(String)

Genau die Methoden wollen wir für ein kleines Spiel nutzen, das berühmte Hangman-Spiel. Hierbei geht es darum, alle Buchstaben eines Wortes zu raten. Am Anfang ist jeder Buchstabe durch einen Unterstrich unkenntlich gemacht. Der Benutzer fängt an zu raten und füllt nach und nach die einzelnen Platzhalter aus. Gelingt es dem Spieler nicht, nach einer festen Anzahl von Runden das Wort zu erraten, hat er verloren.

Listing 4.6: Hangman1.java

```
public class Hangman1
{
    public static void main( String[] args )
    {
        String hangmanWord = "alligatoralley";
        String usedChars = "";

        String guessedWord = "";
        for ( int i = 0; i < hangmanWord.length(); i++ )
            guessedWord += "_";

        for ( int guesses = 1; ; )
        {
            if ( guesses == 10 )
            {
                System.out.printf( "Nach 10 Versuchen ist jetzt Schluss. Sorry! "+
                    "Apropos, das Wort war '%s'.", hangmanWord );
                break;
            }
        }
    }
}
```

```
System.out.printf( "Runde %d. Bisher geraten: %s. Was wählst du für ein
Zeichen?%n", guesses, guessedWord );
char c = new java.util.Scanner( System.in ).next().charAt( 0 );
if ( usedChars.indexOf( c ) >= 0 )
{
    System.out.printf( "%c hast du schon mal getippt!%n", c );
    guesses++;
}
else // Zeichen wurde noch nicht benutzt
{
    usedChars += c;
    if ( hangmanWord.indexOf( c ) >= 0 )
    {
        guessedWord = "";
        for ( int i = 0; i < hangmanWord.length(); i++ )
            guessedWord += usedChars.indexOf( hangmanWord.charAt( i ) ) >= 0 ?
                hangmanWord.charAt( i ) : "_";

        if ( guessedWord.contains( "_" ) )
            System.out.printf( "Gut geraten, '%s' gibt es im Wort. " +
                "Aber es fehlt noch was!%n", c );
    }
    else
    {
        System.out.printf( "Gratulation, du hast das Wort '%s' erraten!",
            hangmanWord );
        break;
    }
}
else // hangmanWord.indexOf( c ) == -1
{
    System.out.printf( "Pech gehabt, %c kommt im Wort nicht vor!%n", c );
    guesses++;
}
}
```

4.4.7 Gut, dass wir verglichen haben

Um Strings zu vergleichen, gibt es viele Möglichkeiten und Optionen:

- Die Methode `equals()` der Klasse `String` achtet auf absolute Übereinstimmung.
- Die Methode `equalsIgnoreCase()` der Klasse `String` ist für einen Vergleich zu haben, der unabhängig von der Groß-/Kleinschreibung ist.
- Seit Java 7 erlaubt `switch` den Vergleich von `String`-Objekten mit einer Liste von Sprungzielen. Der Vergleich wird intern mit `equals()` durchgeführt.
- Ob ein `String` mit einem Wort beginnt oder endet, sagen die `String`-Methoden `startsWith()` und `endsWith()`.
- Zum Vergleichen von Teilen gibt es die `String`-Methode `regionMatches()`, eine Methode, die auch unabhängig von der Groß-/Kleinschreibung arbeiten kann.
- Ist eine Übereinstimmung mit einem regulären Ausdruck gewünscht, helfen die Methode `matches()` von `String` sowie die speziellen Klassen `Pattern` und `Matcher`, die speziell für reguläre Ausdrücke sind.

Hinweis

Während die allermeisten Skript-Sprachen und auch C# Zeichenkettenvergleiche mit `==` erlauben, ist die Semantik für Java immer eindeutig: Der Vergleich mit `==` ist nur dann wahr, wenn die beiden Referenzen gleich sind, also zwei `String`-Objekte identisch sind; die Gleichheit reicht nicht aus.



Die Methode `equals()`

Die Klasse `String` überschreibt die aus der Klasse `Object` geerbte Methode `equals()`, um zwei `Strings` vergleichen zu können. Die Methode gibt `true` zurück, falls die `Strings` gleich lang sind und Zeichen für Zeichen übereinstimmen.

Beispiel

zB

Bei dem Vergleich mit `==` ist das Ergebnis ein anderes als mit `equals()`:

```
String input = javax.swing.JOptionPane.showInputDialog( "Passwort" );
System.out.println( input == "heinzelmann" );           // (1)
System.out.println( input.equals( "heinzelmann" ) );     // (2.1)
System.out.println( "heinzelmann".equals( input ) );    // (2.2)
```

zB Beispiel (Forts.)

Unter der Annahme, dass `input` die Zeichenkette »heinzelmann« referenziert, ergibt der Vergleich (1) über `==` den Wert `false`, da das von `showInputDialog()` gelieferte String-Objekt ein ganz anderes ist als das, was uns die virtuelle Maschine für den Test bereitstellt (später dazu mehr). Nur der `equals()`-Vergleich (2.1) und (2.2) ist hier korrekt, da hier die puren Zeichen verglichen werden, und die sind dann gleich.

Grundsätzlich sind Variante (2.1) und (2.2) gleich, da `equals()` symmetrisch ist. Doch gibt es einen Vorteil bei (2.2), denn da kann `input` auch `null` sein, und es gibt nicht wie bei (2.1) eine `NullPointerException`.

**Hinweis**

Beim `equals()`-Vergleich spielen alle Zeichen eine Rolle, auch wenn sie nicht sichtbar sind. So führen folgende Vergleiche zu `false`:

```
System.out.println( "\t".equals( "\n" ) );           // false
System.out.println( "\t".equals( "\t" ) );           // false
System.out.println( "\u0000".equals( "\u0000\u0000" ) ); // false
```

Die Methode `equalsIgnoreCase()`

`equals()` beachtet beim Vergleich die Groß- und Kleinschreibung. Mit `equalsIgnoreCase()` bietet die Java-Bibliothek eine zusätzliche Methode, um Zeichenketten ohne Beachtung der Groß-/Kleinschreibung zu vergleichen.

zB**Beispiel**

```
String str = "REISEPASS";
boolean result1 = str.equals( "Reisepass" );           // false
boolean result2 = str.equalsIgnoreCase( "ReISePaSs" ); // true
```

Eine kleine Anmerkung noch: Die Implementierung von `equalsIgnoreCase()` basiert intern darauf, beide Zeichenfolgen Zeichen für Zeichen abzulaufen, dabei einzelne Zeichen in Großbuchstaben zu konvertieren und dann zu prüfen, ob die beiden Großbuchstaben gleich sind.

Methodenvergleich

Der Vergleich "naß".toUpperCase().equals("NASS".toUpperCase()) beziehungsweise "NASS".toUpperCase().equals("naß".toUpperCase()) ergibt in beiden Fällen true. Doch "naß".equalsIgnoreCase("NASS") bzw. "NASS".equalsIgnoreCase("naß") ergeben false. Da Character.toUpperCase('ß') wieder ß ist, kann »naß« nicht »NASS« sein.



Lexikografische Vergleiche mit Größer/kleiner-Relation

Wie equals() und equalsIgnoreCase() vergleichen auch die Methoden compareTo(String) und compareToIgnoreCase(String) den aktuellen String mit einem anderen String. Nur ist der Rückgabewert von compareTo() kein boolean, sondern ein int. Das Ergebnis signalisiert, ob das Argument lexikografisch kleiner oder größer als das String-Objekt ist beziehungsweise mit diesem übereinstimmt. Das ist zum Beispiel in einer Sortiermethode wichtig. Der Sortieralgorithmus muss beim Vergleich zweier Strings wissen, wie sie einzusortieren sind.

Beispiel

zB

Drei Strings in ihrer lexikografischen Ordnung. Alle Vergleiche ergeben true:

```
System.out.println( "Justus".compareTo( "Bob" ) > 0 );
System.out.println( "Justus".compareTo( "Justus" ) == 0 );
System.out.println( "Justus".compareTo( "Peter" ) < 0 );
```

Da im ersten Fall »Justus« lexikografisch größer ist als »Bob«, ist die numerische Rückgabe der Methode compareTo() größer 0.

Der von compareTo() vorgenommene Vergleich basiert nur auf der internen numerischen Kodierung der Unicode-Zeichen. Dabei berücksichtigt compareTo() nicht die landestypischen Besonderheiten, etwa die übliche Behandlung der deutschen Umlaute. Dafür müssten wir Collator-Klassen nutzen, die später in Abschnitt 4.12.1 vorgestellt werden.

compareToIgnoreCase() ist mit equalsIgnoreCase() vergleichbar, bei der die Groß-/Kleinbeschreibung keine Rolle spielt.



Hinweis

Das JDK implementiert `compareToIgnoreCase()` mit einem `Comparator<String>`, der zwei beliebige Zeichenketten in eine Reihenfolge bringt. Der `Comparator<String>` ist auch für uns zugänglich als statische Variable `CASE_INSENSITIVE_ORDER`. Er ist zum Beispiel praktisch für sortierte Mengen, bei denen die Groß-/Kleinschreibung keine Rolle spielt. Comparatoren werden genauer in Abschnitt 8.1, »Vergleichen von Objekten«, vorgestellt.

Endet der String mit ..., beginnt er mit ...?

Interessiert uns, ob der String mit einer bestimmten Zeichenfolge beginnt (wir wollen dies *Präfix* nennen), so rufen wir die `startsWith()`-Methode auf. Eine ähnliche Methode gibt es für *Suffixe*: `endsWith()`. Sie überprüft, ob ein String mit einer Zeichenfolge am Ende übereinstimmt.



Beispiel

Teste mit `endsWith()` eine Dateinamenendung und mit `startsWith()` eine Ansprache:

```
String filename = "die besten stellungen (im schach).txt";
boolean isTxt    = filename.endsWith( ".txt" );           // true
String email    = "Sehr geehrte Frau Müller,\nDanke für Ihr Angebot.";
boolean isMale   = email.startsWith( "Sehr geehrter Herr" ); // false
```

String-Teile mit `regionMatches()` vergleichen *

Eine Erweiterung der Ganz-oder-gar-nicht-Vergleichsmethoden bietet `regionMatches()`, die Teile einer Zeichenkette mit Teilen einer anderen vergleicht. Nimmt das erste Argument von `regionMatches()` den Wahrheitswert `true` an, dann spielt die Groß-/Kleinschreibung keine Rolle – damit lässt sich dann auch ein `startsWith()` und `endsWith()` mit Vergleichen unabhängig von der Groß-/Kleinschreibung durchführen. Der Rückgabewert ist wie bei `equalsXXX()` ein `boolean`.



Beispiel

Der Aufruf von `regionMatches()` ergibt `true`.

```
String s = "Deutsche Kinder sind zu dick";
boolean b = s.regionMatches( 9, "Bewegungsarmut bei Kindern", 19, 6 );
```

Die Methode beginnt den Vergleich am neunten Zeichen, also bei »K« im String `s`, und dem 19. Buchstaben in dem Vergleichsstring, ebenfalls ein »K«. Dabei beginnt die Zählung der Zeichen wieder bei 0. Ab diesen beiden Positionen werden sechs Zeichen verglichen. Im Beispiel ergibt der Vergleich von »Kinder« und »Kinder« dann true.

Beispiel

zB

Sollte der Vergleich unabhängig von der Groß-/Kleinschreibung stattfinden, ist das erste Argument der überladenen Methode true:

```
String s = "Deutsche KINDER sind zu dick";
boolean b = s.regionMatches( true, 9, "Bewegungsarmut bei kindern", 19, 6 );
```

4.4.8 Phonetische Vergleiche *

Bei der `equals()`-Methode ist das Ergebnis nur dann true, wenn beide Zeichenketten absolut gleich sind, also jedes Zeichen »passt«. `equalsIgnoreCase()` ist schon etwas großzügiger, und hier sind etwa »vuvuzela« und »VuVuZeLa« gleich. Noch entspanntere Vergleiche erlauben Collator-Objekte, die etwa den Umlauten die Punkte nehmen, sodass »männö« und »manno« dann gleich sind.

Vergleiche aufgrund von Ähnlichkeiten und gleichem »Klang« gibt es jedoch in der Java-Standardbibliothek nicht. Das ist aber bei Namen interessant. Mein Name »Ullendorf« wird oft zu »Uhlenbohm« umgebaut, was sich im Prinzip gleich anhört (und jeder Maier, Meyer, Mayer, Meir, Myer, Meier kennt das Problem).

Zur Erkennung helfen besondere String-Algorithmen weiter. Für (insbesondere englische) Namen sind der *Soundex*-Algorithmus und seine Verbesserungen (*Double Metaphone*) entwickelt worden. Wer eine Realisierung in Java sucht, der findet bei *Apache Commons Codec* (<http://commons.apache.org/codec/userguide.html>) passende Implementierungen. So liefert etwa `isDoubleMetaphoneEqual(String value1, String value2)` einen Wahrheitswert, der aussagt, ob die Strings ähnlich sind. Interessant sind die Algorithmen auch für Korrekturhilfen.⁷ Der Name "Ullendorf" ist dem Soundex-Code U451 zugeordnet. Schreibt jemand diesen Namen falsch, etwa "Uhlenbohm", und ist dieser Name nicht im Wörterbuch, so berechnet das Programm von "Uhlenbohm" ebenfalls den Soun-

⁷ Wobei ich die Korrekturvorschläge »Ullendorf«, »Quellenbox«, »Patrouillenboot« und »Mülleborn« in den Textboxen von Google Chrome schon sehr schräg finde.

dex und kommt auf U451. Ein Blick in die Datenstruktur bei U451 liefert dann den korrekten Namen "Ullenboom" oder andere Vorschläge, die den gleichen Soundex ergeben.

Wie ähnlich denn nun Strings sind, sagen andere Algorithmen. Die *Levenshtein-Distanz* zum Beispiel berechnet sich aus der (kleinstnötigen) Anzahl der einzufügenden, zu löschen oder zu ersetzenen Zeichen, um von einem String zum anderen zu kommen; daher nennt sie sich auch *Edit-Distanz*. Von "Chris" nach "Char" ist die Edit-Distanz drei und von "Ullenboom" nach "Uhlenbohm" vier. *Jaro-Winkler* ist ein weiter Algorithmus, der die Ähnlichkeit zwischen 0 und 1 angibt. Das Projekt *SecondString* (<http://second-string.sourceforge.net/>) implementiert diese Algorithmen – und noch ein Dutzend mehr.

4.4.9 String-Teile extrahieren

Die wichtigste Methode `charAt(int index)` der Klasse `String` haben wir schon mehrfach benutzt. Sie ist aber nicht die einzige Methode, um auf gewisse Teile eines Strings zuzugreifen.

Teile eines Strings als String mit `substring()` erfragen

Wollen wir einen Teilstring aus der Zeichenkette erfragen, so greifen wir zur Methode `substring()`. Sie existiert in zwei Varianten – beide liefern ein neues `String`-Objekt zurück, das dem gewünschten Ausschnitt des Originals entspricht.

zB Beispiel

`substring(int)` liefert eine Teilzeichenkette ab einem Index bis zum Ende. Das Ergebnis ist ein neues `String`-Objekt:

```
String s1 = "Infiltration durch Penetration";
// Position:           19
String s2 = s1.substring( 19 );           // Penetration
```

Der Index von `substring(int)` gibt die Startposition (null-basiert) an, ab der Zeichen in die neue Teilzeichenkette kopiert werden. `substring(int)` liefert den Teil von diesem Zeichen bis zum Ende des ursprünglichen Strings – es ergibt `s.substring(0)` gleich `s`.

Wollen wir die Teilzeichenkette genauer spezifizieren, so nutzen wir die zweite Variante, `substring(int, int)`. Ihre Argumente geben den Anfang und das Ende des gewünschten Ausschnitts an.

zB

Beispiel

Schneide einen Teil des Strings aus:

```
String tear = "'Jede Träne kitzelt auch die Wange.'";
//          0   6   11
System.out.println( tear.substring( 6, 11 ) ); // Träne
```

Während die Startposition inklusiv ist, ist die Endposition exklusiv. Das heißt, bei der Endposition gehört das Zeichen nicht mehr zur Teilzeichenkette.

4

Die Methode `substring(int)` ist nichts anderes als eine Spezialisierung von `substring(int, int)`, denn die erste Variante mit dem Startindex lässt sich auch als `s.substring(beginIndex, s.length())` schreiben.

Selbstverständlich kommen nun diverse Indexüberprüfungen hinzu – eine `StringIndexOutOfBoundsException` meldet fehlerhafte Positionsangaben wie bei `charAt()`.

String vor/nach einem Trennstring *

Ist ein Trennzeichen gegeben und ein Teilstring vor oder nach diesem Trennzeichen gewünscht, bietet die `String`-Klasse keine Bibliotheksmethode an.⁸ Dabei wäre eine solche Methode praktisch, etwa bei Dateien, bei denen der Punkt den Dateinamen vom Suffix trennt. Wir wollen zwei statische Utility-Methoden, `substringBefore(String string, String delimiter)` und `substringAfter(String string, String delimiter)`, schreiben, die genau diese Aufgabe übernehmen. Angewendet sehen sie dann so aus (wir ignorieren für einen Moment, dass der Dateiname selbst auch einen Punkt enthalten kann):

- `substringBefore("index.html", ".") → "index"`
- `substringAfter("index.html", ".") → "html"`

Die Implementierung der Methoden ist einfach: Im ersten Schritt suchen die Methoden mit `indexOf()` nach dem Trenner. Anschließend liefern sie mit `substring()` den Teilstring vor bzw. hinter diesem gefundenen Trennstring. Noch einige Vereinbarungen: Der Trenner ist kein Teil der Rückgabe. Und taucht das Trennzeichen nicht im String auf, ist die Rückgabe von `substringBefore()` der gesamte String und bei `substringAfter()` der

⁸ Selbst XPath bietet mit `substring-before()` und `substring-after()` solche Funktionen. Und Apache Commons Lang (<http://commons.apache.org/lang/>) bildet sie auch nach in der Klasse `org.apache.commons.lang.StringUtils`.

Leerstring. String und Trenner dürfen nicht null sein. Wenn dem so ist, folgt eine NullPointerException und zeigt so den Programmierfehler an. Ausprogrammiert sehen die beiden Methoden so aus:

Listing 4.7: StringUtils.java

```
public class StringUtils
{
    /**
     * Returns the substring before the first occurrence of a delimiter. The
     * delimiter is not part of the result.
     *
     * @param string      String to get a substring from.
     * @param delimiter   String to search for.
     * @return            Substring before the first occurrence of the delimiter.
     */
    public static String substringBefore( String string, String delimiter )
    {
        int pos = string.indexOf( delimiter );

        return pos >= 0 ? string.substring( 0, pos ) : string;
    }

    /**
     * Returns the substring after the first occurrence of a delimiter. The
     * delimiter is not part of the result.
     *
     * @param string      String to get a substring from.
     * @param delimiter   String to search for.
     * @return            Substring after the last occurrence of the delimiter.
     */
    public static String substringAfter( String string, String delimiter )
    {
        int pos = string.indexOf( delimiter );

        return pos >= 0 ? string.substring( pos + delimiter.length() ) : "";
    }
}
```

Zur Übung sei es den Lesern überlassen, noch die zwei Methoden `substringBeforeLast()` und `substringAfterLast()` zu realisieren, die statt `indexOf()` die Methode `lastIndexOf()` einsetzen (mit den beiden Methoden kann auch der Dateiname selbst einen Punkt enthalten). Frage: Lässt sich in der Implementierung einfach `indexOf()` durch `lastIndexOf()` ersetzen, und das war es dann schon?

Mit `getChars()` Zeichenfolgen als Array aus dem String extrahieren *

Während `charAt()` nur ein Zeichen liefert, kopiert `getChars()` mehrere Zeichen aus einem angegebenen Bereich des Strings in ein übergebenes Feld.

Beispiel

zB

Kopiere Teile des Strings in ein Feld:

```
String s = "Blasiussegen";
char[] chars = new char[ 5 ];
int srcBegin = 7;
s.getChars( srcBegin, srcBegin + 5, chars, 0 );
System.out.println( new String(chars) ); // segen
```

`s.getChars()` kopiert ab Position 7 aus dem String `s` fünf Zeichen in die Elemente des Arrays `chars`. Das erste Zeichen aus dem Ausschnitt steht dann in `chars[0]`.

Die Methode `getChars()` muss natürlich wieder testen, ob die gegebenen Argumente im grünen Bereich liegen, das heißt, ob der Startwert nicht < 0 ist und ob der Endwert nicht über die Größe des Strings hinausgeht. Passt das nicht, löst die Methode eine `StringIndexOutOfBoundsException` aus. Liegt zudem der Startwert hinter dem Endwert, gibt es ebenfalls eine `StringIndexOutOfBoundsException`, die anzeigt, wie groß die Differenz der Positionen ist. Am besten ist es, die Endposition aus der Startposition zu berechnen, wie es im obigen Beispiel geschehen ist. Passen alle Zeichen in das Feld, kopiert die Implementierung der Methode `getChars()` mittels `System.arraycopy()` die Zeichen aus dem internen Array des String-Objekts in das von uns angegebene Ziel.

Möchten wir den kompletten Inhalt eines Strings als ein Array von Zeichen haben, so können wir die Methode `toCharArray()` verwenden. Intern arbeitet die Methode auch mit `getChars()`. Als Ziel-Array legt `toCharArray()` nur ein neues Array an, das wir dann zurückbekommen.



Hinweis

Mit folgendem Idiom lässt sich über eine Zeichenkette iterieren:

```
String string = "Herr, schmeiß Java vom Himmel!";
for ( char c : string.toCharArray() ) {
    System.out.println( c );
}
```

Diese Lösung hat aber ihren Preis, denn ein neues `char[]`-Objekt einfach für den Durchlauf zu erzeugen, kostet Speicher und Rechenzeit für die Speicherbereitstellung und die -bereinigung. Daher ist diese Variante nicht empfehlenswert. Hübscher wäre es natürlich, wenn rechts vom Doppelpunkt automatisch ein `String`-Objekt berücksichtigt werden würde und den Compiler dazu anregte, die Zeichenkette zu durchlaufen.

4.4.10 Strings anhängen, Groß-/Kleinschreibung und Leerraum

Obwohl `String`-Objekte selbst unveränderlich sind, bietet die Klasse `String` Methoden an, die aus einer Zeichenkette Teile herausnehmen oder ihr Teile hinzufügen. Diese Änderungen werden natürlich nicht am `String`-Objekt selbst vorgenommen, auf dem die Methode aufgerufen wird, sondern die Methode liefert eine Referenz auf ein neues `String`-Objekt mit verändertem Inhalt zurück.

Anhängen an Strings

Eine weitere Methode erlaubt das Anhängen von Teilen an einen `String`. Wir haben dies schon öfter mit dem Plus-Operator realisiert. Die Methode der `String`-Klasse dazu heißt `concat(String)`. Wir werden später sehen, dass es die `StringBuilder/StringBuffer`-Klassen noch weiter treiben und eine überladene Methode `append()` mit der gleichen Funktionalität anbieten. Das steckt auch hinter dem Plus-Operator. Der Compiler wandelt dies automatisch in eine Kette von `append()`-Aufrufen um.



Beispiel

Hänge das aktuelle Tagesdatum hinter eine Zeichenkette:

```
String s1 = "Das aktuelle Datum ist: ";
String s2 = new Date().toString();
String s3 = s1.concat( s2 ); // Das aktuelle Datum ist:
                           // Tue Sep 01 14:46:41 CEST 2011
```

Die concat()-Methode arbeitet relativ zügig und effizienter als der Plus-Operator, der einen temporären String-Puffer anlegt. Doch mit dem Plus-Operator ist es hübscher anzusehen (doch wie das so ist: Sieht nett aus, aber ...).

Beispiel

zB

4

Ähnlich wie im obigen Beispiel können wir Folgendes schreiben:

```
String s3 = "Das aktuelle Datum ist: " + new Date().toString();
```

Es geht sogar noch kürzer, denn der Plus-Operator ruft automatisch `toString()` bei Objekten auf:

```
String s3 = "Das aktuelle Datum ist: " + new Date();
```

`concat()` legt ein internes Feld an, kopiert die beiden Zeichenreihen per `getChars()` hinein und liefert mit einem String-Konstruktor die resultierende Zeichenkette.

Groß-/Kleinschreibung

Die Klasse Character deklariert einige statische Methoden, um einzelne Zeichen in Groß-/Kleinbuchstaben umzuwandeln. Die Schleife, die das für jedes Zeichen übernimmt, können wir uns sparen, denn dazu gibt es die Methoden `toUpperCase()` und `toLowerCase()` in der Klasse `String`. Interessant ist an beiden Methoden, dass sie einige sprachabhängige Feinheiten beachten. So zum Beispiel, dass es im Deutschen nicht wirklich ein großes »ß« gibt, denn »ß« wird zu »SS«. Gammelige Textverarbeitungen bekommen das manchmal nicht auf die Reihe, und im Inhaltsverzeichnis steht dann so etwas wie »SPAß IN DER NAßZELLE«. Aber bei möglichen Missverständnissen müsste »ß« auch zu »SZ« werden, vergleiche »SPASS IN MASZEN« mit »SPASS IN MASSEN« (ein ähnliches Beispiel steht im Duden). Diese Umwandlung ist aber nur von Klein nach Groß von Bedeutung. Für beide Konvertierungsrichtungen gibt es jedoch im Türkischen Spezialfälle, bei denen die Zuordnung zwischen Groß- und Kleinbuchstaben von der Festlegung in anderen Sprachen abweicht.

Beispiel

zB

Konvertierung von Groß- in Kleinbuchstaben und umgekehrt:

```
String s1 = "Spaß in der Naßzelle.";
String s2 = s1.toLowerCase().toUpperCase();
System.out.println( s2 );      // SPASS IN DER NASSZELLE.
System.out.println( s2.length() - s1.length() ); // 2
```

Das Beispiel dient zugleich als Warnung, dass sich im Fall von »ß« die Länge der Zeichenkette vergrößert. Das kann zu Problemen führen, wenn vorher Speicherplatz bereitgestellt wurde. Dann könnte die neue Zeichenkette nicht mehr in den Speicherbereich passen. Arbeiten wir nur mit String-Objekten, haben wir dieses Problem glücklicherweise nicht. Aber berechnen wir etwa für einen Texteditor die Darstellungsbreite einer Zeichenkette in Pixel auf diese Weise, dann sind Fehler vorprogrammiert.

Um länderspezifische Besonderheiten zu berücksichtigen, lassen sich die `toXXXCase()`-Methoden zusätzlich mit einem `Locale`-Objekt füttern (`Locale`-Objekte repräsentieren eine sprachliche Region).



Hinweis

Es gibt Konvertierungen in Groß-/Kleinbuchstaben, die abhängig von der Landessprache zu unterschiedlichen Zeichenfolgen führen. Die Angabe eines `Locale` bei den beiden `toXXXXCase()`-Methoden ist insbesondere bei türkischsprachigen Applikationen wichtig:

```
System.out.println( "TITANIK".toLowerCase() ); // titanik  
System.out.println( "TITANIK".toLowerCase(new Locale( "tr" ) ) ); // tıtanık
```

Kleiner Unterschied: Im zweiten Ergebnisstring hat das i keinen i-Punkt!



Hinweis

Die parameterlosen Methoden `toUpperCase()` und `toLowerCase()` wählen die Sprachumgebung gemäß den Länder-Einstellungen des Betriebssystems aus. Am Beispiel `toLowerCase()`:

```
public String toLowerCase() {  
    return toLowerCase( Locale.getDefault() );  
}
```

Die Voreinstellung muss nicht die beste sein.

Leerraum entfernen

In einer Benutzereingabe oder Konfigurationsdatei steht nicht selten vor oder hinter dem wichtigen Teil eines Texts Leerraum wie Leerzeichen oder Tabulatoren. Vor der Bearbeitung sollten sie entfernt werden. Die `String`-Klasse bietet dazu `trim()` an.

zB

Beispiel

Entferne Leer- und ähnliche Füllzeichen am Anfang und Ende eines Strings:

```
String s = " \tSprich zu der Hand.\n \t ";
System.out.println( "" + s.trim() + "" ); // 'Sprich zu der Hand.'
```

4

zB

Beispiel

Teste, ob ein String mit Abzug allen Weißraums leer ist:

```
boolean isBlank = "".equals( s.trim() );
```

Alternativ:

```
boolean isBlank = s.trim().isEmpty();
```

Andere Modesprachen wie Visual Basic bieten dazu noch trim()-Methoden an, die nur die Leerzeichen vorher oder nachher verwerfen. Die Java-Bibliothek bietet das so einfach nicht.

4.4.11 Suchen und ersetzen

Da String-Objekte unveränderlich sind, kann eine Veränderungsmethode nur einen neuen String mit den Veränderungen zurückgeben. Wir finden in Java vier Methoden, die suchen und ersetzen:

```
final class java.lang.String
    implements Serializable, Comparable<String>, CharSequence
```

- `String replace(char oldChar, char newChar)`. Ersetzt einzelne Zeichen.
- `String replace(CharSequence target, CharSequence replacement)`. Ersetzt eine Zeichenkette durch eine andere Zeichenkette.
- `String replaceAll(String regex, String replacement)`. Ersetzt alle Strings, die durch einen regulären Ausdruck beschrieben werden.
- `String replaceFirst(String regex, String replacement)`. Ersetzt den ersten String, der durch einen regulären Ausdruck beschrieben wird.

Ersetzen ohne reguläre Ausdrücke

Die `replace(char, char)`-Methode ersetzt einzelne Zeichen.

zB

Beispiel

Ändere den in einer Zeichenkette vorkommenden Buchstaben »o« in »u«:

```
String s1 = "Honolulu";
String s2 = s1.replace( 'o', 'u' );           // s2 = "Hunululu"
```

Das String-Objekt mit dem Namen `s1` wird selbst nicht verändert. Es wird nur ein neues String-Objekt mit dem Inhalt »Hunululu« erzeugt und von `replace()` zurückgegeben.

Gibt es etwas zu ersetzen, erzeugt `replace()` intern ein neues Zeichenfeld, führt die Ersetzungen durch und konvertiert das interne Zeichenfeld in ein String-Objekt, was die Rückgabe ist. Gab es nichts zu ersetzen, bekommen wir das gleiche String-Objekt zurück, das die Anfrage stellte. Die `replace()`-Methode ersetzt immer alle Zeichen. Eine Variante, die nur das erste Zeichen ersetzt, müssen wir uns selbst schreiben.

Eine zweite überladene Variante, `replace(CharSequence, CharSequence)`, sucht nach allen auftretenden Zeichenfolgen und ersetzt sie durch eine andere Zeichenfolge. Der Ersetzungsstring kann auch leer sein.

zB

Beispiel

Im String `s` soll »Schnecke« durch »Katze« ersetzt werden:

```
String s =
    "Schnecken erschrecken, wenn Schnecken an Schnecken schlecken, " +
    "weil zum Schrecken vieler Schnecken Schnecken nicht schmecken.";
System.out.println( s.replace("Schnecke", "Katze") );
```

Das Ergebnis auf dem Bildschirm ist: »Katzen erschrecken, wenn Katzen an Katzen schlecken, weil zum Schrecken vieler Katzen Katzen nicht schmecken.«

Suchen und ersetzen mit regulären Ausdrücken

Die Methoden `replaceAll()` und `replaceFirst()` suchen in Zeichenketten mithilfe von regulären Ausdrücken (mehr dazu folgt in Abschnitt 4.8, »Reguläre Ausdrücke«) und nehmen Ersetzungen vor; `replaceFirst()` ersetzt, wie der Name schon sagt, nur das erste Auftreten.

zB

Beispiel

Mehr als zwei Leerzeichen in Folge sollen auf ein Leerzeichen komprimiert werden:

```
String s = "Alles fit im Schritt?";
System.out.println( s.replaceAll( " +", " " ) ); // Alles fit im Schritt?
System.out.println( s.replaceFirst( " +", " " ) ); // Alles fit im Schritt?
```

4

Weil der Suchstring immer ein regulärer Ausdruck ist und Sonderzeichen wie ».<« oder »+« eine Sonderrolle einnehmen, eignen sich `replaceAll()` und `replaceFirst()` nicht direkt für allgemeine Ersetzungsaufgaben; hier ist die `replace()`-Methode passender.

zB

Beispiel

Für eine String-Ersetzung stellen wir `replace()` und `replaceAll()` nebeneinander:

```
String s =
    "'Tag, Karl.' 'Wie geht's, Karl?' 'Gut, Karl.' 'Kahl, Karl?' 'Ja, Karl, ganz kahl.'";
System.out.println( s.replace( ".", "!" ) );
```

Der Aufruf ersetzt alle Punkte durch Ausrufezeichen, sodass das Ergebnis wie folgt lautet:

'Tag, Karl!' 'Wie geht's, Karl?' 'Gut, Karl!' 'Kahl, Karl?' 'Ja, Karl, ganz kahl!'

Nutzen wir `s.replaceAll(".", "!")`, führt das nicht zum Erfolg, sondern nur zu der Zeichenkette:

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Der Punkt steht in regulären Ausdrücken für beliebige Zeichen. Erst, wenn er mit \\ ausmaskiert wird – wegen des Sonderstatus von ».<« muss auch dieses Zeichen selbst ausmaskiert werden –, liefert die Anweisung wie in `s.replaceAll("\.", "!")` das gewünschte Ergebnis. Die statische Methode `Pattern.quote(String)` maskiert die Pattern-Sonderzeichen für uns aus, sodass auch `s.replaceAll(Pattern.quote("."), "!")` gut funktioniert. Zur Klasse `java.util.regex.Pattern` und regulären Ausdrücken folgt mit Abschnitt 4.8 ein eigenes großes Unterkapitel.

4.4.12 String-Objekte mit Konstruktoren neu anlegen *

Liegt die Zeichenkette nicht als String-Literal vor, lassen sich mit den unterschiedlichen Konstruktoren der `String`-Klasse neue String-Objekte aufbauen. Die meisten Konstruk-

toren sind für Spezialfälle gedacht und kommen in normalen Java-Programmen nicht vor:

```
final class java.lang.String
    implements CharSequence, Comparable<String>, Serializable
```

- `String()`
Erzeugt ein neues Objekt ohne Zeichen (den leeren String "").
- `String(String string)`
Erzeugt ein neues Objekt mit einer Kopie von string. Es wird selten benötigt, da String-Objekte unveränderbar (*immutable*) sind.
- `String(char[] value)`
Erzeugt ein neues Objekt und kopiert die im char-Feld vorhandenen Zeichen in das neue String-Objekt.
- `String(char[] value, int offset, int length)`
Erzeugt wie `String(char[])` einen String aus einem Ausschnitt eines Zeichenfelds. Der verwendete Ausschnitt beginnt bei dem Index offset und umfasst length Zeichen.
- `String(byte[] bytes)`
Erzeugt ein neues Objekt aus dem Byte-Feld. Das byte-Array enthält keine Unicode-Zeichen, sondern eine Folge von Bytes, die nach der Standardkodierung der jeweiligen Plattform in Zeichen umgewandelt werden.
- `String(byte[] bytes, int offset, int length)`
Erzeugt wie `String(byte[])` einen String aus einem Ausschnitt eines Byte-Felds.
- `String(byte[] bytes, String charsetName) throws UnsupportedEncodingException`
Erzeugt einen neuen String von einem Byte-Array mithilfe einer speziellen Zeichenkodierung, die die Umwandlung von Bytes in Unicode-Zeichen festlegt.
- `String(byte[] bytes, int offset, int length, String charset)
 throws UnsupportedEncodingException`
Erzeugt einen neuen String mit einem Teil des Byte-Arrays mithilfe einer speziellen Zeichenkodierung.
- `String(StringBuffer buffer)`
- `String(StringBuilder builder)`
Erzeugt aus einem veränderlichen StringBuffer/StringBuilder-Objekt ein unveränderliches String-Objekt, das dieselbe Zeichenreihe repräsentiert.
- `String(int[] codePoints, int offset, int count)`
Erzeugt ein String-Objekt mit Unicode-Codepoints, die Zeichen über int kodieren.

Die Konstruktoren sind im Speziellen nur dann nötig, wenn aus einer Fremdrepräsentation wie einem `StringBuilder`, `StringBuffer`, `char[]` oder `byte[]` oder Teilen von ihnen ein `String`-Objekt aufgebaut werden soll.

Beispiel

zB

4

Erzeuge einen String einer gegebenen Länge:

```
public static String generateStringWithLength( int len, char fill )
{
    if ( len < 0 )
        return null;

    char[] cs = new char[ len ];
    Arrays.fill( cs, fill );
    return new String( cs );
}
```

In der `String`-Klasse gibt es keine Methode, die eine Zeichenkette einer vorgegebenen Länge aus einem einzelnen Zeichen erzeugt.

Beispiel

zB

Teste, ob zwei Zeichenketten Anagramme darstellen, also Zeichenfolgen, die beim Vertauschen von Buchstaben gleich sind:

```
String a1 = "iPad", a2 = "Paid";
char[] a1chars = a1.toCharArray();
char[] a2chars = a2.toCharArray();
Arrays.sort( a1chars );
Arrays.sort( a2chars );
boolean isAnagram = new String(a1chars).equalsIgnoreCase(new String(a2chars));
System.out.println( isAnagram );      // true
```

Die Methode `Arrays.sort()` haben wir bisher noch nicht kennengelernt, aber ihr Sinn ergibt sich intuitiv: Die Methode sortiert ein Feld, in dem Fall das `char`-Feld.

Über den Konstruktorauftrag `new String(String)`

Ein Konstruktor führt leicht zur Verwirrung, und zwar der Konstruktor, der einen anderen String annimmt. So ergeben die beiden folgenden Zeilen die Referenz auf ein String-Objekt:

```
String rudi = "There is no spoon";
String rudi = new String( "There is no spoon" );
```

Die zweite Lösung erzeugt unnötigerweise ein zusätzliches String-Objekt, denn das Literal ist ja schon ein vollwertiges String-Objekt.



Tuning-Hinweis

Der Konstruktor ist *nur* für eine besondere Optimierung zu gebrauchen, die in der Regel wie folgt aussieht (sei s ein großer String und t ein Teilstring):

```
String s = ...
String t = new String( s.substring(...) );
```

Die String-Klasse in Java ist nichts anderes als eine Abstraktion von einem darunterliegenden char-Feld.⁹ Bei einem `substring()` wird kein neues char-Feld mit der Teilzeichenkette aufgebaut, sondern es wird das ursprüngliche char-Feld (in unserem Beispiel von s) genutzt, und es werden lediglich die Start- und End-Positionen gesetzt. Ein String-Objekt enthält daher nicht nur intern ein Attribut für das char-Feld, sondern auch noch `offset` (also den Startpunkt) und die Länge.¹⁰ Somit ist die Operation `substring()` sehr performant, da keine Zeichenfelder kopiert werden müssen. Das Problem: Ist das von s referenzierte char-Feld sehr groß, wird dieses Feld ebenfalls vom Teilstring referenziert. Wenn das ursprüngliche String-Objekt s vom GC entfernt wird, bleibt trotzdem das große char-Feld bestehen, denn es wird vom Teilstring referenziert. Um den Speicherbedarf in diesem Fall zu optimieren, ist der `new String(String)`-Konstruktor geeignet, denn er legt ein neues kompaktes char-Feld an, das ausschließlich die Zeichen speichert.

⁹ Die ersten Zeilen in der Klasse String beginnen mit:

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
```

¹⁰ `/** The offset is the first index of the storage that is used. */`

```
private final int offset;
/** The count is the number of characters in the String. */
private final int count;
```

**Tuning-Hinweis (Forts.)**

Somit ist kein Verweis mehr auf das ursprüngliche Feld vorhanden, was dann der GC wegräumen kann, wenn das ursprüngliche String-Objekt s auch nicht mehr existiert.

4

Strings im Konstantenpool

Die JVM erzeugt für jedes Zeichenketten-Literal automatisch ein entsprechendes String-Objekt. Das geschieht für jede konstante Zeichenkette höchstens einmal, egal wie oft sie im Programmverlauf benutzt wird und welche Klassen den String nutzen. Dieses String-Objekt »lebt« in einem Bereich, der *Konstantenpool* genannt wird.¹¹

Hinweis

Nehmen wir an, die Anweisung

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

steht in einer Klasse A und in einer anderen Klasse B steht:

```
int len = "tutego".length();
```

Dann gibt es die Zeichenfolge »tutego« als String-Objekt nur ein einziges Mal in der Laufzeitumgebung.

Bei konstanten Werten führt der Compiler Optimierungen durch, etwa in der Art, dass er konstante Ausdrücke gleich berechnet. Nicht nur setzt er für Ausdrücke wie `1 + 2` das Ergebnis `3` ein, auch aufgebrochene konstante String-Teile, die mit Plus konkateniert werden, fügt der Compiler zu einer Zeichenkette zusammen.

Beispiel

Die erste und zweite Deklaration sehen im Bytecode gleich aus:

```
String s =
"Operating systems are like underwear - nobody really wants to look at them.";
String s = "Operating systems are like underwear " +
'-' + " nobody really wants to look at them.;"
```

¹¹ Die Java-Bibliothek implementiert hier das Entwurfsmuster *Fliegengewicht* (Flyweight-Pattern) der Gang of Four.

zB Beispiel (Forts.)

Der Compiler fügt die Zeichenketten¹² automatisch zu einer großen Zeichenkette zusammen, sodass keine Konkatenation zur Laufzeit nötig ist.

Leerer String, Leer-String oder Null-String

Die Anweisungen

```
String s = "";
```

und

```
String s = new String();
```

referenzieren in beiden Fällen String-Objekte, die keine Zeichen enthalten. Die zweite Schreibweise erzeugt aber ein neues String-Objekt, während im ersten Fall das String-Literal im Konstantenpool liegt.

Ein String ohne Zeichen nennen wir *leeren String, Leer-String* oder *Null-String*. Der letzte Begriff ist leider etwas unglücklich gewählt, sodass wir ihn im Buch nicht nutzen, denn der Begriff *Null-String* kann leicht mit dem Begriff *null-Referenz* verwechselt werden. Doch während Zugriffe auf einem Null-String unproblematisch sind, führen Dereferenzierungen auf der *null-Referenz* unweigerlich zu einer *NullPointerException*:

```
String s = null;
System.out.println( s );           // Ausgabe: null
s.length();                      // ☣ NullPointerException
```

`printXXX(null)` führt zu der Konsolenausgabe »null« und zu keiner Ausnahme, da es eine Fallunterscheidung in `printXXX()` gibt, die die `null`-Referenz als Sonderfall betrachtet.¹³ Der Zugriff auf `s` über `s.length()` führt dagegen zur unbeliebten `NullPointerException`.

¹² Das Zitat stammt übrigens von Bill Joy (eigentlich heißt er William Nelson Joy), der Sun Microsystems mit gegründet hat. Er war an der Entwicklung einer beeindruckenden Anzahl von Tools und Technologien beteiligt, wie dem Unix-Kernel, TCP/IP (»Edison of the Internet«), dem Dateisystem NFS, Java, SPARC-Prozessoren, dem vi-Editor usw.

¹³ In der Implementierung von `PrintStream` von Sun:

```
public void print( String s ) { if ( s == null ) s = "null"; write( s ); }
```

4.5 Konvertieren zwischen Primitiven und Strings

Bevor ein Datentyp auf dem Bildschirm ausgegeben, zum Drucker geschickt oder in einer ASCII-Datei gespeichert werden kann, muss das Java-Programm ihn in einen String konvertieren. Wenn wir etwa die Zahl 7 ohne Umwandlung ausgeben, hätten wir keine 7 auf dem Bildschirm, sondern einen Pieps aus dem Lautsprecher – je nach Implementierung. Auch umgekehrt ist eine Konvertierung wichtig: Gibt der Benutzer in einem Dialog sein Alter an, ist das zuerst immer ein String. Diesen muss die Anwendung in einem zweiten Schritt in eine Ganzzahl konvertieren, um etwa eine Altersabfrage zu realisieren.

4.5.1 Unterschiedliche Typen in String-Repräsentationen konvertieren

Die statischen überladenen `String.valueOf()`-Methoden liefern die String-Repräsentation eines primitiven Werts oder eines Objekts.

Beispiel

zB

Konvertierungen einiger Datentypen in Strings:

```
String s1 = String.valueOf( 10 );                                // 10
String s2 = String.valueOf( Math.PI );                            // 3.141592653589793
String s3 = String.valueOf( 1 < 2 );                             // true
```

Die `valueOf()`-Methode ist überladen, und insgesamt gibt es für jeden primitiven Datentyp eine Implementierung:

```
final class java.lang.String
    implements CharSequence, Comparable<String>, Serializable
```

- static String valueOf(boolean b)
 - static String valueOf(char c)
 - static String valueOf(double d)
 - static String valueOf(float f)
 - static String valueOf(int i)
 - static String valueOf(long l)
- Liefert die String-Repräsentation der primitiven Elemente.

- static String valueOf(char[] data)
- static String valueOf(char[] data, int offset, int count)
Liefert vom char-Feld oder einem Ausschnitt des char-Feldes ein String-Objekt.

Die Methode valueOf(Object)

Der valueOf()-Methode kann auch ein beliebiges Objekt übergeben werden:

zB Beispiel

Konvertierungen einiger Objekte in String-Präsentationen:

```
String r = String.valueOf( new java.awt.Point() ); // java.awt.Point[x=0,y=0]
String s = String.valueOf( new java.io.File(".") ); // .
String t = String.valueOf( new java.util.Date() ); // Tue Jul 20 13:07:16 CEST 2010
```

Da jedes Objekt eine `toString()`-Methode besitzt, ruft `valueOf()` diese einfach auf. Das heißt, die String-Umsetzung wird einfach an das Objekt delegiert.



Implementierung von String.valueOf(Object)

```
public static String valueOf( Object obj )
{
    return (obj == null) ? "null" : obj.toString();
}
```

Die Sonderbehandlung testet, ob null übergeben wurde, und liefert dann einen gültigen String mit dem Inhalt "null".

Da `String.valueOf(null)` die Rückgabe »null« liefert, gibt auch eine Ausgabe wie `System.out.println(null)` den String »null« auf der Konsole aus, denn `println()` ruft intern `String.valueOf()` auf. Genauso ergibt `System.out.println(null + "0")` die Ausgabe "nullo", da null als Glied in der Additionskette steht.

```
final class java.lang.String
implements CharSequence, Comparable<String>, Serializable
```

- static String valueOf(Object obj)
Ist obj ungleich null, liefert die Methode `obj.toString()`, andernfalls die Rückgabe »null«.

4.5.2 Stringinhalt in einen primitiven Wert konvertieren

Für das Parsen eines Strings – zum Beispiel von "123" aus einer Benutzereingabe in die Ganzzahl 123 – ist nicht die Klasse `String` verantwortlich, sondern spezielle Klassen, die für jeden primitiven Datentyp vorhanden sind. Die Klassen deklarieren statische `parseXXX()`-Methoden, wie die folgende Tabelle zeigt:

Klasse	Konvertierungsmethode	Rückgabetyp
<code>java.lang.Boolean</code>	<code>parseBoolean(String s)</code>	<code>boolean</code>
<code>java.lang.Byte</code>	<code>parseByte(String s)</code>	<code>byte</code>
<code>java.lang.Short</code>	<code>parseShort(String s)</code>	<code>short</code>
<code>java.lang.Integer</code>	<code>parseInt(String s)</code>	<code>int</code>
<code>java.lang.Long</code>	<code>parseLong(String s)</code>	<code>long</code>
<code>java.lang.Double</code>	<code>parseDouble(String s)</code>	<code>double</code>
<code>java.lang.Float</code>	<code>parseFloat(String s)</code>	<code>float</code>

Tabelle 4.7: Methoden zum Konvertieren eines Strings in einen primitiven Typ

Für jeden primitiven Typ gibt es eine sogenannte *Wrapper-Klasse* mit `parseXXX()`-Konvertiermethoden. Die Bedeutung der Klassen erklärt Abschnitt 8.2 »Wrapper-Klassen und Autoboxing«, genauer. An dieser Stelle betrachten wir nur die Konvertierungsfunktionalität.

Die Methode `Double.parseDouble()` wollen wir in einem Beispiel nutzen. Der Benutzer soll in einem grafischen Dialog nach einer Fließkommazahl gefragt werden, und von dieser Zahl soll dann der Sinus und Kosinus auf dem Bildschirm ausgegeben werden:

Listing 4.8: SinusAndCosinus.java

```
import javax.swing.JOptionPane;

public class SinusAndCosinus
{
    public static void main( String[] args )
    {
        String s = JOptionPane.showInputDialog( "Bitte Zahl eingeben" );
        double value = Double.parseDouble( s );
    }
}
```

```

        System.out.println( "Sinus: " + Math.sin( value ) );
        System.out.println( "Kosinus: " + Math.cos( value ) );
    }
}

```

parseXXX() und mögliche NumberFormatException-Fehler

Kann eine parseXXX()-Methode eine Konvertierung nicht durchführen, weil sich ein String wie "1lala2lö" eben nicht konvertieren lässt, löst sie eine NumberFormatException aus. Das ist auch der Fall, wenn parseDouble() als Dezimaltrenner ein Komma statt eines Punktes empfängt. Bei der statischen Methode parseBoolean() ist die Groß-/Kleinschreibung irrelevant.



Hinweis

Dieser NumberFormatException-Fehler kann als Test dienen, ob eine Zeichenkette eine Zahl enthält oder nicht, denn eine Prüfmethode wie Integer.isInteger() gibt es *nicht*. Eine Alternative ist, einen regulären Ausdruck zu verwenden – diese Variante wird später in Abschnitt 4.8 vorgestellt.

parseXXX() und Leerzeichen *

Die statische Konvertierungsmethode parseInt() schneidet keine Leerzeichen ab und würde einen Parserfehler melden, wenn der String etwa mit einem Leerzeichen endet. (Die Helden der Java-Bibliothek haben allerdings bei Float.parseFloat() und Double.parseDouble() anders gedacht: Hier wird die Zeichenkette vorher schlank getrimmt.)



Beispiel

Leerzeichen zur Konvertierung einer Ganzzahl abschneiden:

```

String s = " 1234      ".trim();           // s = "1234"
int i = Integer.parseInt( s );            // i = 1234

```

Das, was bei einem String.valueOf() als Ergebnis erscheint – und das ist auch das, worauf zum Beispiel System.out.print() basiert –, kann parseXXX() wieder in den gleichen Wert zurückverwandeln.

**Hinweis**

Eine Methode `Character.parseCharacter(String)` fehlt. Eine vergleichbare Realisierung ist, auf das erste Zeichen eines Strings zuzugreifen, etwa so: `char c = s.charAt(0)`.

4.5.3 String-Repräsentation im Format Binär, Hex, Oktal *

Neben den überladenen statischen `String.valueOf(primitive)`-Methoden, die eine Zahl als String-Repräsentation im vertrauten Dezimalsystem liefern, und den `parseXXX()`-Umkehrmethoden der Wrapper-Klassen gibt es weitere Methoden zum Konvertieren und Parsen in der

- binären (Basis 2)
- oktalen (Basis 8)
- hexadezimalen (Basis 16)
- und in der Darstellung einer beliebigen Basis (bis 36).

Die Methoden zum Bilden der String-Repräsentation sind nicht an `String`, sondern zusammen mit Methoden zum Parsen an den Klassen `Integer` und `Long` festgemacht.

String-Repräsentationen aufbauen

Zum einen gibt es in den Klassen `Integer` und `Long` die allgemeinen Klassenmethoden `toString(int i, int radix)` für einen beliebigen Radix, und zum anderen gibt es Spezialmethoden für den Radix 2, 8 und 16.

```
final class java.lang.Integer
extends Number
implements Comparable<Integer>, Serializable
```

- `static String toBinaryString(int i)`
- `static String toOctalString(int i)`
- `static String toHexString(int i)`
Erzeugt eine Binärrepräsentation (Basis 2), Oktalzahlrepräsentation (Basis 8) beziehungsweise Hexadezimalrepräsentation (Basis 16) der vorzeichenlosen Zahl.
- `static String toString(int i, int radix)`
Erzeugt eine String-Repräsentation der Zahl zur angegebenen Basis.

```
final class java.lang.Long
extends Number
implements Comparable<Long>, Serializable
```

- static String toBinaryString(long i)
- static String toOctalString(long i)
- static String toHexString(long i)

Erzeugt eine Binärrepräsentation (Basis 2), Oktalzahlrepräsentation (Basis 8) beziehungsweise Hexadezimalrepräsentation (Basis 16) der vorzeichenlosen Zahl. Achtung: Wenn die Zahl negativ ist, wird i ohne Vorzeichen behandelt und 2^{32} addiert.

- static String toString(long i, int radix)

Erzeugt eine String-Repräsentation der Zahl zur angegebenen Basis. Negative Zahlen bekommen auch ein negatives Vorzeichen.

Der Parametertyp ist `int` beziehungsweise `long` und nicht `byte`. Dies führt zu Ausgaben, die einkalkuliert werden müssen. Genauso werden führende Nullen grundsätzlich nicht mit ausgegeben.

Anweisung	Ergebnis
<code>Integer.toHexString(15)</code>	f
<code>Integer.toHexString(15)</code>	10
<code>Integer.toHexString(127)</code>	7f
<code>Integer.toHexString(128)</code>	80
<code>Integer.toHexString(255)</code>	ff
<code>Integer.toHexString(256)</code>	100
<code>Integer.toHexString(-1)</code>	fffffff

Tabelle 4.8: Beispiele für die `toHexString()`-Methode

Die Ausgaben mit `printf()` beziehungsweise die Formatierung mit `String.format()` bieten eine Alternative, die später vorgestellt wird.



Hinweis

Eine Konvertierung mit `toHexString(x)` ist bei negativen Zahlen nicht die gleiche wie mit `toString(x, 16)`:

```
System.out.println( Integer.toHexString( -10 ) ); // ffffff6
System.out.println( Integer.toString( -10, 16 ) ); // -a
```

Hinweis (Forts.)

Hier kommt bei `toHexString()` zum Tragen, was als Bemerkung in der Java-Dokumentation angegeben ist, nämlich dass bei negativen Zahlen die Zahl ohne Vorzeichen genommen wird (also 10) und dann 2^{32} addiert wird. Bei `toString()` und einem beliebigen Radix ist das nicht so.

**Parsen von String mit Radix**

Eine Methode zum Konvertieren eines Strings in eine Ganzzahl-Methode für eine gegebene Basis findet sich in den Klassen `Integer` und `Long`:

- `Integer.parseInt(String s, int radix)`
- `Long.parseLong(String s, int radix)`

Einige Anwendungsfälle:

Konvertierungsauftrag	Ergebnis
<code>parseInt("0", 10)</code>	0
<code>parseInt("473", 10)</code>	473
<code>parseInt("-0", 10)</code>	0
<code>parseInt("-FF", 16)</code>	-255
<code>parseInt("1100110", 2)</code>	102
<code>parseInt("2147483647", 10)</code>	2147483647
<code>parseInt("-2147483648", 10)</code>	-2147483648
<code>parseInt("2147483648", 10)</code>	throws NumberFormatException
<code>parseInt("99", 8)</code>	throws NumberFormatException
<code>parseInt("Papa", 10)</code>	throws NumberFormatException
<code>parseInt("Papa", 27)</code>	500050

Tabelle 4.9: Beispiele für `Integer.parseInt()` mit unterschiedlichen Zahlenbasen

Beispiel

zB

Der Radix geht bis 36 (zehn Ziffern und 26 Kleinbuchstaben). Mit Radix 36 können zum Beispiel ganzzahlige IDs kompakter dargestellt werden, als wenn sie dezimal wären:

zB Beispiel (Forts.)

```
String string = Long.toString( 2656437647773L, 36 );
System.out.println( string ); // xwcmdz8d
long parseInt = Long.parseLong( string, 36 );
System.out.println( parseInt ); // 265643764773
```

Im Fall von String-Konvertierung existieren für die Standard-Basen 2, 8 und 16 spezielle Methoden wie `toHexString()`, aber zum Parsen gibt es sie nicht. Eine Hexadezimalzahl wird daher mit `parseInt(s, 16)` verarbeitet, aber eine Methode wie `parseHex(String)` steht nicht bereit.

Nur in den Klassen `Integer` und `Long` gibt es die Unterstützung für eine Basis auch ungleich 10:

```
final class java.lang.Integer
extends Number
implements Comparable<Integer>, Serializable
```

- static int parseInt(String s)
- static int parseInt(String s, int radix)

```
final class java.lang.Long
extends Number
implements Comparable<Long>, Serializable
```

- static long parseLong(String s)
- static long parseLong(String s, int radix)

→ Hinweis

Die Methoden `parseInt()` und `parseLong()` verhalten sich bei der String-Repräsentation von negativen Zahlen nicht so, wie zu erwarten wäre:

```
System.out.println( Integer.parseInt( "7fffffff", 16 ) ); // 2147483647
System.out.println( Integer.parseInt( "80000000", 16 ) ); // ☠ Number-
                                                       // FormatException
```

Hinweis (Forts.)

0x7fffffff ist die größte darstellbare positive int-Zahl. Statt bei 0x80000000 den Wert –2147483648 zu liefern, gibt es aber eine NumberFormatException. Die Java-API-Dokumentation gibt zwar auch dieses Beispiel an, stellt dieses Verhalten aber nicht besonders klar. Es gibt den Fall, dass bei negativen Zahlen und parseInt()/parseLong() auch ein Minus als Vorzeichen angegeben werden muss. Die parseXXX()-Methoden sind also keine Umkehrmethoden zu etwa toHexString(), aber immer zu toString():

```
System.out.println( Integer.toString( -2147483648, 16 ) ); // -80000000  
System.out.println( Integer.parseInt( "-80000000", 16 ) ); // -2147483648
```

4.6 Veränderbare Zeichenketten mit StringBuilder und StringBuffer

Zeichenketten, die in der virtuellen Maschine in String-Objekten gespeichert sind, haben die Eigenschaft, dass ihr Inhalt nicht mehr verändert werden kann. Anders verhalten sich die Exemplare der Klasse `StringBuilder` und `StringBuffer`, an denen sich Veränderungen vornehmen lassen. Die Veränderungen betreffen anschließend das `StringBuilder/StringBuffer`-Objekt selbst, und es wird kein neu erzeugtes Objekt als Ergebnis geliefert, wie zum Beispiel beim Plus-Operator und der `concat()`-Methode bei herkömmlichen String-Objekten. Sonst sind sich aber die Implementierung von String-Objekten und `StringBuilder/StringBuffer`-Objekten ähnlich. In beiden Fällen nutzen die Klassen ein internes Zeichenfeld.

Die Klasse `StringBuilder` bietet die gleichen Methoden wie `StringBuffer`, nur nicht synchronisiert. Bei nebenläufigen Programmen kann daher die interne Datenstruktur vom `StringBuilder`-Objekt inkonsistent werden, sie ist aber dafür bei nicht-nebenläufigen Zugriffen ein wenig schneller.

Überblick

`StringBuilder` und `StringBuffer` sind schnell erklärt: Es gibt einen Konstruktor, der die Objekte aufbaut, Modifizierungsmethoden wie `append()` und eine `toString()`-Methode, die das Ergebnis als String liefert. Nutzen wir dies für eine eigene Methode `enumerate()`, die ein Stringfeld abläuft und eine Bildschirmaufzählung erzeugt:

Listing 4.9: Enumerator.java

```
public class Enumerator
{
    public static String enumerate( String... lines )
    {
        if ( lines == null || lines.length == 0 )
            return "";

        StringBuilder sb = new StringBuilder();

        for ( int i = 0; i < lines.length; i++ )
        {
            sb.append( i + 1 );
            sb.append( ". " );
            sb.append( lines[i] );
            sb.append( '\n' );
        }

        return sb.toString().trim();
    }

    public static void main( String[] args )
    {
        System.out.println(enumerate( "Aufstehen", "Frühstück" ) );
    }
}
```

Die Ausgabe ist:

1. Aufstehen
2. Frühstück

4.6.1 Anlegen von **StringBuilder/StringBuffer**-Objekten

Mit mehreren Konstruktoren lassen sich **StringBuilder/StringBuffer**-Objekte aufbauen:

```
final class java.lang.StringBuffer
final class java.lang.StringBuilder
implements Appendable, CharSequence, Serializable
```

- `StringBuffer()`

- `StringBuilder()`

Legt ein neues Objekt an, das die leere Zeichenreihe enthält und Platz für (zunächst) bis zu 16 Zeichen bietet. Bei Bedarf wird automatisch Platz für weitere Zeichen bereitgestellt.

- `StringBuffer(int length)`

- `StringBuilder(int length)`

Wie oben, jedoch reicht die anfängliche Kapazität des Objekts für die angegebene Anzahl an Zeichen. Optimalerweise ist die Größe so zu setzen, dass sie der Endgröße der dynamischen Zeichenfolge nahekommt.

- `StringBuffer(String str)`

- `StringBuilder(String str)`

Baut ein Objekt, das eine Kopie der Zeichen aus `str` enthält. Zusätzlich wird bereits Platz für 16 weitere Zeichen eingeplant.

- `StringBuffer(CharSequence seq)`

- `StringBuilder(CharSequence seq)`

Erzeugt ein neues Objekt aus einer `CharSequence`. Damit können auch die Zeichenfolgen anderer `StringBuffer`- und `StringBuilder`-Objekte Basis dieses Objekts werden.

Da nur `String`-Objekte von der Sprache bevorzugt werden, bleibt uns allein der explizite Aufruf eines Konstruktors, um `StringBuilder/StringBuffer`-Exemplare anzulegen. Alle `String`-Literale in Anführungszeichen sind ja schon Exemplare der Klasse `String`.

Hinweis



Weder in der Klasse `String` noch in `StringBuilder/StringBuffer` existiert ein Konstruktor, der explizit ein `char` als Parameter zulässt, um aus dem angegebenen Zeichen eine Zeichenkette aufzubauen. Dennoch gibt es bei `StringBuilder/StringBuffer` einen Konstruktor, der ein `int` annimmt, wobei die übergebene Ganzzahl die interne Startgröße des Puffers spezifiziert. Rufen wir den Konstruktor mit `char` auf – etwa einem »*« –, so konvertiert der Compiler automatisch das Zeichen in ein `int`. Das resultierende Objekt enthält kein Zeichen, sondern hat nur eine anfängliche Kapazität von 42 Zeichen, da 42 der ASCII-Code des Sternchens ist. Korrekt ist daher für ein Zeichen `c` nur Folgendes:
`new StringBuilder("") + c` oder `new StringBuilder().append(c)`.

4.6.2 StringBuilder/StringBuffer in andere Zeichenkettenformate konvertieren

StringBuilder/StringBuffer werden in der Regel intern in Methoden eingesetzt, aber tauchen selten als Parameter- oder Rückgabetyp auf. Aus den Konstruktoren der Klassen konnten wir ablesen, wie bei einem Parametertyp String etwa ein StringBuilder aufgebaut wird, es fehlt aber der Weg zurück.

```
final class java.lang.StringBuffer  
final class java.lang.StringBuilder  
implements Appendable, CharSequence, Serializable
```

- `String toString()`
Erzeugt aus der aktuellen Zeichenkette ein String-Objekt.
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
Kopiert einen gewünschten Ausschnitt in ein char-Feld.

4.6.3 Zeichen(folgen) erfragen

Die bekannten Anfragemethoden aus String finden wir auch beim StringBuilder/StringBuffer wieder. So verhalten sich `charAt()` und `getChars()` bei Exemplaren beider Klassen identisch. Auch `substring(int start)` und `substring(int start, int end)` sind aus der Klasse String bekannt. Wenn nur diese Methoden nötig sind, ist auch ein StringBuilder/StringBuffer unnötig und ein String-Objekt selbst reicht.

4.6.4 Daten anhängen

Die häufigste Anwendung von StringBuilder/StringBuffer-Objekten ist das Zusammenfügen von Texten aus Daten unterschiedlichen Typs. Dazu deklarieren die Klassen eine Reihe von `append()`-Methoden, die mit unterschiedlichen Datentypen überladen sind. Die `append()`-Methoden von StringBuilder geben einen StringBuilder zurück und die von StringBuffer einen StringBuffer. Die `append()`-Methoden hängen sich immer an das Ende an und vergrößern den internen Platz – das interne char-Feld –, falls es nötig ist. Ein neues StringBuilder/StringBuffer-Objekt erzeugen sie nicht.

zB

4

Beispiel

Hänge alle Argumente aneinander, und liefere das Ergebnis als String:

Listing 4.10: `StringBuilderToStringDemo.java, join`

```
public static String join( Object... strings )
{
    StringBuilder result = new StringBuilder();

    for ( Object string : strings )
        result.append( string );

    return result.toString();
}
```

Die mit `Object` parametrisierte `append()`-Methode ruft automatisch `toString()` auf den Objekten auf. Ein Beispelauftrag könnte so aussehen: `join("Aus", ' ', "die Maus")`.

Die Zusammenfassung listet alle Methoden auf:

```
final class java.lang.StringBuffer
final class java.lang.StringBuilder
implements Appendable, CharSequence, Serializable
```

- `StringBuilder/StringBuffer append(boolean b)`
- `StringBuilder/StringBuffer append(char c)`
- `StringBuilder/StringBuffer append(char[] str)`
- `StringBuilder/StringBuffer append(char[] str, int offset, int len)`
- `StringBuilder/StringBuffer append(CharSequence s)`
- `StringBuilder/StringBuffer append(CharSequence s, int start, int end)`
- `StringBuilder/StringBuffer append(double d)`
- `StringBuilder/StringBuffer append(float f)`
- `StringBuilder/StringBuffer append(int i)`
- `StringBuilder/StringBuffer append(long lng)`
- `StringBuilder/StringBuffer append(Object obj)`
- `StringBuilder/StringBuffer append(String str)`

- `StringBuilder/StringBuffer append(StringBuffer sb)`

Die Methoden `append(char)`, `append(CharSequence)` und `append(CharSequence, int, int)` werden von der Schnittstelle `Appendable` vorgeschrieben.

Besonders nützlich ist in der Praxis `append(CharSequence, int, int)`, da sich auf diese Weise Teile von `String`-, `StringBuilder`- und `StringBuffer`-Objekten anhängen lassen.



Hinweis

Jede `append()`-Methode verändert den `StringBuilder/StringBuffer` und liefert als Rückgabewert noch eine Referenz darauf. Das hat den großen Vorteil, dass sich Aufrufe der `append()`-Methoden einfach hintereinandersetzen (kaskadieren) lassen:

```
StringBuilder sb = new StringBuilder( "George Peppard " ).append(',')  
sb.append(" Mr. T, ").append("Dirk Benedict, ").append("Dwight Schultz");
```

Die Auswertung erfolgt von links nach rechts, sodass das Ergebnis ist: »George Peppard, Mr. T, Dirk Benedict, Dwight Schultz«.

4.6.5 Zeichen(folgen) setzen, löschen und umdrehen

Da sich bei einem `StringBuilder/StringBuffer` Zeichen verändern lassen, gibt es neben der `append()`-Methode weitere Modifikationsmethoden, die in der Klasse `String` fehlen.

Einzelne Zeichen setzen

Neu ist `setCharAt()`, um in einem `StringBuilder/StringBuffer` an eine bestimmte Stelle ein Zeichen zu setzen.



Beispiel

Ändere das erste Zeichen im `StringBuilder` in einen Großbuchstaben:

```
StringBuilder sb = new StringBuilder( "spare Wasser und dusche mit dem Partner" );  
char upperCharacter = Character.toUpperCase( sb.charAt(0) );  
sb.setCharAt( 0, upperCharacter );
```

Das erste Argument 0 in `setCharAt()` steht für die Position des zu setzenden Zeichens.

Zeichenfolgen einfügen

Die Methode `insert(int offset, element)` fügt die Zeichenketten-Repräsentation eines Werts vom Typ Typ an der Stelle offset ein. Sie ähnelt der überladenen `append()`-Methode.

Beispiel

zB

Lies eine Datei ein, und drehe die Zeilen so um, dass die letzte Zeile der Datei oben steht und die erste Zeile der Datei unten. Das Ergebnis auf der Konsole soll ein String sein, der keinen Weißraum zu Beginn und am Ende aufweist:

Listing 4.11: ReverseFile.java, main()

```
Scanner scanner = new Scanner( ReverseFile.class.getResourceAsStream(
                            "EastOfJava.txt" ) );
StringBuilder result = new StringBuilder();
while ( scanner.hasNextLine() )
    result.insert( 0, scanner.nextLine() + "\n" );
System.out.println( result.toString().trim() );
scanner.close();
```

Für char-Arrays existiert `insert()` in einer abgewandelten Art: `insert(int index, char[] str, int offset, int len)`. Es wird nicht das komplette Array in den `StringBuilder/StringBuffer` übernommen, sondern nur ein Ausschnitt.

Einzelnes Zeichen und Zeichenbereiche löschen

Eine Folge von Zeichen lässt sich durch `delete(int start, int end)` löschen. `deleteCharAt(int index)` löscht nur ein Zeichen. In beiden Fällen wird ein inkorrekt Index durch eine `StringIndexOutOfBoundsException` bestraft.

Zeichenbereiche ersetzen

Die Methode `replace(int start, int end, String str)` löscht zuerst die Zeichen zwischen start und end und fügt anschließend den neuen String str ab start ein. Dabei sind die Endpositionen wie immer exklusiv, das heißt, sie geben das erste Zeichen hinter dem zu verändernden Ausschnitt an.

zB Beispiel

Ersetze den Teilstring an der Position 4 und 5 (also bis exklusive 6):

```
StringBuilder sb = new StringBuilder( "Sub-XX-Sens-0-Matic" );
//                                0123456
System.out.println( sb.replace( 4, 6, "Etha" ) ); // Sub-Etha-Sens-0-Matic
```

Zeichenfolgen umdrehen

Eine weitere Methode `reverse()` dreht die Zeichenfolge um.

zB Beispiel

Teste unabhängig von der Groß-/Kleinschreibung, ob der String `s` ein Palindrom ist.

Palindrome lesen sich von vorn genauso wie von hinten, etwa »Rentner«:

```
boolean isPalindrome =
    new StringBuilder( s ).reverse().toString().equalsIgnoreCase( s );
```

4.6.6 Länge und Kapazität eines `StringBuilder/StringBuffer`-Objekts *

Wie bei einem `String` lässt sich die Länge und die Anzahl der enthaltenen Zeichen mit der Methode `length()` erfragen. `StringBuilder/StringBuffer`-Objekte haben jedoch auch eine interne Puffergröße, die sich mit `capacity()` erfragen lässt und die im Konstruktor wie beschrieben festgelegt wird. In diesem Puffer, der genauer gesagt ein Array vom Typ `char` ist, werden die Veränderungen wie das Ausschneiden oder Anhängen von Zeichen vorgenommen. Während `length()` die Anzahl der Zeichen angibt, ist `capacity()` immer größer oder gleich `length()` und sagt etwas darüber aus, wie viele Zeichen der Puffer noch aufnehmen kann, ohne dass intern ein neues, größeres Feld benötigt würde.

zB Beispiel

`sb.length()` ergibt 14, aber `sb.capacity()` ergibt $14 + 16 = 30$:

```
StringBuilder sb = new StringBuilder( "www.tutego.de" );
System.out.println( sb.length() );                                // 14
System.out.println( sb.capacity() );                             // 30
```

Die Startgröße sollte mit der erwarteten Größe initialisiert werden, um ein späteres teures internes Vergrößern zu vermeiden. Falls der `StringBuilder/StringBuffer` einen großen internen Puffer hat, aber auf lange Sicht nur wenig Zeichen besitzt, lässt er sich mit `trimToSize()` auf eine kleinere Größe schrumpfen.

Ändern der Länge

Soll der `StringBuilder/StringBuffer` mehr Daten aufnehmen, so ändert `setLength()` die Länge auf eine angegebene Anzahl von Zeichen. Der Parameter ist die neue Länge. Ist sie kleiner als `length()`, so wird der Rest der Zeichenkette einfach abgeschnitten. Die Größe des internen Puffers ändert sich dadurch nicht. Ist `setLength()` größer, so vergrößert sich der Puffer, und die Methode füllt die übrigen Zeichen mit Nullzeichen '\0000' auf. Die Methode `ensureCapacity()` fordert, dass der interne Puffer für eine bestimmte Anzahl von Zeichen ausreicht. Wenn nötig, legt sie ein neues, vergrößertes char-Array an, verändert aber nicht die Zeichenfolge, die durch das `StringBuilder-/StringBuffer-Objekt` repräsentiert wird.

4.6.7 Vergleichen von String mit StringBuilder und StringBuffer

Zum Vergleichen von Zeichenketten bietet sich die bekannte `equals()`-Methode an. Diese ist aber bei `StringBuilder/StringBuffer` nicht wie erwartet implementiert. Dazu gesellen sich andere Methoden, die zum Beispiel unabhängig von der Groß-/Kleinschreibung vergleichen.

equals() bei der String-Klasse

Ein Blick in die API-Dokumentation der Klasse `String` zeigt die bekannte `equals(Object)`-Methode. Zwar erlaubt der Parametertyp durch den Basistyp `Object` beliebige Objekte (also etwa `Point`, `String`, `Date`, `StringBuilder`), doch das `equals()` von `String` vergleicht nur `String/String-Paare`. Die Methode beginnt erst dann den Vergleich, wenn das Argument auch vom Typ `String` ist. Das testet die Methode mit dem speziellen Operator `instanceof`. Das bedeutet, dass der Compiler mit dem Argumenttyp `StringBuilder/StringBuffer` bei `equals()` kein Problem hat, doch zur Laufzeit ist das Ergebnis immer `false`, da eben ein `StringBuilder/StringBuffer` nicht `instanceof String` ist. Ob die Zeichenfolgen dabei gleich sind, spielt keine Rolle.

Eine Lösung für den Vergleich von `String` mit `StringBuilder/StringBuffer` ist, zunächst mit `toString()` den `StringBuilder/StringBuffer` in einen `String` zu überführen und dann die beiden `Strings` mit `equals()` zu vergleichen.

contentEquals() beim String

Eine allgemeine Methode zum Vergleichen eines Strings mit entweder einem anderen String oder mit StringBuilder/StringBuffer ist `contentEquals(CharSequence)`. Die Methode liefert die Rückgabe `true`, wenn der String und die CharSequence (String, StringBuilder und StringBuffer sind Klassen vom Typ CharSequence) den gleichen Zeicheninhalt haben. Die interne Länge des Puffers spielt keine Rolle. Ist das Argument `null`, wird eine `NullPointerException` ausgelöst.

zB Beispiel

Vergleiche einen String mit einem StringBuffer:

```
String      s  = "Elektrisch-Zahnbürster";
StringBuffer sb = new StringBuffer( "Elektrisch-Zahnbürster" );
System.out.println( s.equals(sb) );                      // false
System.out.println( s.equals(sb.toString()) );           // true
System.out.println( s.contentEquals(sb) );                // true
```

equals() bei StringBuffer beziehungsweise StringBuilder?

Wollen wir zwei StringBuffer-beziehungsweise StringBuilder-Objekte miteinander vergleichen, werden wir noch mehr enttäuscht: Die Klassen deklarieren überhaupt keine eigene `equals()`-Methode. Es gibt zwar die übliche von `Object` geerbte Methode, doch das heißt, nur Objektreferenzen werden verglichen. Wenn also zwei verschiedene String-Builder/StringBuffer-Objekte mit gleichem Inhalt mit `equals()` verglichen werden, kommt trotzdem immer `false` heraus.

zB Beispiel

Um den inhaltlichen Vergleich von zwei StringBuilder-Objekten zu realisieren, müssen wir diese erst mit `toString()` in Strings umwandeln.

```
StringBuilder sb1 = new StringBuilder( "www.tutego.de" );
StringBuilder sb2 = new StringBuilder( "www.tutego.de" );
System.out.println( sb1.equals( sb2 ) );                  // false
System.out.println( sb1.toString().equals( sb2.toString() ) ); // true
System.out.println( sb1.toString().contentEquals( sb2 ) ); // true
```

4.6.8 hashCode() bei StringBuilder/StringBuffer *

Die obige Betrachtung zeigt, dass eine Methode equals(), die den Inhalt von StringBuilder/StringBuffer-Objekten vergleicht, nicht schlecht wäre. Dennoch besteht das Problem, wann StringBuilder/StringBuffer-Objekte als gleich angesehen werden sollen. Das ist interessant, denn StringBuilder/StringBuffer-Objekte sind nicht nur durch ihren Inhalt bestimmt, sondern auch durch die Größe ihres internen Puffers, also durch ihre Kapazität. Sollte equals() den Rückgabewert true haben, wenn die Inhalte gleich sind, oder nur dann, wenn Inhalt und Puffergröße gleich sind? Da jeder Entwickler andere Ansichten über die Gleichheit besitzt, bleibt es bei dem standardmäßigen Test auf identische Objektreferenzen.

Eine ähnliche Argumentation gilt bei der hashCode()-Methode, die für alle inhaltsgleichen Objekte denselben, im Idealfall eindeutigen Zahlenwert liefert. Die Klasse String besitzt eine hashCode()-Methode, doch StringBuilder/StringBuffer erbt die Implementierung aus der Klasse Object unverändert. Mit anderen Worten: Die Klassen selbst bieten keine Implementierung an.

4.7 CharSequence als Basistyp *

Bisher kennen wir die Klassen String, StringBuilder und StringBuffer, um Zeichenketten zu speichern und weiterzugeben. Ein String ist ein Wertobjekt und ein wichtiges Hilfsmittel in Programmen, da durch ihn unveränderliche Zeichenkettenwerte repräsentiert werden, während StringBuilder/StringBuffer veränderliche Zeichenfolgen umfassen.

Aber wie sieht es aus, wenn eine Teilzeichenkette gefordert ist, bei der es egal sein soll, ob das Original als String-, StringBuffer oder StringBuilder-Objekt vorliegt? Und was ist, wenn nur lesender Zugriff gestattet sein soll, sodass Veränderungen ausgeschlossen sind? Eine Lösung ist, alles als ein String-Objekt zu erwarten (und das macht die Java-Bibliothek auch). Doch dann müssen die Programmteile, die intern mit StringBuilder/StringBuffer arbeiten, erst einen neuen String konstruieren, und das kostet Ressourcen.

Zum Glück besitzen die Klassen String sowie StringBuilder/StringBuffer einen gemeinsamen Basistyp CharSequence. CharSequence steht für eine unveränderliche, nur lesbare Sequenz von Zeichen (Schnittstellen und Basistypen sowie Implementierungen werden präziser in Abschnitt 5.13, »Schnittstellen«, vorgestellt). Methoden müssen sich also nicht mehr für konkrete Klassen entscheiden, sondern können einfach ein CharSequence-Objekt als Argument akzeptieren oder als Rückgabe weitergeben. Ein String und ein StringBuilder/StringBuffer-Objekt können zwar mehr, als CharSequence vorschreibt,

beide lassen sich aber als CharSequence einsetzen, wenn das »Mehr« an Funktionalität nicht benötigt wird.

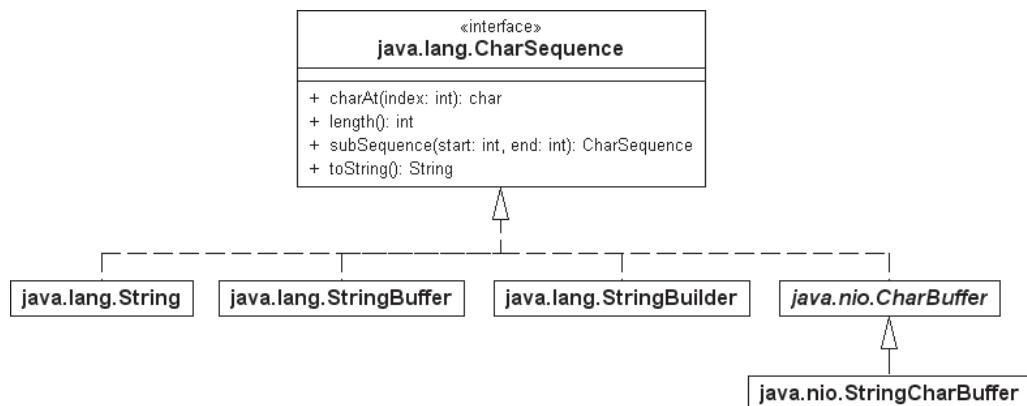


Abbildung 4.5: Einige implementierende Klassen von der Schnittstelle CharSequence

interface java.lang.CharSequence

- `char charAt(int index)`
Liefert das Zeichen an der Stelle index.
- `int length()`
Gibt die Länge der Zeichensequenz zurück.
- `CharSequence subSequence(int start, int end)`
Liefert eine neue CharSequence von start bis end.
- `String toString()`
Gibt einen String der Sequenz zurück. Die Länge des `toString()`-Strings entspricht genau der Länge der Sequenz.

zB

Beispiel

Soll eine Methode eine Zeichenkette bekommen und ist die Herkunft egal, so implementieren wir etwa

```

void giveMeAText( CharSequence s )
{
    ...
}
  
```

statt der beiden Methoden:

zB

4

Beispiel (Forts.)

```
void giveMeAText( String s )
{
    ...
}

void giveMeAText( StringBuffer s )
{
    void giveMeAText( new String(s) ); // oder Ähnliches
}
```

Anwendung von CharSequence in String

In den Klassen `String` und `StringBuilder/StringBuffer` existiert eine Methode `subSequence()`, die ein `CharSequence`-Objekt liefert. Die Signatur ist in beiden Fällen die gleiche. Die Methode macht im Prinzip nichts anderes als ein `substring(begin, end)`.

```
class java.lang.String implements CharSequence, ...
class java.lang.StringBuffer implements CharSequence, ...
class java.lang.StringBuilder implements CharSequence, ...
```

- `CharSequence subSequence(int beginIndex, int endIndex)`
Liefert eine neue Zeichensequenz von `String` beziehungsweise `StringBuffer`.

Die Implementierung sieht so aus, dass mit `substring()` ein neuer Teilstring zurückgeliefert wird. Das ist eine einfache Lösung, aber nicht unbedingt die schnellste. Für `String`-Objekte ist das Erzeugen von Substrings ziemlich schnell, da die Methode speziell optimiert ist. Da `Strings` unveränderlich sind, wird einfach das gleiche `char`-Feld wie im Original-`String` verwendet, nur eine Verschiebung und ein Längenwert werden angepasst. Zur Optimierung siehe auch den Tuning-Hinweis in Abschnitt 4.4.12.

4.8 Reguläre Ausdrücke

Ein regulärer Ausdruck (engl. *regular expression*) ist eine Beschreibung eines Musters (engl. *pattern*). Reguläre Ausdrücke werden bei der Zeichenkettenverarbeitung beim Suchen und Ersetzen eingesetzt. Für folgende Szenarien bietet die Java-Bibliothek entsprechende Unterstützung an:

- *Frage nach einer kompletten Übereinstimmung:* Passt eine Zeichenfolge komplett auf ein Muster? Wir nennen das *match*. Die Rückgabe einer solchen Anfrage ist einfach wahr oder falsch.
- *Finde Teilstrings:* Das Pattern beschreibt einen Teilstring, und gesucht sind alle Vorkommen dieses Musters in einem Suchstring.
- *Ersetze Teilstufen:* Das Pattern beschreibt Wörter, die durch andere Wörter ersetzt werden.
- *Zerlegen einer Zeichenfolge:* Das Muster steht für Trennzeichen, sodass nach dem Zerlegen eine Sammlung von Zeichenfolgen entsteht.

Ein *Pattern-Matcher* ist die »Maschine«, die reguläre Ausdrücke verarbeitet. Zugriff auf diese Mustermaschine bietet die Klasse `Matcher`. Dazu kommt die Klasse `Pattern`, die die regulären Ausdrücke in einem vorcompilierten Format repräsentiert. Beide Klassen befinden sich im Paket `java.util.regex`. Um die Sache etwas zu vereinfachen, gibt es bei `String` zwei kleine Hilfsmethoden, die im Hintergrund auf die Klassen verweisen, um eine einfachere API anbieten zu können; diese nennen sich auch *Fassaden-Methoden*.

4.8.1 `Pattern.matches()` bzw. `String#matches()`

Die statische Methode `java.util.regex.Pattern.matches()` und die Objektmethode `matches()` der Klasse `String` testen, ob ein regulärer Ausdruck eine Zeichenfolge komplett beschreibt.

Wir wollen testen, ob eine Zeichenfolge in einfache Hochkommata eingeschlossen ist:

Ausdruck	Ergebnis
<code>Pattern.matches(".*'", "'Hallo Welt'")</code>	true
<code>'Hallo Welt'".matches(".*'")</code>	true
<code>Pattern.matches(".*'", "'''")</code>	true
<code>Pattern.matches(".*'", "Hallo Welt")</code>	false
<code>Pattern.matches(".*'", "'Hallo Welt'")</code>	false

Tabelle 4.10: Einfache reguläre Ausdrücke und ihr Ergebnis

Der Punkt im regulären Ausdruck steht für ein beliebiges Zeichen, und der folgende Stern ist ein Quantifizierer, der wahllos viele beliebige Zeichen erlaubt.

Regeln für reguläre Ausdrücke

Für reguläre Ausdrücke existiert eine ganze Menge von Regeln. Während die meisten Zeichen aus dem Alphabet erlaubt sind, besitzen Zeichen wie der Punkt, die Klammer, ein Sternchen und einige weitere Zeichen Sonderfunktionen. So maskiert auch ein vorangestelltes »\« das folgende Sonderzeichen aus, was bei besonderen Zeichen wie ».<« oder »\« wichtig ist. Zunächst gilt es, die Anzahl an Wiederholungen zu bestimmen. Dazu dient ein *Quantifizierer* (auch *Wiederholungsfaktor* genannt). Drei wichtige gibt es. Für eine Zeichenkette X gilt:

Quantifizierer	Anzahl an Wiederholungen
X?	X kommt einmal oder keinmal vor.
X*	X kommt keinmal oder beliebig oft vor.
X+	X kommt einmal oder beliebig oft vor.

Tabelle 4.11: Quantifizierer im Umgang mit einer Zeichenkette X

Eine Sonderform ist X(? !Y) – das drückt aus, dass der reguläre Ausdruck Y dem regulären Ausdruck X *nicht* folgen darf (die API-Dokumentation spricht von »zero-width negative lookahead«).

Ausdruck	Ergebnis
Pattern.matches("0", "0")	true
Pattern.matches("0", "1")	false
Pattern.matches("0", "00")	false
Pattern.matches("0*", "0000")	true
Pattern.matches("0*", "01")	false
Pattern.matches("0*", "01")	false
Pattern.matches("0*", "0*")	true

Tabelle 4.12: Beispiele für reguläre Ausdrücke mit Wiederholungen

Da in regulären Ausdrücken oftmals ein Bereich von Zeichen, etwa alle Buchstaben, abgedeckt werden muss, gibt es die Möglichkeit, *Zeichenklassen* zu definieren.

Zeichenklasse	Enthält
[aeiou]	Zeichen a, e, i, o oder u
[^aeiou]	nicht die Zeichen a, e, i, o, u
[0-9a-fA-F]	Zeichen 0, 1, 2, ..., 9 oder Groß-/Klein-Buchstaben a, b, c, d, e, f

Tabelle 4.13: Definition von Zeichenklassen

Das »^« definiert *negative Zeichenklassen*, also Zeichen, die nicht vorkommen dürfen. Mit dem »-« lässt sich ein Bereich von Zeichen angeben.

Listing 4.12: RegExDemo.java, main(), Ausschnitt

```
System.out.println( Pattern.matches( "[01]*", "0" ) );           // true
System.out.println( Pattern.matches( "[01]*", "01001" ) );        // true
System.out.println( Pattern.matches( "[0123456789]*", "112" ) ); // true
```

Daneben gibt es vordefinierte Zeichenklassen, die in erster Linie Schreibarbeit ersparen. Die wichtigsten sind:

Zeichenklasse	Enthält
.	jedes Zeichen
\d	Ziffer: [0-9]
\D	keine Ziffer: [^0-9] beziehungsweise [^\d]
\s	Weißraum: [\t\n\x0B\f\r]
\S	keinen Weißraum: [^\s]
\w	Wortzeichen: [a-zA-Z_0-9]
\W	keine Wortzeichen: [^\w]
\p{Blank}	Leerzeichen oder Tab: [\t]
\p{Lower}, \p{Upper}	einen Klein-/Großbuchstaben: [a-z] beziehungsweise [A-Z]
\p{Alpha}	einen Buchstaben: [\p{Lower}\p{Upper}]
\p{Alnum}	ein alphanumerisches Zeichen: [\p{Alpha}\p{Digit}]
\p{Punct}	ein Punkt-Zeichen: !"#\$%&'()*+,.-/:;<=>?@[\'`{ }~
\p{Graph}	ein sichtbares Zeichen: [\p{Alnum}\p{Punct}]
\p{Print}	ein druckbares Zeichen: [\p{Graph}]

Tabelle 4.14: Vordefinierte Zeichenklassen

Bei den Wortzeichen handelt es sich standardmäßig um die ASCII-Zeichen und nicht um deutsche Zeichen mit unseren Umlauten oder allgemeine Unicode-Zeichen. Eine umfassende Übersicht liefert die API-Dokumentation der Klasse `java.util.regex.Pattern`.

Listing 4.13: RegExDemo.java, main(), Ausschnitt

```
System.out.println( Pattern.matches( "\\\d*", "112" ) );           // true
System.out.println( Pattern.matches( "\\\d*", "112a" ) );          // false
System.out.println( Pattern.matches( "\\\d*.", "112a" ) );         // true
System.out.println( Pattern.matches( ".\\\\d*.", "x112a" ) );       // true
```

Tipp

Die Methode `contains()` der `String`-Klasse testet nur Teilzeichenfolgen, aber diese Zeichenfolge ist kein regulärer Ausdruck (sonst würde so etwas wie `contains(".")` auch eine völlig andere Bedeutung haben). Wer ein `s.contains("pattern")` sucht, kann es als `s.matches(".*pattern.*")` umschreiben.



4.8.2 Die Klassen Pattern und Matcher

Der Aufruf der Objektmethode `matches()` auf einem `String`-Objekt beziehungsweise das statische `Pattern.matches()` ist nur eine Abkürzung für die Übersetzung eines Patterns und Anwendung von `matches()`:

<code>String#matches()</code>	<code>Pattern.matches()</code>
<pre>public boolean matches(String regex) { return Pattern.matches(regex, this); }</pre>	<pre>public static boolean matches(String regex, CharSequence input) { Pattern p = Pattern.compile(regex); Matcher m = p.matcher(input); return m.matches(); }</pre>

Tabelle 4.15: Implementierungen der beiden `matches()`-Methoden

Während die `String`-Mitläufer-Methode `matches()` zur `Pattern.matches()` delegiert, steht hinter der statischen Fassadenmethode `Pattern.matches()` die wirkliche Nutzung der beiden zentralen Klassen `Pattern` für das Muster und `Matcher` für die Mustermaschine.

Für unser erstes Beispiel `Pattern.matches("'.*'", "'Hallo Welt'")` hätten wir also äquivalent schreiben können:

```
Pattern p = Pattern.compile("'.*'");
Matcher m = p.matcher( "'Hallo Welt'" );
boolean b = m.matches();
```

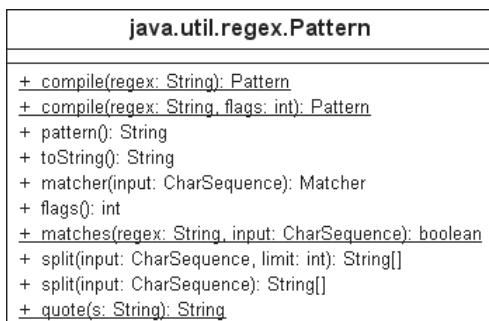


Abbildung 4.6: UML-Diagramm von Pattern



Hinweis

Bei mehrmaliger Anwendung des gleichen Patterns sollte es compiliert gecacht werden, denn das immer wieder nötige Übersetzen über die Objektmethode `String#matches()` beziehungsweise die Klassenmethode `Pattern.matches()` kostet Speicher und Laufzeit.

```
final class java.util.regex.Pattern
    implements Serializable
```

- static Pattern compile(String regex)
Übersetzt den regulären Ausdruck in ein Pattern-Objekt.
- static Pattern compile(String regex, int flags)
Übersetzt den regulären Ausdruck in ein Pattern-Objekt mit Flags. Als Flags sind CASE_INSENSITIVE, MULTILINE, DOTALL, UNICODE_CASE und CANON_EQ erlaubt. In Java 7 kommt UNICODE_CHARACTER_CLASS hinzu.
- int flags()
Liefert die Flags, nach denen geprüft wird.
- Matcher matcher(CharSequence input)
Liefert ein Matcher-Objekt, das prüft.

- static boolean matches(String regex, CharSequence input)
Liefert true, wenn der reguläre Ausdruck regex auf die Eingabe passt.
- static Stringquote(String s)
Maskiert die Metazeichen/Escape-Sequenzen aus. So liefert Pattern.quote("*.[\d]") den String \Q*.[\d]\E.
- String pattern()
Liefert den regulären Ausdruck, den das Pattern-Objekt repräsentiert.

Pattern-Flags *

Die Flags sind in speziellen Situationen ganz hilfreich, etwa wenn die Groß-/Kleinschreibung keine Rolle spielt oder sich die Suche über eine Zeile erstrecken soll. Doch Java zwingt uns nicht, die Pattern-Klasse zu nutzen, um die Flags einsetzen zu können, sondern erlaubt es, mit einer speziellen Schreibweise die Flags auch im regulären Ausdruck selbst anzugeben, was die Nutzung bei String#matches() ermöglicht.

Flag in der Pattern-Klasse	Eingebetteter Flag-Ausdruck
Pattern.CASE_INSENSITIVE	(?i)
Pattern.COMMENTS	(?x)
Pattern.MULTILINE	(?m)
Pattern.DOTALL	(?s)
Pattern.UNICODE_CASE	(?u)
Pattern.UNICODE_CHARACTER_CLASS	(?U)
Pattern.UNIX_LINES	(?d)

Tabelle 4.16: Pattern-Flags

In einem regulären Ausdruck sind die Varianten rechts sehr praktisch, da sie an unterschiedlichen Positionen ein- und ausgeschaltet werden können. Ein nach dem Fragezeichen platziertes Minus stellt die Option wieder ab, etwa "(?i)jetzt insensitive(?-i) wieder sensitive". Mehrere Flag-Ausdrücke lassen sich auch zusammensetzen, etwa zu "(?ims)".

In der Praxis häufiger im Einsatz sind Pattern.DOTALL/(?s), Pattern.CASE_INSENSITIVE/(?i) und Pattern.MULTILINE/(?m). Es folgen Beispiele, wobei wir MULTILINE bei den Wortgrenzen vorstellen.

Standardmäßig matcht der ».<« kein Zeilenendezeichen, sodass ein regulärer Ausdruck einen Zeilenumbruch nicht erkennt. Das lässt sich mit `Pattern.DOTALL`-Flag beziehungsweise `(?s)` ändern.

zB

Beispiel

Die Auswirkung vom DOTALL beziehungsweise `(?s)`:

```
System.out.println( "wau wau miau".matches( "wau.+wau.*" ) );      // true
System.out.println( "wau\nwau miau".matches( "wau.+wau.*" ) );      // false
System.out.println( "wau wau miau".matches( "(?s)wau.+wau.*" ) ); // true
System.out.println( "wau\nwau miau".matches( "(?s)wau.+wau.*" ) ); // true
```

Quantifizierer und Wiederholungen *

Neben den Quantifizierern `?` (einmal oder keinmal), `*` (keinmal oder beliebig oft) und `+` (einmal oder beliebig oft) gibt es drei weitere Quantifizierer, die es erlauben, die Anzahl eines Vorkommens genauer zu beschreiben:

- `X{n}`. X muss genau n -mal vorkommen.
- `X{n,}`. X kommt mindestens n -mal vor.
- `X{n,m}`. X kommt mindestens n -, aber maximal m -mal vor.

zB

Beispiel

Eine E-Mail-Adresse endet mit einem Domain-Namen, der 2 oder 3 Zeichen lang ist:

```
Static Pattern p = Pattern.compile( "[\w-]+@\w[\w-]*\.[a-z]{2,3}" );
```

Ränder und Grenzen testen *

Die bisherigen Ausdrücke waren nicht ortsgebunden, sondern haben geprüft, ob es irgendwo im String eine Übereinstimmung gibt. Dateiendungen zum Beispiel sind aber – wie der Name schon sagt – am Ende zu prüfen, genauso wie ein URL-Protokoll wie »`http://`« am Anfang stehen muss. Um diese Anforderungen mit berücksichtigen zu können, können bestimmte Positionen mit in einem regulären Ausdruck gefordert werden. Die Pattern-API erlaubt folgende Matcher:

Matcher	Bedeutung
^	Beginn einer Zeile
\$	Ende einer Zeile
\b	Wortgrenze
\B	Keine Wortgrenze
\A	Beginn der Eingabe
\Z	Ende der Eingabe ohne Zeilenabschlusszeichen wie \n oder \r
\z	Ende der Eingabe mit allen Zeilenabschlusszeichen
\G	Ende des vorherigen Matches. Sehr speziell für iterative Suchvorgänge

Tabelle 4.17: Erlaubte Matcher

Wichtig ist zu verstehen, dass diese Matcher keine »Breite« haben, also nicht wirklich ein Zeichen oder eine Zeichenfolge matchen, sondern lediglich die Position beschreiben.

Die Matcher ^ und \$ lösen gut das Problem mit den Dateiendungen und HTTP-Protokollen und leisten gute Dienste bei bestimmten Löschanweisungen.

Beispiel

zB

Die String-Methode trim() schneidet den Weißraum vorne und hinten ab. Mit replaceAll() und den Matchern für den Beginn und das Ende einer Zeile ist schnell ein Ausdruck gefunden, der nur den Weißraum vorne oder nur hinten entfernt:

```
String s = " \tWo ist die Programmiersprache des Lächelns?\t\t ";
String ltrim = s.replaceAll( "^\\s+", "" );
String rtrim = s.replaceAll( "\\s+$", "" );
System.out.printf( "%s%n", ltrim ); // 'Wo ist die Programmiersprache des
// Lächelns? '
System.out.printf( "%s%n", rtrim ); // ' Wo ist die Programmiersprache des
// Lächelns?' 
```

Der Matcher \b ist nützlich, wenn es darum geht, ein Wort umrandet von Weißraum in einer Teilzeichenkette zu finden. In der Zeichenkette »Spaß in China innerhalb der Grenzen« wird die Suche nach »in« drei Fundstellen ergeben, aber \bin\b nur eine und \bin\B auch eine, und zwar »innerhalb«. Es matcht demnach ein \b genau die Stelle, bei der ein \w auf ein \W folgt (beziehungsweise andersherum).

Multiline-Modus *

Normalerweise sind `^` und `$` nicht zeilenorientiert, das heißt, es ist ihnen egal, ob im String Zeilenumbruchzeichen wie `\n` oder `\r` vorkommen oder nicht. Mitunter soll der Test aber lokal auf einer Zeile stattfinden – hierzu muss der Multiline-Modus aktiviert werden.

zB

Beispiel

Teste, ob eine E-Mail die Zeile »Hi,« enthält:

```
System.out.println( "Hi,".matches( ".^Hi,$.*" ) );
System.out.println( "Fwd:\nHi,mir geht's gut!".matches( ".^Hi,$.*" ) );
System.out.println( "Fwd:\nHi,\nmir geht's gut!".matches( "(?sm).^Hi,$.*" ) );
```

Der Test auf `".^Hi,$.*"` gibt im ersten Fall `true` zurück, da der String wirklich matcht und wir auch überhaupt keinen Zeilentrenner haben, der uns Probleme bereiten könnte. Die zweite Zeile aber liefert `false`, da sie global mit »Fwd« und nicht mit »Hi« beginnt und mit »!« endet statt mit einem Komma. Führen wir den Test mit der Option `(?sm)` zeilenweise durch und überspringen wir die Zeilentrenner, dann ist das Ergebnis `true`, denn die 2. Zeile in

Fwd:

Hi,

mir geht's gut!

passt genau auf unseren regulären Ausdruck.

Der Multiline-Modus erklärt auch den Grund, warum es gleich mehrere Grenz-Matcher gibt. Die Matches `\A` und `\Z` beziehungsweise `\z` sind im Prinzip wie `^` und `$`, unterscheiden sich aber dann, wenn der Multiline-Modus aktiviert ist. Dann arbeiten (wie im Beispiel) `^` und `$` zeilenorientiert, `\A` und `\Z` beziehungsweise `\z` aber nie – die letzten drei Matcher kennen Zeilentrenner überhaupt nicht. Damit ist `"Fwd:\nHi,\nalles OK!".matches("(?sm).*\\"AHi,\\"Z.*")` auch trotz `(?sm)` ganz einfach `false`.

Es bleiben `\z` und `\Z`. Sie unterscheiden, ob bei Zeilen, die abschließende Zeilentrenner wie `\n` oder `\r` besitzen, diese Zeilentrenner mit zum Match gehören oder nicht. Das `\z` ist wie `$` ein Matcher auf das absolute Ende inklusive aller Zeilentrenner. Das große `\Z` ignoriert am Ende stehende Zeilentrenner, sodass sozusagen der Match schon vorher zu Ende ist.

zB

4

Beispiel

Das Trennzeichen beim split() soll einmal \z und einmal \Z sein:

```
String[] tokens1 = "Lena singt\r\n".split( "\\z" );
String[] tolens2 = "Lena singt\r\n".split( "\\Z" );
System.out.printf( "%d %s%n", tokens1.length, Arrays.toString( tokens1 ) );
System.out.printf( "%d %s%n", tolens2.length, Arrays.toString( tolens2 ) );
```

Bei \z gehören alle Zeilentrener zum String, und daher ist die Ausgabe:

```
1 [Lena singt
]
```

Die zweite Ausgabe ist:

```
2 [Lena singt,
]
```

Und die abschließenden Zeilentrener sind ein zweites Token.

4.8.3 Finden und nicht matchen

Bisher haben wir mit regulären Ausdrücken lediglich festgestellt, ob eine Zeichenfolge vollständig auf ein Muster passt. Die Matcher-Klasse kann jedoch auch feststellen, ob sich eine durch ein Muster beschriebene Teilfolge im String befindet. Dazu dient die Methode find(). Sie hat zwei Aufgaben: Zunächst sucht sie nach einer Fundstelle und gibt bei Erfolg true zurück. Das Nächste ist, dass jedes Matcher-Objekt einen Zustand mit Fundstellen besitzt, den find() aktualisiert. Einem Matcher-Objekt entlockt die Methode group() den erkannten Substring, und start()/end() liefert die Positionen. Wiederholte Aufrufe von find() setzen die Positionen weiter:

Listing 4.14: RegExAllNumbers.java, main()

```
String s = "'Demnach, welcher verheiratet, der tut wohl; welcher aber " +
          "nicht verheiratet, der tut besser.' 1. Korinther 7, 38";
Matcher matcher = Pattern.compile( "\\d+" ).matcher( s );
while ( matcher.find() )
    System.out.printf( "%s an Position [%d,%d]%
                      , matcher.group(),
                      matcher.start(), matcher.end() );
```

Die Ausgabe des Zahlenfinders ist:

```
1 an Position [94,95]
7 an Position [107,108]
38 an Position [110,112]
```

zB Beispiel

Da es in der String-Klasse zwar ein `contains()`, aber kein `containsIgnoreCase()` gibt, lässt sich für diesen Zweck entweder ein Ausdruck wie `s1.toLowerCase().contains(s2.toLowerCase())` formen oder ein Pattern-Flag verwenden:

```
String s1 = "Prince Michael I, Paris, Prince Michael II (Blanket)";
String s2 = "PARIS";
boolean in = Pattern.compile( Pattern.quote( s2 ),
                             Pattern.CASE_INSENSITIVE ).matcher( s1 ).find();
System.out.println( in );                                // true
```

4.8.4 Gierige und nicht gierige Operatoren *

Die drei Operatoren `?`, `*` und `+` haben die Eigenschaft, die längste mögliche Zeichenfolge abzudecken – das nennt sich *gierig* (engl. *greedy*). Deutlich wird diese Eigenschaft bei dem Versuch, in einem HTML-String alle fett gesetzten Teile zu finden. Gesucht ist also ein Ausdruck, der im String

```
String string = "Echt <b>fett</b>. <b>Cool</b>!";
```

die Teilstufen `fett` und `Cool` erkennt. Der erste Versuch für ein Programm könnte so aussehen:

```
Pattern pattern = Pattern.compile( "<b>.*</b>" );
Matcher matcher = pattern.matcher( string );
while ( matcher.find() )
    System.out.println( matcher.group() );
```

Nun ist die Ausgabe aber `fett. Cool!` Das verwundert nicht, denn mit dem Wissen, dass `*` gierig ist, passt `.*` auf die Zeichenkette vom ersten `` bis zum letzten ``.

Die Lösung ist der Einsatz eines *nicht gierigen Operators* (auch *genügsam*, *zurückhaltend*, *non-greedy* oder *reluctant* genannt). In diesem Fall wird hinter den Qualifizierer einfach ein Fragezeichen gestellt.

Gieriger Operator	Nicht gieriger Operator
X?	X??
X*	X*?
X+	X+?
X{n}	X{n}?
X{n,}	X{n,}?
X{n,m}	X{n,m}?

Tabelle 4.18: Gierige und nicht gierige Operatoren

Mit diesem nicht gierigen Operator lösen wir einfach das Fettproblem:

Listing 4.15: RegExFindBold.java, main()

```
Pattern pattern = Pattern.compile( "<b>.*?</b>" );
Matcher matcher = pattern.matcher( "Echt <b>fett</b>. <b>Cool</b>!" );
while ( matcher.find() )
    System.out.println( matcher.group() );
```

Wie gewünscht ist die Ausgabe:

```
<b>fett</b>
<b>Cool</b>
```

4.8.5 Mit MatchResult alle Ergebnisse einsammeln *

Die Schnittstelle `java.util.regex.MatchResult` deklariert Operationen, die Zugriff auf das Ergebnis (`String`, `Startposition`, `Endposition`, `Anzahl der Gruppen`) eines Matches ermöglichen. Ein `Matcher`-Objekt wird dafür mit `toMatchResult()` nach dem `MatchResult`-Objekt gefragt.

Ein einfaches Beispiel verdeutlicht die Arbeitsweise: Die eigene statische Utility-Methode `findMatches()` soll für ein Muster und eine Zeichenkette alle Ergebnisse zurückliefern:

Listing 4.16: MatchResultDemo.java, Teil 1

```
static Iterable<MatchResult> findMatches( String pattern, CharSequence s )
{
    List<MatchResult> results = new ArrayList<MatchResult>();

    for ( Matcher m = Pattern.compile(pattern).matcher(s); m.find(); )
        results.add( m.toMatchResult() );

    return results;
}
```

**Abbildung 4.7:** Die Matcher-Klasse implementiert die MatchResult-Schnittstelle.

Die Methode liefert ein einfaches Iterable zurück, was in unserem Beispiel ausreicht, um die Methode auf der rechten Seite des Doppelpunktes vom erweiterten for nutzen zu können. Vor dem Schleifendurchlauf übersetzt compile() den Muster-String in ein Pattern-Objekt, und matcher() gibt Zugang zum konkreten Mustererkenner, also Matcher-Objekt. Die Bedingung der Schleife ist so, dass pro Durchlauf ein Muster erkannt wird. Im Rumpf der Schleife sammelt die Ergebnisliste die MatchResult-Objekte, die die Funddaten repräsentieren. Nach Ablauf der Schleife liefert die Methode die gesammelten Objekte zurück.

Ein paar Programmzeilen zeigen schnell die Möglichkeiten. Ein einfaches Muster soll für ISBN-10-Nummern stehen – ohne Leerzeichen oder Bindestriche:

Listing 4.17: MatchResultDemo.java, Teil 2

```
String pattern = "\\d{9,10}[\\d|x|X]";  
String s = "Insel: 3898425266, Reguläre Ausdrücke: 3897213494";  
  
for ( MatchResult r : findMatches( pattern, s ) )  
    System.out.println( r.group() + " von " + r.start() + " bis " + r.end() );
```

Das Ergebnis auf der Konsole ist:

```
3898425266 von 7 bis 17  
3897213494 von 39 bis 49
```

Die Informationen in einem MatchResult entsprechen also einem Zustand eines Matcher während des Parsens, genauer gesagt nach dem Erkennen einer Zeichenfolge. Daher implementiert auch die Klasse Matcher die Schnittstelle MatchResult.

4.8.6 Suchen und Ersetzen mit Mustern

Von der Pattern/Matcher-Klasse haben wir bisher zwei Eigenschaften kennengelernt: zum einen, wie sie prüft, ob eine komplette Zeichenkette auf ein Muster passt, und zum anderen die Suchmöglichkeit, dass find() uns sagt, an welchen Stellen ein Muster in einer Zeichenkette vorkommt. Für den zweiten Fall gibt es noch eine Erweiterung, dass nämlich die Pattern-Klasse die Fundstellen nicht nur ermittelt, sondern sie auch durch etwas anderes ersetzen kann.

zB Beispiel

In einem String sollen alle Nicht-JVM-Sprachen ausgepielt werden:

```
String text = "Ich mag Java, Groovy und auch ObjectiveC und PHP.";
Matcher matcher = Pattern.compile("ObjectiveC|PHP").matcher( text );
StringBuffer sb = new StringBuffer();
while ( matcher.find() )
    matcher.appendReplacement( sb, "[PIEP]" );
matcher.appendTail( sb );
System.out.println( sb ); // Ich mag Java, Groovy und auch [PIEP] und [PIEP].
```

Um mit dem Mechanismus »Suchen und Ersetzen« zu arbeiten, wird zunächst ein StringBuffer aufgebaut, denn in dem echten String kann Pattern die Fundstellen nicht ersetzen. Erkennt der Matcher ein Muster, ersetzt appendReplacement() es durch eine Alternative, die in den StringBuffer kommt. So wächst der StringBuffer von Schritt zu Schritt. Nach der letzten Fundstelle setzt appendTail() das noch verbleibende Teilstück an den StringBuffer.

Toll an appendReplacement() ist, dass die Ersetzung nicht einfach nur ein einfacher String ist, sondern dass er mit \$ Zugriff auf die Suchgruppe hat. Damit lassen sich sehr elegante Lösungen bauen. Nehmen wir an, wir müssen in einer Zeichenkette alle URLs in HTML-Hyperlinks konvertieren. Dann rahmen wir einfach jede Fundstelle in die nötigen HTML-Tags ein. In Quellcode sieht das so aus:

Listing 4.18: RegExSearchAndReplace.java, main()

```
String text = "Hi, schau mal bei http://stackoverflow.com/ " +
             "oder http://www.tutego.de/ vorbei.";
String regex = "http://[a-zA-Z0-9\-\\.]+\.\.[a-zA-Z]{2,3}(\s*)?";
Matcher matcher = Pattern.compile( regex ).matcher( text );
StringBuffer sb = new StringBuffer( text.length() );

while ( matcher.find() )
    matcher.appendReplacement( sb, "<a href=\"$0\">$0</a>" );
    matcher.appendTail( sb );

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

Der StringBuffer enthält dann zum Schluss "Hi, schau mal bei http://stackoverflow.com/ oder http://www.tutego.de/ vorbei." (Der gewählte reguläre Ausdruck für URLs ist kurz, aber nicht vollständig. Für das Beispiel spielt das aber keine Rolle.)

Hinweis

Der Ersetzungsausdruck "\$0" enthält mit \$ Steuerzeichen für den Matcher. Wenn die Ersetzung aber überhaupt nicht mit \$n auf das gefundene Wort zurückgreift, sollten die beiden Sonderzeichen \ und \$ ausmaskiert werden. Auf diese Weise werden merkwürdige Fehler vermeiden, wenn doch in der Ersetzung ein Dollar oder Backslash vorkommt. Das Ausmaskieren übernimmt die Methode quoteReplacement(), sodass sich zum Beispiel Folgendes ergibt:

```
matcher.appendReplacement( sb, Matcher.quoteReplacement( replacement ) );
```

4.8.7 Hangman Version 2

Mit regulären Ausdrücken lässt sich eine ganz spezielle Aufgabe unseres Hangman-Spiels noch verbessern. Wir hatten die Aufgabe, dass ungeratene Zeichen durch einen Unterstrich ersetzt werden. Diese Ersetzung kann sehr gut replaceAll() übernehmen.

Listing 4.19: Hangman2.java

```
import java.util.*;  
  
public class Hangman2  
{  
    public static void main( String[] args )  
    {  
        List<String> hangmanWords = Arrays.asList( "samoa", "tonga", "fiji", "vanuatu" );  
        Collections.shuffle( hangmanWords );  
  
        String hangmanWord = hangmanWords.get( 0 );  
        String usedChars = "";  
        String guessedWord = hangmanWord.replaceAll( ".", "_" );
```

```
for ( int guesses = 1; ; )
{
    if ( guesses == 10 )
    {
        System.out.printf( "Nach 10 Versuchen ist jetzt Schluss. Sorry! Apropos,
                           das Wort war '%s'.", hangmanWord );
        break;
    }

    System.out.printf( "Runde %d. Bisher geraten: %s. Was wählst du für ein
                       Zeichen?%n", guesses, guessedWord );
    char c = new java.util.Scanner( System.in ).next().charAt( 0 );
    if ( usedChars.indexOf( c ) >= 0 )
    {
        System.out.printf( "%c hast du schon mal getippt!%n", c );
        guesses++;
    }
    else // Zeichen wurde noch nicht benutzt
    {
        usedChars += c;
        if ( hangmanWord.indexOf( c ) >= 0 )
        {
            guessedWord = hangmanWord.replaceAll( "[" + usedChars + "]", "_" );
            if ( guessedWord.contains( "_" ) )
                System.out.printf( "Gut geraten, '%s' gibt es im Wort. Aber es
                                   fehlt noch was!%n", c );
        }
        else
        {
            System.out.printf( "Gratulation, du hast das Wort '%s' erraten!",
                               hangmanWord );
            break;
        }
    }
}
```

```
        else // hangmanWord.indexOf( c ) == -1
    {
        System.out.printf( "Pech gehabt, %c kommt im Wort nicht vor!\n", c );
        guesses++;
    }
}
}
}
}
```

4.9 Zerlegen von Zeichenketten

Die Java-Bibliothek bietet einige Klassen und Methoden, um nach bestimmten Mustern große Zeichenketten in kleinere zu zerlegen. In diesem Kontext sind die Begriffe *Token* und *Delimiter* zu nennen: Ein Token ist ein Teil eines Strings, der durch bestimmte Trennzeichen (engl. *delimiter*) von anderen Tokens getrennt wird. Nehmen wir als Beispiel den Satz »Moderne Musik ist Instrumentespielen nach Noten« (Peter Sellers). Wählen wir Leerzeichen als Trennzeichen, lauten die einzelnen Tokens »Moderne«, »Musik« und so weiter.

Die Java-Bibliothek bietet eine Reihe von Möglichkeiten zum Zerlegen von Zeichenfolgen, von denen einige in den nachfolgenden Abschnitten vorgestellt werden:

- **split()** von `String`: Aufteilen mit einem Delimiter, der durch reguläre Ausdrücke beschrieben wird.
- `Scanner`: Schöne Klasse zum Ablauen einer Eingabe.
- `StringTokenizer`: Der Klassiker aus Java 1.0. Delimiter sind nur einzelne Zeichen.
- `BreakIterator`: Findet Zeichen-, Wort-, Zeilen- oder Satz-Grenzen.

4.9.1 Splitten von Zeichenketten mit `split()`

Die Objektmethode `split()` eines `String`-Objekts zerlegt die eigene Zeichenkette in Teilzeichenketten. Die Trenner sind völlig frei wählbar und als regulärer Ausdruck beschrieben. Die Rückgabe ist ein Feld der Teilzeichenketten.

zB Beispiel

Zerlege einen Domain-Namen in seine Bestandteile:

```
String path = "www.tutego.com";
String[] segs = path.split( Pattern.quote( "." ) );
System.out.println( Arrays.toString(segs) ); // [www, tutego, com]
```

Da der Punkt als Trennzeichen ein Sonderzeichen für reguläre Ausdrücke ist, muss er passend mit dem Backslash auskommentiert werden. Das erledigt die statische Methode `quote()`. Andernfalls liefert `split(".")` auf jedem String ein Feld der Länge 0.

Ein häufiger Trenner ist `\s`, also Weißraum.

zB Beispiel

Zähle die Anzahl der Wörter in einem Satz:

```
String string = "Hört es euch an, denn das ist mein Gedudel!";
int nrOfWords = string.split( "(\\s|\\p{Punct})+" ).length;
System.out.println( nrOfWords ); // 9
```

Der Trenner ist entweder Weißraum oder ein Satzzeichen.

String.split() geht auf Pattern#split()

Die `split()`-Methode aus der `String`-Klasse delegiert wie auch bei `match()` an das `Pattern`-Objekt:

```
public String[] split( String regex, int limit )
{
    return Pattern.compile( regex ).split( this, limit );
}
public String[] split( String regex )
{
    return split( regex, 0 );
}
```

Am Quellcode ist zu erkennen, dass für jeden Methodenaufruf von `split()` auf dem `String`-Objekt ein Pattern übersetzt wird. Das ist nicht ganz billig, und so soll bei mehrmaligem Split mit dem gleichen Zerlege-Muster gleich ein Pattern-Objekt und dort das `split()` verwendet werden:

```
final class java.lang.String
    implements CharSequence, Comparable<String>, Serializable
```

- `String[] split(String regex)`
Zerlegt die aktuelle Zeichenkette mit dem regulären Ausdruck.
- `String[] split(String regex, int limit)`
Zerlegt die aktuelle Zeichenkette mit dem regulären Ausdruck, liefert jedoch maximal begrenzt viele Teilzeichenfolgen.

```
final class java.util.regex.Pattern
    implements Serializable
```

- `String[] split(CharSequence input)`
Zerlegt die Zeichenfolge `input` in Teilzeichenketten, wie es das aktuelle Pattern-Objekt befiehlt.
- `String[] split(CharSequence input, int limit)`
Wie `split(CharSequence)`, doch nur höchstens `limit` viele Teilzeichenketten.

4.9.2 Die Klasse Scanner

Die Klasse `java.util.Scanner` kann eine Zeichenkette in Tokens zerlegen und einfach Daten zeilenweise einlesen. Bei der Zerlegung kann ein regulärer Ausdruck den Delimiter beschreiben. Damit ist Scanner flexibler als ein StringTokenizer, der nur einzelne Zeichen als Trenner zulässt.

Zum Aufbau der Scanner-Objekte bietet die Klasse einige Konstruktoren an, die die zu zerlegenden Zeichenfolgen unterschiedlichen Quellen entnehmen, etwa einem String, einem Datenstrom (beim Einlesen von der Kommandozeile wird das `System.in` sein), einem File-Objekt oder diversen NIO-Objekten. Falls ein Objekt vom Typ Closeable dahintersteckt, wie ein `Writer`, sollte mit `close()` der Scanner geschlossen werden, der das `close()` zum Closeable weiterleitet. Beim String ist das nicht nötig, und bei File schließt der Scanner selbstständig.

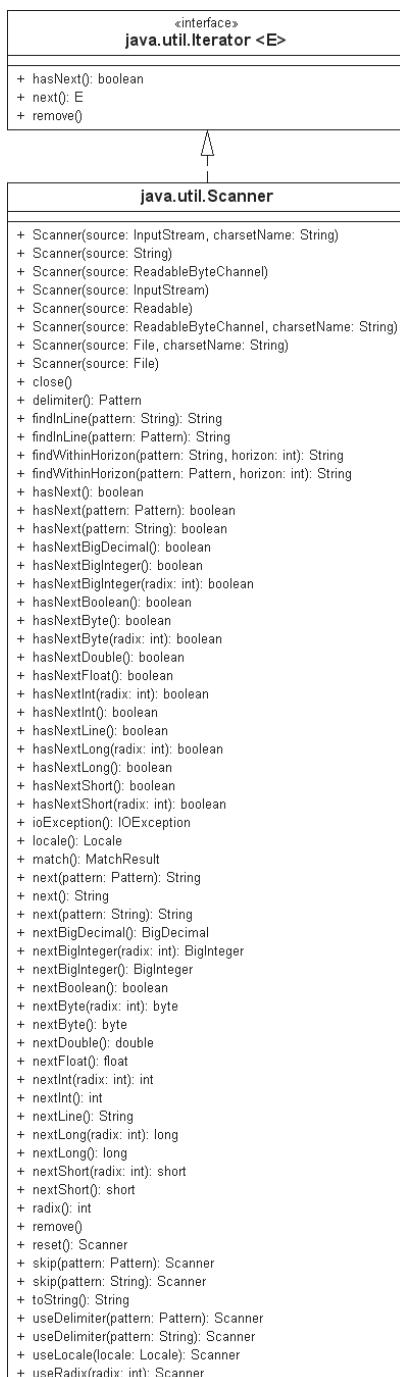


Abbildung 4.8: UML-Diagramm der Scanner-Klasse

```
final class java.util.Scanner  
implements Iterator<String>, Closeable
```

- Scanner(String source)
 - Scanner(File source)
 - Scanner(File source, String charsetName)
 - Scanner(Path source)
 - Scanner(Path source, String charsetName)
 - Scanner(InputStream source)
 - Scanner(InputStream source, String charsetName)
 - Scanner(Readable source)
 - Scanner(ReadableByteChannel source)
 - Scanner(ReadableByteChannel source, String charsetName)
- Erzeugt ein neues Scanner-Objekt aus diversen Quellen.**

Zeilenweises Einlesen einer Datei

Ist das Scanner-Objekt angelegt, lässt sich mit dem Paar `hasNextLine()` und `nextLine()` einfach eine Datei zeilenweise auslesen:

Listing 4.20: ReadAllLines.java

```
import java.io.*;  
import java.util.Scanner;  
  
public class ReadAllLines  
{  
    public static void main( String[] args ) throws FileNotFoundException  
    {  
        Scanner scanner = new Scanner( new File("EastOfJava.txt") );  
        while ( scanner.hasNextLine() )  
            System.out.println( scanner.nextLine() );  
        scanner.close();  
    }  
}
```

Da der Konstruktor von Scanner mit der Datei eine Ausnahme auslösen kann, müssen wir diesen möglichen Fehler behandeln. Wir machen es uns einfach und leiten einen möglichen Fehler an die Laufzeitumgebung weiter. Den Umgang mit Exceptions erklärt das gleichnamige Kapitel 6 genauer. Auch sollte immer die Kodierung angegeben werden, doch auch das sparen wir uns für das kleine Beispiel.

```
final class java.util.Scanner
    implements Iterator<String>, Closeable
```

- `boolean hasNextLine()`
Liefert true, wenn eine nächste Zeile gelesen werden kann.
- `String nextLine()`
Liefert die nächste Zeile.

Der Nächste, bitte

Nach dem Erzeugen des Scanner-Objekts liefert die Methode `next()` die nächste Zeichenfolge, wenn denn ein `hasNext()` die Rückgabe `true` ergibt. (Das sind dann auch die Methoden der Schnittstelle `Iterator`, wobei `remove()` nicht implementiert ist.)

zB Beispiel

Von der Standardeingabe soll ein String gelesen werden:

```
Scanner scanner = new Scanner( System.in );
String s = scanner.next();
```

Neben der `next()`-Methode, die nur einen String als Rückgabe liefert, bietet Scanner diverse `next<Typ>()`-Methoden an, die das nächste Token einlesen und in ein gewünschtes Format konvertieren, etwa in ein `double` bei `nextDouble()`. Über gleich viele `hasNext<Typ>()`-Methoden lässt sich erfragen, ob ein weiteres Token von diesem Typ folgt.

zB Beispiel

Die einzelnen `nextXXX()`- und `hasNextXXX()`-Methoden in einem Beispiel:

Listing 4.21: ScannerDemo.java, main()

```
Scanner scanner = new Scanner( "tutego 12 1973 12,03 True 123456789000" );
System.out.println( scanner.hasNext() );           // true
System.out.println( scanner.next() );             // tutego
```

zB

4

Beispiel (Forts.)

```
System.out.println( scanner.hasNextByte() );      // true
System.out.println( scanner.nextByte() );          // 12
System.out.println( scanner.hasNextInt() );        // true
System.out.println( scanner.nextInt() );           // 1973
System.out.println( scanner.hasNextDouble() );     // true
System.out.println( scanner.nextDouble() );         // 12.03
System.out.println( scanner.hasNextBoolean() );    // true
System.out.println( scanner.nextBoolean() );        // true
System.out.println( scanner.hasNextLong() );        // true
System.out.println( scanner.nextLong() );           // 123456789000
System.out.println( scanner.hasNext() );            // false
```

Sind nicht alle Tokens interessant, überspringt Scanner skip(Pattern pattern) beziehungsweise Scanner skip(String pattern) sie – Delimiter werden nicht beachtet.

```
final class java.util.Scanner
    implements Iterator<String>, Closeable
```

- boolean hasNext()
- boolean hasNextBigDecimal()
- boolean hasNextBigInteger()
- boolean hasNextBigInteger(int radix)
- boolean hasNextBoolean()
- boolean hasNextByte()
- boolean hasNextByte(int radix)
- boolean hasNextDouble()
- boolean hasNextFloat()
- boolean hasNextInt()
- boolean hasNextInt(int radix)
- boolean hasNextLong()
- boolean hasNextLong(int radix)
- boolean hasNextShort()

- `boolean hasNextShort(int radix)`
- `String next()`
- `BigDecimal nextBigDecimal()`
- `BigInteger nextBigInteger()`
- `BigInteger nextBigInteger(int radix)`
- `boolean nextBoolean()`
- `byte nextByte()`
- `byte nextByte(int radix)`
- `double nextDouble()`
- `float nextFloat()`
- `int nextInt()`
- `int nextInt(int radix)`
- `long nextLong()`
- `long nextLong(int radix)`
- `short nextShort()`
- `short nextShort(int radix)`

Liefert das nächste Token.

Die Methode `useRadix(int)` ändert die Basis für Zahlen und `radix()` erfragt sie.

Trennzeichen definieren *

`useDelimiter()` setzt für die folgenden Filter-Vorgänge den Delimiter. Um nur lokal für das nächste Zerlegen einen Trenner zu setzen, lässt sich mit `next(String)` oder `next(Pattern)` ein Trennmuster angeben. `hasNext(String)` beziehungsweise `hasNext(Pattern)` liefern `true`, wenn das nächste Token dem Muster entspricht.

zB Beispiel

Der String `s` enthält eine Zeile wie `a := b`. Uns interessieren der linke und der rechte Teil:

```
String s = "Url := http://www.tutego.com";
Scanner scanner = new Scanner( s ).useDelimiter( "\\s*:=\\s*" );
System.out.printf( "%s = %s", scanner.next(), scanner.next() );
// Url = http://www.tutego.com
```

Mit `findInLine(String)` beziehungsweise `findInLine(Pattern)` wird der Scanner angewiesen, nach dem Muster nur bis zum nächsten Zeilenendezeichen zu suchen; Delimiter ignoriert er.

Beispiel

zB

Suche mit `findInLine()` nach einem Muster:

```
String text = "Hänsel-und-Gretel\ngingen-durch-den-Wald";
Scanner scanner = new Scanner( text ).useDelimiter( "-" );
System.out.println( scanner.findInLine( "Wald" ) ); // null
System.out.println( scanner.findInLine( "ete" ) ); // "ete"
System.out.println( scanner.next() );           // "l" "gingen"
System.out.println( scanner.next() );           // "durch"
```

Mit `findWithinHorizon(Pattern, int)` beziehungsweise `findWithinHorizon(String, int)` lässt sich eine Obergrenze von Code-Points (vereinfacht ausgedrückt, von Zeichen) angeben. Liefert die Methode in dieser Grenze kein Token, liefert sie `null` und setzt auch den Positionszeiger nicht weiter.

Landessprachen *

Auch ist die Scanner-Klasse in der Lage, die Dezimalzahlen unterschiedlicher Sprachen zu erkennen.

Beispiel

zB

Mit dem passenden `Locale`-Objekt erkennt der Scanner bei `nextDouble()` auch Fließkommazahlen mit Komma, etwa "12,34":

```
Scanner scanner = new Scanner( "12,34" ).useLocale( Locale.GERMAN );
System.out.println( scanner.nextDouble() ); // 12.34
```

Das klingt logisch, funktioniert aber bei einem deutschsprachigem Betriebssystem in der Regel auch ohne `useLocale(Locale.GERMAN)`. Der Grund ist einfach: Der Scanner setzt das `Locale` vorher standardmäßig auf `Locale.getDefault()`, und bei auf Deutsch eingestellten Betriebssystemen ist das eben `Locale.GERMAN`. Andersherum bedeutet das, dass eine in englischer Schreibweise angegebene Zahl wie 12.34 nicht erkannt wird und der Scanner eine `java.util.InputMismatchException` meldet.

```
final class java.util.Scanner
    implements Iterator<String>, Closeable
```

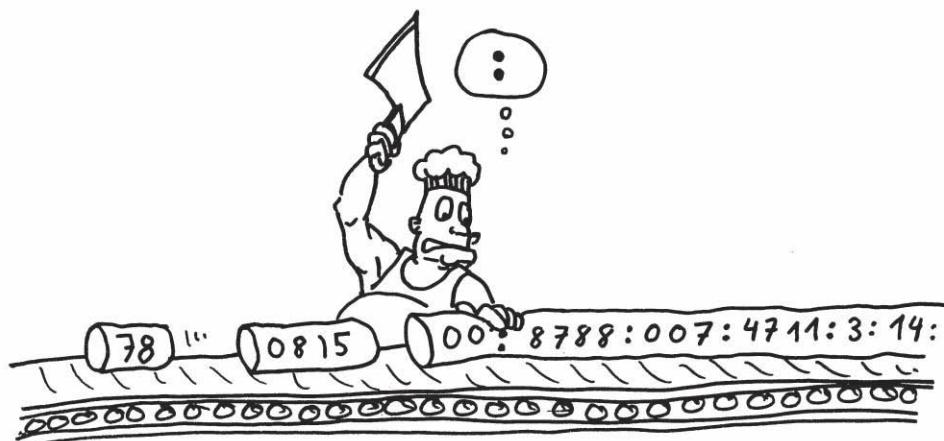
- `Scanner useLocale(Locale locale)`
Setzt die Sprache zum Erkennen der *lokalisierten Zahlen*, insbesondere der Fließkomma-Zahlen.
- `Locale locale()`
Liefert die eingestellte Sprache.

IO-Fehler während des Parsens *

Bezieht der Scanner die Daten von einem Readable, kann es Ein-/Ausgabefehler in Form von IOExceptions geben. Methoden wie `next()` geben diese Fehler nicht weiter, sondern fangen sie ab und speichern sie intern. Die Methode `ioException()` liefert dann das letzte IOException-Objekt oder `null`, falls es keinen Fehler gab.

4.9.3 Die Klasse StringTokenizer *

Die Klasse `StringTokenizer` zerlegt ebenfalls eine Zeichenkette in Tokens. Der `StringTokenizer` ist jedoch auf *einzelne* Zeichen als Trennsymbole beschränkt, während die Methode `split()` und die Klassen um `Pattern` einen regulären Ausdruck zur Beschreibung der Trennsymbole erlauben. Es sind keine Zeichenfolgen wie »:=« denkbar.



zB

4

Beispiel

Um einen String mithilfe eines StringTokenizer-Objekts zu zerlegen, wird dem Konstruktor der Klasse der zu unterteilende Text als Argument übergeben:

```
String s = "Faulheit ist der Hang zur Ruhe ohne vorhergehende Arbeit";
StringTokenizer tokenizer = new StringTokenizer( s );
while ( tokenizer.hasMoreTokens() )
    System.out.println( tokenizer.nextToken() );
```

Der Text ist ausschließlich ein Objekt vom Typ String.

Um den Text abzulaufen, gibt es die Methoden `nextToken()` und `hasMoreTokens()`.¹⁴ Die Methode `nextToken()` liefert das nächste Token im String. Ist kein Token mehr vorhanden, wird eine `NoSuchElementException` ausgelöst. Damit wir frei von diesen Überraschungen sind, können wir mit der Methode `hasMoreTokens()` nachfragen, ob noch ein weiteres Token vorliegt.

In der Voreinstellung sind Tabulator, Leerzeichen und Zeilentrenner die Delimiter. Sollen andere Zeichen als die voreingestellten Trenner den Satz zerlegen, kann dem Konstruktor als zweiter String eine Liste von Trennern übergeben werden. Jedes Zeichen, das in diesem String vorkommt, fungiert als einzelnes Trennzeichen:

```
StringTokenizer st = new StringTokenizer( "Blue=0000ff\nRed:ff0000\n", "=:\n" );
```

Neben den beiden Konstruktoren existiert noch ein dritter, der auch die Trennzeichen als eigenständige Bestandteile bei `nextToken()` übermittelt.

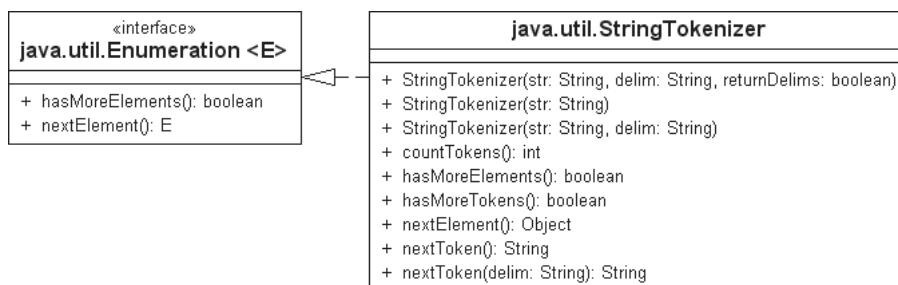


Abbildung 4.9: UML-Diagramm vom StringTokenizer

¹⁴ Die Methode `hasMoreElements()` ruft direkt `hasMoreTokens()` auf und wurde nur implementiert, da ein StringTokenizer die Schnittstelle Enumeration implementiert.

```
class java.util.StringTokenizer
    implements Enumeration<Object>
```

- StringTokenizer(String str, String delim, boolean returnDelims)
 Ein StringTokenizer für str, wobei jedes Zeichen in delim als Trennzeichen gilt. Ist returnDelims gleich true, so sind auch die Trennzeichen Tokens der Aufzählung.
- StringTokenizer(String str, String delim)
 Ein StringTokenizer für str, wobei alle Zeichen in delim als Trennzeichen gelten. Entspricht dem Aufruf von this(str, delim, false);
- StringTokenizer(String str)
 Ein StringTokenizer für str. Entspricht dem Aufruf von this(str, "\t\n\r\f", false);.
 Die Trennzeichen sind Leerzeichen, Tabulator, Zeilenende und Seitenvorschub.
- boolean hasMoreTokens()
 Testet, ob ein weiteres Token verfügbar ist. hasMoreElements() implementiert die Methode der Schnittstelle Enumeration, aber beide Methoden sind identisch.
- String nextToken()
 Liefert das nächste Token vom StringTokenizer. nextElement() existiert nur, damit der Tokenizer als Enumeration benutzt werden kann. Der weniger spezifische Ergebnistyp Object macht eine Typumwandlung erforderlich.
- String nextToken(String delim)
 Setzt die Delimiter-Zeichen neu und liefert anschließend das nächste Token.
- int countTokens()
 Zählt die Anzahl der noch möglichen nextToken()-Methodenaufrufe, ohne die aktuelle Position zu berühren. Der Aufruf der Methode ist nicht billig.

4.9.4 BreakIterator als Zeichen-, Wort-, Zeilen- und Satztrenner *

Benutzer laufen Zeichenketten aus ganz unterschiedlichen Gründen ab. Ein Anwendungsszenario ist das Ablaufen eines Strings Zeichen für Zeichen. In anderen Fällen sind nur einzelne Wörter interessant, die durch Wort- oder Satztrenner separiert sind. In wieder einem anderen Fall ist eine Textausgabe auf eine bestimmte Zeilenlänge gewünscht.

Zum Zerlegen von Zeichenfolgen sieht die Standardbibliothek im Java-Paket `java.text` die Klasse `BreakIterator` vor. Einen konkreten Iterator erzeugen diverse statische Methoden,

die optional auch nach speziellen Kriterien einer Sprache trennen. Wenn keine Sprache übergeben wird, wird automatisch die Standardsprache verwendet.

```
abstract class java.text.BreakIterator
    implements Cloneable
```

- static BreakIterator getCharacterInstance()
- static BreakIterator getCharacterInstance(Locale where)
Trennt nach Zeichen. Vergleichbar mit einer Iteration über charAt().
- static BreakIterator getSentenceInstance()
- static BreakIterator getSentenceInstance(Locale where)
Trennt nach Sätzen. Delimiter sind übliche Satztrenner wie »«, »!«, »?«.
- static BreakIterator getWordInstance()
- static BreakIterator getWordInstance(Locale where)
Trennt nach Wörtern. Trenner wie Leerzeichen und Satzzeichen gelten ebenfalls als Wörter.
- static BreakIterator getLineInstance()
- static BreakIterator getLineInstance(Locale where)
Trennt *nicht* nach Zeilen, wie der Name vermuten lässt, sondern ebenfalls nach Wörtern. Nur werden Satzzeichen, die am Wort »hängen«, zum Wort hinzugezählt. Praktisch ist dies für Algorithmen, die Textblöcke in eine bestimmte Breite bringen wollen. Ein Beispiel für die drei Typen zeigt das gleich folgende Programm.

Hinweis

Auf den ersten Blick ergibt ein BreakIterator von getCharacterInstance() keinen großen Sinn, denn für das Ablaufen einer Zeichenkette ließe sich viel einfacher eine Schleife nehmen und mit charAt() arbeiten. Der BreakIterator kann jedoch korrekt mit Unicode 4 umgehen, wo zwei char ein Unicode 4-Zeichen bilden. Zum zeichenweisen Iterieren über Strings ist auch CharacterIterator eine gute Lösung.



Beispiel für die drei BreakIterator-Typen

Das nächste Beispiel zeigt, wie ohne großen Aufwand durch Zeichenketten gewandert werden kann. Die Verwendung eines String-Tokenizers ist nicht nötig. Unsere statische Hilfsmethode out() gibt die Abschnitte der Zeichenkette bezüglich eines BreakIterator aus:

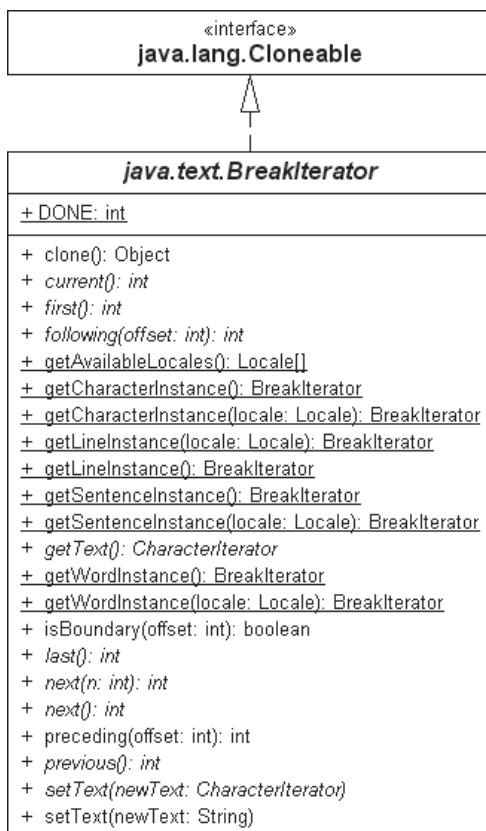


Abbildung 4.10: Klassendiagramm vom BreakIterator

Listing 4.22: BreakIteratorDemo.java, out()

```

static void out( String s, BreakIterator iter )
{
    iter.setText( s );

    for ( int last = iter.first(), next = iter.next();
          next != BreakIterator.DONE;
          last = next, next = iter.next() )
    {
        CharSequence part = s.subSequence( last, next );

        if ( Character.isLetterOrDigit( part.charAt( 0 ) ) )
    }
}
  
```

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

Einmal sollen die Wörter und einmal die Sätze ausgegeben werden:

Listing 4.23: BreakIteratorDemo.java, main()

```
public static void main( String[] args )
{
    String helmutKohl1 = "Ich weiß, dass ich 1945 fünfzehn war und 1953 achtzehn.",
           helmutKohl2 = "Das ist eine klassische journalistische Behauptung. " +
                          "Sie ist zwar richtig, aber sie ist nicht die Wahrheit./";

    BreakIterator sentenceIter = BreakIterator.getSentenceInstance();
    BreakIterator wordIter     = BreakIterator.getWordInstance();
    BreakIterator lineIter     = BreakIterator.getLineInstance();

    out( helmutKohl1, sentenceIter );
    out( helmutKohl2, sentenceIter );

    System.out.println( "-----" );

    out( helmutKohl1, wordIter );
    out( helmutKohl2, wordIter );

    System.out.println( "-----" );

    out( helmutKohl1, lineIter );
    out( helmutKohl2, lineIter );
}
```

Die Ausgabe enthält (skizziert):

```
Ich weiß, dass ich 1945 fünfzehn war und 1953 achtzehn.  

Das ist eine klassische journalistische Behauptung.  

Sie ist zwar richtig, aber sie ist nicht die Wahrheit.
```

```
Ich  
weiß  
...  
die  
Wahrheit  
-----
```

```
Ich  
weiß,  
...  
die  
Wahrheit.
```

Im letzten Beispiel ist gut zu sehen, dass die Wörter am Ende ihre Leer- und Satzzeichen behalten.

4.10 Zeichenkodierungen, XML/HTML-Entitys, Base64 *

4.10.1 Unicode und 8-Bit-Abbildungen

Einzelne Zeichen sind in Java intern immer in 16-Bit-Unicode kodiert, und ein String ist eine Folge von Unicode-Zeichen. Wollen wir diese Unicode-Zeichenkette in eine Datei schreiben, können mitunter andere Programme die Dateien nicht wieder einlesen, da sie keine Unicode-Zeichen erwarten oder nicht damit umgehen können. Die Unicode-Strings müssen daher in unterschiedliche Codepages, etwa Latin-1, umkodiert werden.

Kodierungen über die Klasse String vornehmen

Die String-Klasse konvertiert mit der Methode `getBytes(String charsetName)` beziehungsweise `getBytes(Charset charset)` den String in ein Byte-Feld mit einer bestimmten Zeichenkodierung. Auf diese Weise kann Java die interne Unicode-Repräsentation zum Beispiel in den EBCDIC-Zeichensatz eines IBM-Mainframes übertragen. Jede Kodierung (engl. *encoding*) ist durch eine Zeichenfolge oder ein Charset-Objekt definiert; die Namen sind unter <http://tutego.de/go/encoding> aufgeführt. Für den EBCDIC-Zeichensatz ist das die Codepage »Cp037«. Die DOS-Konsole unter Windows nutzt einen veränderten IBM-Zeichensatz, dessen Codepage »Cp850« heißt.

zB

4

Beispiel

Kodiere den String "Vernaschen" in EBCDIC:

```
try
{
    byte[] ebcnid = "Vernaschen".getBytes( "Cp037" );
    System.out.println( Arrays.toString(ebcnid) );
    // [-27, -123, -103, -107, -127, -94, -125, -120, -123, -107]
}
catch ( UnsupportedEncodingException e ) { ... }
```

Zur Kodierung in die andere Richtung, also von einem Byte-Feld in einen Unicode-String, müssen Sie einen Konstruktor der String-Klasse mit der Kodierung nutzen. Auch hier kann eine `UnsupportedEncodingException` folgen, wenn es die Kodierung nicht gibt.

zB

Beispiel

Kodiere das Byte-Feld mit den Zeichen nach dem EBCDIC-Alphabet zurück in einen String:

```
byte[] ebcnid = "Vernaschen".getBytes( "Cp037" );
String s = new String( ebcnid, "Cp037" );
System.out.println( s );                                // Vernaschen
```

4.10.2 Das Paket `java.nio.charset` und der Typ `Charset`

Konvertierungen zwischen Unicode-Strings und Byte-Folgen übernehmen `java.nio.charset.Charset`-Implementierungen. Die statische Methode `Charset.availableCharsets()` liefert eine `Map<String, Charset>` mit etwa 150 Einträgen – und somit Namen und assoziierte Klassen aller angemeldeten Kodierer. Ein `Charset`-Objekt lässt sich über einen Namen und dann mit `Charset.forName(String charsetName)` erfragen.

zB

Beispiel

Gib alle Kodierungen aus:

```
for ( String charsetName : Charset.availableCharsets().keySet() )
{
    System.out.println( charsetName );
```

zB Beispiel (Forts.)

```
Charset charset = Charset.forName( charsetName );
System.out.println( charset );      // Ausgabe wie oben
}
```

Mit dem konkreten Charset-Objekt lässt sich auf zwei Wegen weiterverfahren:

- Wir können es direkt mit den Methoden encode() und decode() konvertieren oder
- über die Methode newDecoder() einen CharsetDecoder beziehungsweise über newEncoder() einen CharsetEncoder erfragen und damit arbeiten.

Oftmals wird ein Charset aber an Klassen übergeben, die einen Charset für ihre Arbeit nutzen. Eine kleine Auswahl:

- byte[] String.getBytes(Charset charset)
- InputStreamReader(InputStream in, Charset cs)
- OutputStreamWriter(OutputStream out, Charset cs)
- String(byte[] bytes, Charset charset)
- String(byte[] bytes, int offset, int length, Charset charset)

Standards-Charsets

Das am System voreingestellte Charset liefert die statische Methode Charset.defaultCharset(). Weiterhin gibt es eine Klasse StandardCharsets mit Kontanten für oft gebrauchte Charset-Objekte:

```
final class java.nio.charset.StandardCharsets
    implements Cloneable
```

- final static Charset ISO_8859_1
- final static Charset US_ASCII
- final static Charset UTF_16
- final static Charset UTF_16BE
- final static Charset UTF_16LE
- final static Charset UTF_8

4.10.3 Konvertieren mit OutputStreamWriter/InputStreamReader-Klassen *

Neben der Klasse `String` mit `getBytes()` unterstützen auch andere Klassen die Umkodierung. Dazu zählen:

- `OutputStreamWriter`: Ein spezieller Writer, der Unicode-Zeichen mit einer gewählten Kodierung in einen binären Datenstrom schreibt.
- `InputStreamReader`: Übernimmt den anderen Weg zum Lesen von Byte-Folgen und Konvertieren in Unicode. Ist ein Reader.

Genauer stellt Kapitel 6, »Datenströme«, im 2. Band die Klassen vor, daher folgt an dieser Stelle nur kurz ein Beispiel.

Konvertieren in DOS-Latin-1

Zum korrekten Darstellen der Umlaute auf der Windows-DOS-Konsole wird ein `OutputStreamWriter` mit der Codepage 850 (DOS-Latin-1) verwendet.

Listing 4.24: GetBytesConverter.java, main()

```
try
{
    System.out.println( "Ich kann Ä Ü Ö und ß" );
    PrintWriter out = new PrintWriter(
        new OutputStreamWriter(System.out, "Cp850") );
    out.println( "Ich kann Ä Ü Ö und ß" );
    out.flush();
}
catch ( UnsupportedEncodingException e ) { e.printStackTrace(); }
```

Die Standard-Kodierung von Windows, »Cp1252« (Windows-1252 beziehungsweise Windows Latin-1), ist eine Anpassung von ISO 8859-1, die andere Zeichen in den Bereich 0x80 bis 0x9f setzt.

Tipp

Sollen ganze Dateien umkodiert werden, lässt sich auf der Kommandozeile das Dienstprogramm `native2ascii` nutzen. Siehe dazu auch »Enkodierung vom Quellcode festlegen *« in Abschnitt 4.1.6.



4.10.4 XML/HTML-Entitys ausmaskieren

In einer XML-Datei dürfen bestimmte Zeichen im normalen Textstrom nicht vorkommen und müssen umkodiert werden.

Zeichen	Umkodierung
"	"
&	&
'	'
<	<
>	>

Tabelle 4.19: Umkodierungen für eine XML-Datei

Eine Konstruktion wie " nennt sich *Entity*. Die gültigen Entitys werden im XML-Standard beschrieben.

Weiterhin gilt, dass bei einer Webseitenkodierung in ISO-8859-1 nur die »sicheren« Zeichen wie Ziffern und Buchstaben verwendet werden können, aber keine Sonderzeichen, wie etwa das Copyright- oder das Euro-Zeichen. Daher bietet HTML eine Umkodierung für Sonderzeichen an, die nicht im Zeichenvorrat von ISO 8859-1 enthalten sind – für das Copyright-Zeichen ist es etwa © und das Euro-Zeichen €. In XML ist diese Umkodierung nicht nötig, da XML leicht als UTF-8 geschrieben werden kann, und dann heißt es für das Euro-Zeichen nach der Position in der Unicode-Tabelle einfach €¹⁵

Java-Programme, die XML- oder HTML-Ausgaben erstellen oder XML/HTML-Dokumente lesen, müssen auf die korrekte Konvertierung achten. Die Standardbibliothek bringt hier nichts Offensichtliches mit, aber Open-Source-Bibliotheken füllen diese Lücke – so etwa *Apache Commons Lang* (<http://commons.apache.org/lang/>), das mit der Klasse org.apache.commons.lang.StringEscapeUtils einige Kodierungsmethoden bietet, um einen String in XML/HTML umzukodieren und einen XML/HTML-String mit Entitys in einen Java-String zu bringen, bei dem insbesondere die HTML-Entitys aufgelöst wurden. Die Klasse StringEscapeUtils bringt neben den statischen Methoden

- String escapeHtml3(String input)
- String unescapeHtml3(String input)

¹⁵ Das führt in HTML zu viel mehr Entitys als bei XML, sodass es ein Problem werden kann, eine HTML-Datei als XML einzulesen – der XML-Parser meckert dann über die unbekannten Entitys.

- String escapeHtml4(String input)
- String unescapeHtml4(String input)
- String escapeXml(String input)
- String unescapeXml(String input)

auch Methoden zum Maskieren von CSV-, Java- und JavaScript-Strings.

Beispiel

zB

Für eine einfache Kodierung (ohne Hochkommata) lässt sich ein XMLStreamWriter einsetzen:

```
StringWriter sw = new StringWriter();
XMLStreamWriter w = XMLOutputFactory.newInstance().createXMLStreamWriter(sw);
w.writeCharacters( "&'Müsli'" );
System.out.println( out.toString() ); // &lt;&amp;'Müsli"&gt;
```

4.10.5 Base64-Kodierung

Für die Übertragung von Binärdaten hat sich im Internet die Base64-Kodierung durchgesetzt, die zum Beispiel bei E-Mail-Anhängen und SOAP-Nachrichten zu finden ist. Die im RFC 1521 beschriebene Methode übersetzt drei Bytes (24 Bit) in vier Base64-kodierte Zeichen (vier Zeichen mit jeweils sechs repräsentativen Bits). Die Base64-Zeichen bestehen aus den Buchstaben des lateinischen Alphabets, den Ziffern 0 bis 9 sowie »+«, »/« und »=«. Die Konsequenz dieser Umformung ist, dass Binärdaten rund 33 % größer werden.

Das JDK liefert zwar Unterstützung für diese Base64-Umsetzung mit den Klassen `BASE64Encoder` und `BASE64Decoder`, aber da die Kodierer im nicht-öffentlichen Paket `sun.misc` liegen, könnte Oracle sie prinzipiell jederzeit entfernen.¹⁶ Wem das nicht ganz geheuer ist, der kann `javax.mail.internet.MimeUtility` von der JavaMail-API nutzen¹⁷ oder unter <http://jakarta.apache.org/commons/codec/> die *Commons Codec*-Bibliothek beziehen.

¹⁶ Siehe dazu <http://java.sun.com/products/jdk/faq/faq-sun-packages.html>. Bisher existieren sie aber seit über zehn Jahren, und wer Oracles Philosophie kennt, der weiß, dass die Abwärtskompatibilität oberste Priorität hat.

¹⁷ <http://www.rgagnon.com/javadetails/java-0598.html> gibt ein Beispiel. Die JavaMail-API ist Teil von Java EE 5 und muss sonst für das Java SE als Bibliothek hinzugenommen werden.

Beispiel

Das folgende Beispiel erzeugt zuerst ein Byte-Feld der Größe 112 und belegt es mit Zufallszahlen. Die internen JDK-Klassen kodieren das Byte-Feld in einen String, der auf dem Bildschirm ausgegeben wird. Nachdem der String wieder zurückkodiert wurde, werden die Byte-Felder verglichen und liefern natürlich true:

Listing 4.25: Base64Demo.java

```
import java.io.IOException;
import java.util.*;
import sun.misc.*;

public class Base64Demo
{
    public static void main( String[] args ) throws IOException
    {
        byte[] bytes1 = new byte[ 112 ];
        new Random().nextBytes( bytes1 );

        // Byte array -> to String
        String s = new BASE64Encoder().encode( bytes1 );
        System.out.println( s );

        // String enthält etwa:
        // QFgwDyiQ28/4GsF75fqLMj/bAIWNwOuBmE/SCl3H2XQFpSsSz0jtyROLU+kLiwWsnSUZljJr97Hy
        // LA3YUbf96Ym2zx9F9Y1N7P5ls0Cb/vr2crTQ/gXs757qaJF9E3szMN+EOCSSs1DrrzcNBrlcQg==

        // String -> byte[]
        byte[] bytes2 = new BASE64Decoder().decodeBuffer( s );
        System.out.println( Arrays.equals( bytes1, bytes2 ) );    // true
    }
}
```

4.11 Ausgaben formatieren

Immer wieder müssen Zahlen, Datumsangaben und Text auf verschiedenste Art und Weise formatiert werden. Zur Formatierung bietet Java mehrere Lösungen:

- Seit Java 5 realisieren die `format()`- und `printf()`-Methoden eine Ausgabe, so wie sie unter C mit `printf()` gesetzt wurde.
- Formatieren über Format-Klassen: Allgemeines Formatierungsverhalten wird in einer abstrakten Klasse `Format` fixiert; konkrete Unterklassen, wie `NumberFormat` und `DateFormat`, nehmen sich spezielle Datenformate vor.
- Umsetzung eines Strings nach einer gegebenen Maske mit einem `MaskFormatter`.
- Die Format-Klassen bieten nicht nur landes- beziehungsweise sprachabhängige Ausgaben per `format()`, sondern auch den umgekehrten Weg, Zeichenketten wieder in Typen wie `double` oder `Date` zu zerlegen. Jede Zeichenkette, die vom Format-Objekt erzeugt wurde, kann auch mit dem Parser wieder eingelesen werden.

4.11.1 Formatieren und Ausgeben mit `format()`

Die Klasse `String` stellt mit der statischen Methode `format()` eine Möglichkeit bereit, Zeichenketten nach einer Vorgabe zu formatieren.

Beispiel

zB

```
String s = String.format( "Hallo %s. Es gab einen Anruf von %s.",
                         "Chris", "Joy" );
System.out.println( s ); // Hallo Chris. Es gab einen Anruf von Joy.
```

Der erste an `format()` übergebene String nennt sich *Format-String*. Er enthält neben auszugebenden Zeichen weitere sogenannte *Format-Spezifizierer*, die dem Formatierer darüber Auskunft geben, wie er das Argument formatieren soll. `%s` steht für eine unformatierte Ausgabe eines Strings. Nach dem Format-String folgt ein Vararg (oder alternativ das Feld direkt) mit den Werten, auf die sich die Format-Spezifizierer beziehen.

Spezifizierer	Steht für...	Spezifizierer	Steht für...
<code>%n</code>	neue Zeile	<code>%b</code>	Boolean
<code>%%</code>	Prozentzeichen	<code>%s</code>	String

Tabelle 4.20: Die wichtigsten Format-Spezifizierer im Überblick

Spezifizierer	Steht für...	Spezifizierer	Steht für...
%c	Unicode-Zeichen	%d	Dezimalzahl
%x	Hexadezimalschreibweise	%t	Datum und Zeit
%f	Fließkommazahl	%e	wissenschaftliche Notation

Tabelle 4.20: Die wichtigsten Format-Spezifizierer im Überblick (Forts.)

**Tipp**

Der Zeilenvorschub ist vom Betriebssystem abhängig, und %n gibt uns ein gutes Mittel an die Hand, um an dieses Zeilenvorschubzeichen (oder diese Zeichenfolge) zu kommen. Dann kommt der `format()`-Aufruf auch mit einem Argument aus, und es lautet `String.format("%n")`.

```
final class java.lang.String
    implements CharSequence, Comparable<String>, Serializable
```

- static String format(String format, Object... args)
Liefert einen formatierten String, der aus dem String und den Argumenten hervorgeht.
- static String format(Locale l, String format, Object... args)
Liefert einen formatierten String, der aus der gewünschten Sprache, dem String und den Argumenten hervorgeht.

Intern werkeln `java.util.Formatter` (keine `java.text.Format`-Objekte), die sich auch direkt verwenden lassen; dort ist auch die Dokumentation festgemacht.

System.out.printf()

Soll eine mit `String.format()` formatierte Zeichenkette gleich ausgegeben werden, so muss dazu nicht `System.out.print(String.format(format, args))`; angewendet werden. Praktischerweise findet sich zum Formatieren und Ausgeben die aus String bekannte Methode `format()` auch in den Klassen `PrintWriter` und `PrintStream` (das `System.out`-Objekt ist vom Typ `PrintStream`). Da jedoch der Methodename `format()` nicht wirklich konsistent zu den anderen `printXXX()`-Methoden ist, haben die Entwickler die `format()`-Methoden auch unter dem Namen `printf()` zugänglich gemacht (die Implementierung von `printf()` ist eine einfache Weiterleitung zur Methode `format()`).

zB

Beispiel

Gib die Zahlen von 0 bis 16 hexadezimal aus:

```
for ( int i = 0x0; i <= 0xf; i++ )
    System.out.printf( "%x%n", i ); // 0 1 2 ... e f
```

4

Auch bei `printf()` ist als erstes Argument ein `Locale` möglich.

Pimp my String mit Format-Spezifizierern *

Die Anzahl der Format-Spezifizierer ist so groß und ihre weitere Parametrisierung ist so vielfältig, dass ein Blick in die API-Dokumentation auf jeden Fall nötig ist. Die wichtigsten Spezifizierer sind:

- `%n` ergibt das beziehungsweise die Zeichen für den Zeilenvorschub, jeweils bezogen auf die aktuelle Plattform. Die Schreibweise ist einem harten `\n` vorzuziehen, da dies nicht das Zeilenvorschubzeichen der Plattform sein muss.
- `%%` liefert das Prozentzeichen selbst, wie auch `\\"` in einem String den Backslash ausmaskiert.
- `%s` liefert einen String, wobei `null` zur Ausgabe »null« führt. `.S` schreibt die Ausgabe groß.
- `%b` schreibt ein `Boolean`, und zwar den Wert »true« oder »false«. Die Ausgabe ist immer »false« bei `null` und »true« bei anderen Typen wie `Integer`, `String`. `%B` schreibt den String groß.
- `%c` schreibt ein Zeichen, wobei die Typen `Character`, `Byte` und `Short` erlaubt sind. `.C` schreibt das Zeichen in Großbuchstaben.
- Für die ganzzahligen numerischen Ausgaben mit `%d` (Dezimal), `%x` (Hexadezimal), `%o` (Oktal) sind `Byte`, `Short`, `Integer`, `Long` und `BigInteger` erlaubt – `%X` schreibt die hexadezimalen Buchstaben groß.
- Bei den Fließkommazahlen mit `%f` oder `%e` (`%E`), `%g` (`%G`), `%a` (`%A`) sind zusätzlich die Typen `Float`, `Double` und `BigDecimal` zulässig. Die Standardpräzision für `%e`, `%E`, `%f` sind sechs Nachkommastellen.
- Im Fall von Datums-/Zeitangaben mit `%t` beziehungsweise `%T` sind erlaubt: `Long`, `Calendar` und `Date`. `%t` benötigt zwingend ein Suffix.
- Den Hashcode schreibt `%h` beziehungsweise `%H`. Beim Wert `null` ist auch das Ergebnis »null«.

Zusätzliche Flags, etwa für Längenangaben und die Anzahl an Nachkommastellen, sind möglich und werden im folgenden Beispiel gezeigt:

Listing 4.26: PrintfDemo.java, main()

```
PrintStream o = System.out;

int i = 123;
o.printf( "|%d|%d|%\n" , i, -i ); // |123|-123|
o.printf( "|%5d|%5d|%\n" , i, -i ); // | 123| -123|
o.printf( "|%-5d|%-5d|%\n" , i, -i ); // |123| |-123 |
o.printf( "|%+-5d|%+-5d|%\n" , i, -i ); // |+123| |-123 |
o.printf( "|%05d|%05d|%\n%\n" , i, -i ); // |00123|-0123|


o.printf( "|%X|%x|%\n" , 0xabc, 0xabc ); // |ABC|abc|
o.printf( "|%04x| %#x|%\n%\n" , 0xabc, 0xabc ); // |0abc|0xabc|


double d = 12345.678;
o.printf( "|%f|%f|%\n" , d, -d ); // |12345,678000| |-12345,678000|
o.printf( "|%+f|%+f|%\n" , d, -d ); // |+12345,678000| |-12345,678000|
o.printf( "|% f|% f|%\n" , d, -d ); // | 12345,678000| |-12345,678000|
o.printf( "|%.2f|%.2f|%\n" , d, -d ); // |12345,68| |-12345,68|
o.printf( "|%,.2f|%,.2f|%\n" , d, -d ); // |12.345,68| |-12.345,68|
o.printf( "|%.2f|%(2f|%\n" , d, -d ); // |12345,68| |(12345,68)|
o.printf( "|%10.2f|%10.2f|%\n" , d, -d ); // | 12345,68| | -12345,68|
o.printf( "|%010.2f|%010.2f|%\n" , d, -d ); // |0012345,68| |-012345,68|


String s = "Monsterbacke";
o.printf( "%n|%s|%\n" , s ); // |Monsterbacke|
o.printf( "|%S|%\n" , s ); // |MONSTERBACKE|
o.printf( "|%20s|%\n" , s ); // |           Monsterbacke|
o.printf( "|%-20s|%\n" , s ); // |Monsterbacke           |
o.printf( "|%7s|%\n" , s ); // |Monsterbacke|
o.printf( "|%.7s|%\n" , s ); // |Monster|
o.printf( "|%20.7s|%\n" , s ); // |           Monster|
```

```
Date t = new Date();
o.printf( "%tT%n", t );                                // 11:01:39
o.printf( "%tD%n", t );                                // 04/18/08
o.printf( "%1$te. %1$tb%n", t );                      // 18. Apr
```

Im Fall von Fließkommazahlen werden diese nach dem Modus `BigDecimal.ROUND_HALF_UP` gerundet, sodass etwa `System.out.printf("%.1f", 0.45);` die Ausgabe 0,5 ergibt.

Aus den Beispielen lassen sich einige Flags ablesen, insbesondere bei Fließkommazahlen. Ein Komma steuert, ob Tausendertrenner eingesetzt werden. Ein + gibt an, ob immer ein Vorzeichen angegeben wird, und ein Leerzeichen besagt, ob dann bei positiven Zeichen ein Platz freibleibt. Eine öffnende Klammer setzt bei negativen Zahlen kein Minus, sondern setzt diese in Klammern.

Beispiel

zB

Gib die Zahlen von 1 bis 10 aus. Die Zahlen 1 bis 9 sollen eine führende Null bekommen:

```
for ( int i = 1 ; i < 11; i++ )
    System.out.printf( "%02d%n", i ); // 01 02 ... 10
```

Format-Spezifizierer für Datumswerte

Aus dem Beispiel wird ersichtlich, dass %t nicht einfach die Zeit ausgibt, sondern immer ein weiteres Suffix erwartet, das genau angibt, welcher Datums-/Zeitteil eigentlich gewünscht ist. Tabelle 4.21 gibt die wichtigsten Suffixe an, und weitere finden Sie in der API-Dokumentation. Alle Ausgaben berücksichtigen die gegebene Locale-Umgebung.

Symbol	Beschreibung
%tA, %ta	Vollständiger/abgekürzter Name des Wochentags
%tB, %tb	Vollständiger/abgekürzter Name des Monatsnamens
%tC	Zweistelliges Jahrhundert (00–99)
%te, %td	Monatstag numerisch ohne bzw. mit führenden Nullen (1–31 bzw. 01–31)
%tk, %tl	Stundenangabe bezogen auf 24 bzw. 12 Stunden (0–23, 1–12)
%th, %tI	Zweistellige Stundenangabe bezogen auf 24 bzw. 12 Stunden (00–23, 01–12)
%tj	Tag des Jahres (001–366)
%tM	Zweistellige Minutenangabe (00–59)

Tabelle 4.21: Suffixe für Datumswerte

Symbol	Beschreibung
%tm	Zweistellige Monatsangabe (in der Regel 01–12)
%tS	Zweistellige Sekundenangabe (00–59)
%tY	Vierstellige Jahresangabe
%ty	Die letzten beiden Ziffern der Jahresangabe (00–99)
%tZ	Abgekürzte Zeitzone
%tz	Zeitzone mit Verschiebung zur GMT
%tR	Stunden und Minuten in der Form %tH:%tM
%tT	Stunden/Minuten/Sekunden in der Form %tH:%tM:%tS
%tD	Datum in der Form %tm/%td/%ty
%tF	ISO-8601-Format %tY-%tm-%td
%tc	Komplettes Datum mit Zeit in der Form %ta %tb %td %tT %tZ %tY

Tabelle 4.21: Suffixe für Datumswerte (Forts.)

Positionsangaben

Im vorangegangenen Beispiel lautete eine Zeile:

```
System.out.printf( "%te. %1$tb%n", t ); // 28. Okt
```

Die Angabe mit `Position$` ist eine Positionsangabe, und so bezieht sich `1$` auf das erste Argument, `2$` auf das zweite und so weiter (interessant ist, dass hier die Nummerierung nicht bei null beginnt).

Die Positionsangabe im Formatstring ermöglicht zwei Dinge:

- Wird, wie in dem Beispiel, das gleiche Argument mehrmals verwendet, ist es unnötig, es mehrmals anzugeben. So wiederholt `printf("%te. %tb%n", t, t)` das Argument `t`, was die Angabe einer Position vermeidet. Statt `%te. %1$tb%n` lässt sich natürlich auch `%1$te. %1$tb%n` schreiben, also auch für das erste Argument ausdrücklich die Position `1` vorschreiben.
- Die Reihenfolge der Parameter kann immer gleich bleiben, aber der Formatstring kann die Reihenfolge später ändern.

Der zweite Punkt ist wichtig für lokalisierte Ausgaben. Dazu ein Beispiel: Eine Bildschirmausgabe soll den Vor- und Nachnamen in unterschiedlichen Sprachen ausgeben. Die Reihenfolge der Namensbestandteile kann jedoch unterschiedlich sein, und nicht immer steht in jeder Sprache der Vorname vor dem Nachnamen. Im Deutschen heißt es im

Willkommenstext dann »Hallo Christian Ullenboom«, aber in der (erfundenen) Sprache Bwatuti hieße es »Jambo Ullenboom Christian«:

Listing 4.27: FormatPosition.java, main()

```
Object[] formatArgs = { "Christian", "Ullenboom" };

String germanFormat = "Hallo %1$s %2$s";
System.out.printf( germanFormat, formatArgs );
System.out.println();

String bwatutiFormat = "Jambo %2$s %1$s";
System.out.printf( bwatutiFormat, formatArgs );
```

Die Aufrufreihenfolge für Vor-/Nachname ist immer die gleiche, aber der Formatstring, der zum Beispiel extern aus einer Konfigurationsdatei oder Datenbank kommt, kann diese Reihenfolge ändern und so der Landessprache anpassen.

Tipp

Bezieht sich ein nachfolgendes Formatelement auf das vorangehende Argument, so kann ein < gesetzt werden:

```
Calendar c1 = new GregorianCalendar( 1973, 2, 12 );
Calendar c2 = new GregorianCalendar( 1985, 8, 2 );
System.out.printf( "%te. %<tb %<ty, %2$te. %<tb %<ty%n",
                  c1,           c2 );      // 12. Mrz 73, 2. Sep 85
```

Die Angaben für Monat und Jahr beziehen sich jeweils auf die vorangehenden Positionen. So muss nur einmal c1 und c2 angegeben werden.

4.11.2 Die Formatter-Klasse *

Die Methoden `format()` und `printf()` übernehmen die Aufbereitung nicht selbst, sondern delegieren sie an die Klasse `java.util.Formatter`. Das ist auch der Grund, warum die Dokumentation für die Formatspezifizierer nicht etwa an `String.format()` hängt, sondern an `Formatter`.

Ein Blick auf die Methode `format()` der Klasse `String` verrät, wie der Formatter ins Spiel kommt:

Listing 4.28: java.lang.String.format()

```
public static String format( String format, Object ... args )
{
    return new Formatter().format( format, args ).toString();
}
```

Ein `Formatter` übernimmt zwei Aufgaben. Er übernimmt zum einen die tatsächliche Formatierung, und zum anderen gibt er die formatierten Ausgaben an ein Ziel weiter. Wird der `Formatter` mit dem Standardkonstruktor aufgerufen, so baut er selbst das Ausgabeziel aus einem `StringBuilder` auf, den folgende `format()`-Aufrufe dann füllen. `toString()` vom `Formatter` ist so implementiert, dass es auf dem Ausgabeziel (also in unserem Fall dem `StringBuilder`) `toString()` aufruft.

Das Wissen um diesen Mechanismus ist für die Optimierung wichtig, um nicht zu viele Zwischenobjekte zu erzeugen. So führt die Schleife

```
StringBuilder sb = new StringBuilder();
for ( double d = 0; d <= 1; d += 0.1 )
{
    String s = String.format( "%.1f%n", d );
    sb.append( s );
}
System.out.println( sb ); // 0,1 0,2 ... 1,0
```

zu:

```
StringBuilder sb = new StringBuilder();
for ( double d = 0; d <= 1; d += 0.1 )
{
    String s = new Formatter().format( "%.1f%n", d ).toString();
    sb.append( s );
}
System.out.println( sb ); // 0,1 0,2 ... 1,0
```

Bei jedem Schleifendurchlauf wird also ein neuer `Formatter` aufgebaut. Intern entsteht damit ein neuer `StringBuilder` als Ziel für die formatierten Strings und schlussendlich über `toString()` ein String-Objekt. Nicht zu vergessen sind die internen char-Felder und der GC, der die Objekte wieder wegräumen muss.

Würden wir gleich das Ziel angeben, so könnte das viel effizienter werden. Dazu wird nicht der Standardkonstruktor von `Formatter` eingesetzt, der das Ziel mit einem neuen `StringBuilder` vorbestimmt, sondern ein eigenes Zielobjekt, das unser `StringBuilder` sein kann (es ist alles erlaubt, was vom Typ `Appendable` ist). Optimiert folgt somit:

Listing 4.29: `FormatterDemo.java`, `main()`

```
StringBuilder sb = new StringBuilder();
Formatter formatter = new Formatter( sb );

for ( double d = 0; d <= 1; d += 0.1 )
    formatter.format( "%.1f%n", d );

System.out.println( formatter ); // 0,1 0,2 ... 1,0
```

Wir weisen in der Schleife den `Formatter` an, die Formatierung vorzunehmen. Da dieser mit dem Ziel `StringBuilder` aufgebaut wurde, füllen die Zahlen nach und nach unseren `StringBuilder`. Temporäre Zwischenobjekte werden so minimiert. Zum Schluss wird der `Formatter` nach dem Ergebnis gefragt.

Formattable und `formatTo()`

Der Formatspezifizierer `%s` kann auf jedem Argumenttyp angewendet werden, denn durch die Varargs werden auch primitive Elemente zu Wrapper-Objekten (zu Wrapper-Klassen, siehe Abschnitt 8.2, »Wrapper-Klassen und Autoboxing«), die eine `toString()`-Methode haben. Nun kann es aber sein, dass `toString()` besonders implementiert werden muss und nicht unbedingt die Zeichenkette liefert, die für die Ausgabe gewünscht ist. Für diesen Fall berücksichtigt der `Formatter` einen besonderen Typ. Implementiert die Klasse die besondere Schnittstelle `java.util.Formattable`, so ruft der `Formatter` nicht die `toString()`-Methode auf, sondern `formatTo(Formatter formatter, int flags, int width, int precision)`. Die API-Dokumentation liefert ein Beispiel.

4.11.3 Formatieren mit Masken *

Oftmals unterscheidet sich bei grafischen Oberflächen die Darstellung von Daten von dem tieferliegenden Datenmodell. Während ein Datum zum Beispiel intern als große Zahl vorliegt, soll der Anwender sie in der gewünschten Landessprache sehen können. Bei einigen Ausgaben kommen Trennzeichen in die Ausgabe, um sie für den Leser bes-

ser verständlich zu machen. Eine IP-Adresse enthält Punkte an ganz bestimmten Stellen, eine Telefonnummer trennt die Vorwahl vom Rest ab, und die Segmente eines Datums trennen in der Regel die Zeichen »/« oder »-«.

Für Formatierungen, bei denen ein Originalstring in einen Ausgabestring konvertiert wird und dabei neue Zeichen zur Ausgabe eingefügt werden, bietet die Java-API eine Klasse `javax.swing.text.MaskFormatter`. Die Swing-Klasse hilft bei der Formatierung und dem Parsen:

Listing 4.30: MaskFormatterDemo.java, main()

```
MaskFormatter mf = new MaskFormatter( "***-*-*-*" );
mf.setValueContainsLiteralCharacters( false );
String valueToString = mf.valueToString( "12031973" );
System.out.println( valueToString );           // 12-03-1973
Object stringToValue = mf.stringToValue( valueToString );
System.out.println( stringToValue );           // 12031973
```

Der Konstruktor von `MaskFormatter` bekommt ein Muster, wobei es Platzhalter gibt. Das Sternchen * steht für ein Zeichen. Die Methode `valueToString()` bringt einen String in das Muster. Der gemusterte String wandelt `stringToValue()` wieder in das Original um.

Das Schöne ist, dass die Muster-Definitionen aus einer externen Quelle stammen können, ohne den Programmcode mit speziellen Formatierungsanweisungen zu verschmutzen. Neben * gibt es weitere Platzhalter, die erlaubte Zeichen eingrenzen, sodass bei der Umwandlung mit `valueToString()` eine `ParseException` ausgelöst wird, wenn das Zeichen nicht im Format des Musterplatzhalters ist.

Musterzeichen	Steht für
*	jedes Zeichen
#	eine Zahl, wie <code>Character.isDigit()</code> sie testet
?	ein Zeichen nach <code>Character.isLetter()</code>
A	ein Zeichen oder eine Ziffer, also <code>Character.isLetter()</code> oder <code>Character.isDigit()</code>
U	ein Zeichen nach <code>Character.isLetter()</code> , aber konvertiert in einen Großbuchstaben
L	ein Zeichen nach <code>Character.isLetter()</code> , aber konvertiert in einen Kleinbuchstaben

Tabelle 4.22: Musterplatzhalter

Musterzeichen	Steht für
H	ein Hexadezimalzeichen (0–9, a–f oder A–F)
'	einen ausmaskierten und nicht interpretierten Bereich

Tabelle 4.22: Musterplatzhalter (Forts.)

Weitere Möglichkeiten der Klasse beschreibt die API-Dokumentation.

4.11.4 Format-Klassen

Die Methode `format()` formatiert Zahlen, Datumswerte und sonstige Ausgaben und benötigt wegen ihrer Komplexität eine Beschreibung von mehreren Bildschirmseiten. Dabei gibt es noch eine andere Möglichkeit, für unterschiedliche Typen von zu formatierenden Werten eigene Klassen zu haben:

- **DateFormat:** Formatieren von Datums-/Zeitwerten
- **NumberFormat:** Formatieren von Zahlen
- **MessageFormat:** Formatieren für allgemeine Programmmeldungen

Die Klassen haben gemeinsam, dass sie die abstrakte Klasse `Format` erweitern und so eine gemeinsame Schnittstelle haben. Jede dieser Klassen implementiert auf jeden Fall die Methode `format()` zur Ausgabe und zum Parsen, also zur Konvertierung vom String in das Zielobjekt, die Methode `parseObject()`.

Zwei Gründe sprechen für den Einsatz der Format-Klassen:

- Es gibt in `String` zwar eine `format()`-Methode, aber keine `parseXXX()`-Methode.
- Die Format-Klassen liefern mit statischen `getXXXInstance()`-Methoden vordefinierte Format-Objekte, die übliche Standardausgaben erledigen, etwa gerundete Ganzzahlen, Prozente oder unterschiedlich genaue Datums-/Zeitangaben.

Das folgende Beispiel zeigt einige Anwendungen zum zweiten Punkt.

Ergebnis	Formatiert mit
02.09.2005	<code>DateFormat.getDateInstance().format(new Date())</code>
15:25:16	<code>DateFormat.getTimeInstance().format(new Date())</code>
02.09.2005 15:25:16	<code>DateFormat.getDateTimeInstance().format(new Date())</code>

Tabelle 4.23: Formatobjekte im Einsatz

Ergebnis	Formatiert mit
12.345,679	NumberFormat.getInstance().format(12345.6789)
12.345,68 €	NumberFormat.getCurrencyInstance().format(12345.6789)
12 %	NumberFormat.getPercentInstance().format(0.123)

Tabelle 4.23: Formatobjekte im Einsatz (Forts.)

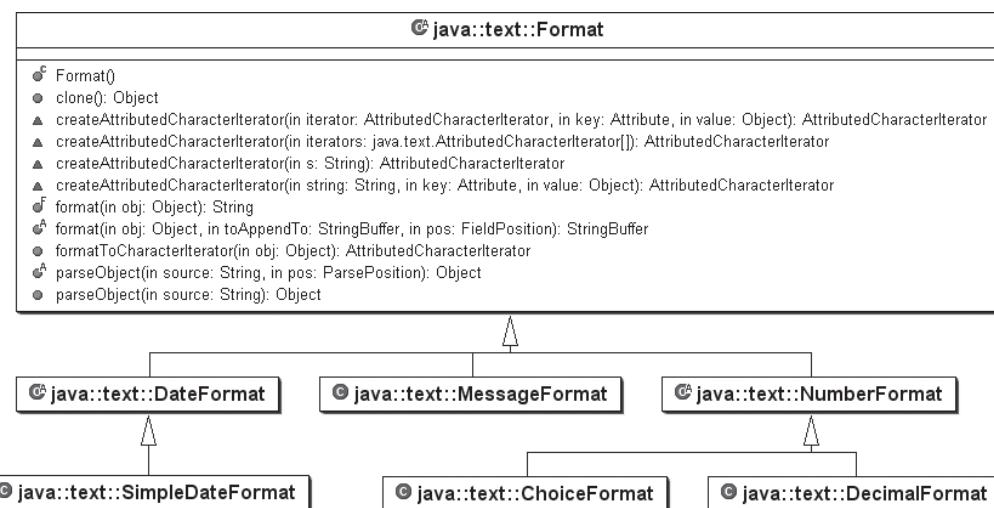


Abbildung 4.11: DateFormat, MessageFormat und NumberFormat erweitern die abstrakte Klasse Format. Die Unterklassen übernehmen die Ein-/Ausgabe für Datumsangaben, für allgemeine Programmmeldungen und für Zahlen.

Beim Einsatz von `DateFormat.getDateInstance().format(date)` berücksichtigt die Methode korrekt je nach Land die Reihenfolge von Tag, Monat und Jahr und das Trennzeichen. Bei einem `String.format()` über `%t` müssten die drei Segmente je nach Sprache in die richtige Reihenfolge gebracht werden, sodass die Variante über `DateFormat` besser ist.

→ Hinweis

`NumberFormat` stellt die Zahlen nicht in Exponentialschreibweise dar, und standardmäßig ist die Anzahl an Nachkommastellen beschränkt:

```
System.out.println( NumberFormat.getInstance().format( 2E30 ) );
System.out.println( NumberFormat.getInstance().format( 2E-30 ) );
```

Hinweis (Forts.)

Die Ausgabe ist:

2.000.000.000.000.000.000.000.000.000

0



```
abstract class java.text.Format  
implements Serializable, Cloneable
```

- `String format(Object obj)`
Formatiert das Objekt `obj` und gibt eine Zeichenkette zurück.
 - `abstract StringBuffer format(Object obj, StringBuffer toAppendTo, FieldPosition pos)`
Formatiert ein Objekt und hängt den Text an den angegebenen `StringBuffer` an (eine Methode mit `StringBuilder` gibt es nicht). Kann die Zeichenkette nicht mit `format()` nach den Regeln des Format-Objekts ausgegeben werden, löst die Methode eine `IllegalArgumentException` aus. Ist die Formatierungsanweisung falsch, so gibt `format()` das Unicode-Zeichen `\uFFFD` zurück.
 - `Object parseObject(String source)`
Analysiert den Text von Anfang an.
 - `abstract Object parseObject(String source, ParsePosition pos)`
Der Text wird ab der Stelle `pos` umgewandelt. Konnte `parseObject()` die Zeichenkette nicht zurückübersetzen, so folgt eine `ParseException`. `parseObject(String, ParsePosition)` verändert das `ParsePosition`-Objekt nicht und gibt die null-Referenz zurück.
 - `Object clone()`
Gibt eine Kopie zurück.

Die Mehrzahl der Format-Unterklassen implementiert statische Fabrikmethoden der Art:

- static XXXFormat getYYYInstance()
Liefert ein Formatierungsobjekt mit den Formatierungsregeln für das voreingestellte Land.
 - static XXXFormat getYYYInstance(Locale l)
Liefert ein Formatierungsobjekt mit den Formatierungsregeln für das angegebene Land. So erlauben die Unterklassen von Format es dem Benutzer auch, weitere Objekte zu erzeugen, die an die speziellen Sprachbesonderheiten der Länder angepasst sind.

4.11.5 Zahlen, Prozente und Währungen mit NumberFormat und DecimalFormat formatieren *

NumberFormat widmet sich der Ausgabe von Zahlen. Dabei unterstützt die Klasse vier Typen von Ausgaben, für die es jeweils eine statische Fabrikmethode gibt.

```
abstract class java.text.NumberFormat
extends Format
```

- static NumberFormat getInstance()
 Liefert den einfachen Formatierer für Zahlen.
- static NumberFormat getIntegerInstance()
 Liefert einen Formatierer, der den Nachkommateil abschneidet und rundet.
- static NumberFormat getPercentInstance()
 Liefert einen Formatierer, der Fließkommazahlen über die format()-Methode im Bereich von 0 bis 1 annimmt und dann als Prozentzahl formatiert. Nachkommastellen werden abgeschnitten.
- static NumberFormat getCurrencyInstance()
 Liefert einen Formatierer für Währungen, der ein Währungszeichen zur Ausgabe hinzufügt.

Die genannten vier statischen Methoden gibt es jeweils in der parameterlosen Variante und in der Variante mit einem Locale-Objekt, um etwa das Währungszeichen oder das Kommazeichen anzupassen.

Dezimalzahlformatierung mit DecimalFormat

DecimalFormat ist eine Unterklasse von NumberFormat und ermöglicht individuellere Anpassungen an die Ausgabe. Dem Konstruktor kann ein Formatierungsstring übergeben werden, sozusagen eine Vorlage, wie die Zahlen zu formatieren sind. Die Formatierung einer Zahl durch DecimalFormat erfolgt mit Rücksicht auf die aktuell eingestellte Sprache:

Listing 4.31: DecimalFormatDemo.java, main()

```
double d = 12345.67890;
DecimalFormat df = new DecimalFormat( "###,##0.00" );
System.out.println( df.format(d) );           // 12.345,68
```

Der Formatierungsstring kann eine Menge von Formatierungsanweisungen vertragen; im Beispiel kommen #, 0 und das Komma vor. Die beiden wichtigen Symbole sind je-

doch 0 und #. Beide repräsentieren Ziffern. Der Unterschied tritt erst dann zutage, wenn weniger Zeichen zum Formatieren da sind, als im Formatierungsstring genannt werden.

Symbol	Bedeutung
0	Repräsentiert eine Ziffer – ist die Stelle nicht belegt, wird eine Null angezeigt.
#	Repräsentiert eine Ziffer – ist die Stelle nicht belegt, bleibt sie leer, damit führende Nullen und unnötige Nullen hinter dem Komma nicht angezeigt werden.
.	Dezimaltrenner. Trennt Vor- und Nachkommastellen.
,	Gruppert die Ziffern (eine Gruppe ist so groß wie der Abstand von »« zu »».«).
;	Trennzeichen. Links davon steht das Muster für positive Zahlen, rechts davon das Muster für negative Zahlen.
-	Das Standardzeichen für das Negativpräfix
%	Die Zahl wird mit 100 multipliziert und als Prozentwert ausgewiesen.
\u2030	Die Zahl wird mit 1.000 multipliziert und als Promillewert auszeichnet.
\u00A4	Nationales Währungssymbol (€ für Deutschland)
\u00A4\u00A4	Internationales Währungssymbol (EUR für Deutschland)
X	Alle anderen Zeichen – symbolisch X – können ganz normal benutzt werden.
'	Ausmaskieren von speziellen Symbolen im Präfix oder Suffix

Tabelle 4.24: Formatieranweisungen für »DecimalFormat«

Hier sehen wir ein Beispiel für die Auswirkungen der Formatanweisungen auf einige Zahlen:

Format	Eingabezahl	Ergebnis
0000	12	0012
0000	12,5	0012
0000	1234567	1234567
##	12	12

Tabelle 4.25: Beispiel für verschiedene Formatanweisungen

Format	Eingabezahl	Ergebnis
##	12.3456	12
##	123456	123456
.00	12.3456	12,35
.00	.3456	,35
0.00	.789	0,79
#.000000	12.34	12,340000
,###	12345678.901	12.345.679
#.#;(#.#)	12345678.901	12345678,9
#.#;(#.#)	-12345678.901	(12345678,9)
,###.### \u00A4	12345.6789	12.345,68 €
,#00.00 \u00A4\u00A4	-12345678.9	-12.345.678,90 EUR
,#00.00 \u00A4\u00A4	0.1	00,10 EUR

Tabelle 4.25: Beispiel für verschiedene Formatanweisungen (Forts.)

Währungen angeben und die Klasse Currency

Die NumberFormat-Klasse liefert mit `getCurrencyInstance()` ein Format-Objekt, das neben der Dezimalzahl auch noch ein Währungssymbol mit anzeigt. So liefert `NumberFormat.getCurrencyInstance().format(12345.6789)` dann 12.345,68 €, also automatisch mit einem Euro-Zeichen. Dass es ein Euro-Zeichen ist und kein Yen-Symbol, liegt einfach daran, dass Java standardmäßig das eingestellte Land »sieht« und daraus die Währung ableitet. Wenn wir explizit den Formatter mit einem Land initialisieren, etwa wie in

```
NumberFormat frmt1 = DecimalFormat.getCurrencyInstance( Locale.FRANCE );
System.out.println( frmt1.format( 12345.6789 ) ); // 12 345,68 €
```

so ist die Währung automatisch Euro (denn Frankreich nutzt den Euro); schreiben wir `DecimalFormat.getCurrencyInstance(Locale.JAPAN)`, ist sie Yen, und wir bekommen ¥12,346 (es gibt standardmäßig keine Nachkommastellen beim Yen). Locale-Objekte repräsentieren immer eine Sprachregion.

DecimalFormat beziehungsweise schon die Oberklasse NumberFormat ermöglicht die explizite Angabe der Währung. In der Java-Bibliothek wird sie durch die Klasse `java.util.Currency` repräsentiert. NumberFormat liefert mit `getCurrency()` die eingestellte Currency, die zur Formatierung verwendet wird, und `setCurrency()` setzt sie neu. Das löst

Szenarios, in denen etwa ein Euro-Zeichen die Währung darstellt, aber die Zahlenformatierung englisch ist, wie die folgenden Zeilen zeigen:

```
NumberFormat frmt = DecimalFormat.getCurrencyInstance( Locale.ENGLISH );
frmt.setCurrency( Currency.getInstance( "EUR" ) );
System.out.println( frmt.format( 12345.6789 ) ); // EUR12,345.68
```

Die `Currency`-Klasse bietet drei statische Methoden, die `Currency`-Objekte liefern. Da ist zum einen `getAvailableCurrencies()`, was ein `Set<Currency>` liefert, und zum anderen gibt es die beiden Fabrikfunktionen `getInstance(Locale locale)` und `getInstance(String currencyCode)`. `Currency`-Objekte besitzen eine ganze Reihe von Objektfunktionen, die etwa den ISO-4217-Währungscode liefern oder den ausgeschriebenen Währungsnamen (und das auch noch in verschiedenen Sprachen, wenn gewünscht).

Folgendes Programm geht über alle Währungen und gibt die zentralen Informationen aus:

```
for ( Currency currency : Currency.getAvailableCurrencies() )
{
    System.out.printf( "%s, %s, %s (%s)%n",
        currency.getCurrencyCode(),
        currency.getSymbol(),
        currency.getDisplayName(),
        currency.getDisplayName(Locale.ENGLISH) );
}
```

Wir bekommen dann mehr als 200 Ausgaben, und die Ausgabe beginnt mit:

```
EGP, EGP, Ägyptisches Pfund (Egyptian Pound)
IQD, IQD, Irak Dinar (Iraqi Dinar)
GHS, GHS, Ghana Cedi (Ghana Cedi)
AFN, AFN, Afghani (Afghani)
MUR, MUR, Mauritius Rupie (Mauritius Rupee)
SGD, SGD, Singapur Dollar (Singapore Dollar)
...
```

4.11.6 MessageFormat und Pluralbildung mit ChoiceFormat

`MessageFormat` ist eine besondere Unterklasse von `Format`, die für die Formulierung von Nachrichten gedacht ist. Die Klasse ist in etwa mit den Formatierungen über `printf()`,

nur werden bei `MessageFormat` die Platzhalter immer per Index in geschweiften Klammern angesprochen.

zB Beispiel

Formuliere einen Nachrichtenstring mit drei Feldern:

```
int soldCars = 10;
double sum    = 1234534534;
String s = MessageFormat.format( "{0} Auto(s) verkauft am {1,date} zum "+  

                    "Gesamtpreis von {2,number,currency}." ,  

                    soldCars, new Date(), sum );
System.out.println( s );
```

Die Ausgabe ist:

»10 Auto(s) verkauft am 01.07.2011 zum Gesamtpreis von 1.234.534,534,00 €.«

Sie ist automatisch lokalisiert, die Sprache lässt sich jedoch wieder als Locale-Objekt übergeben.

ChoiceFormat

Eine besondere Möglichkeit ist die Verbindung von `MessageFormat` und `ChoiceFormat`, um das Problem zu lösen, das unser Beispiel im Fall von verkauften Autos mit »Auto(s)« löst. Im Deutschen ist die Pluralbildung anspruchsvoll, da es »0 Autos, 1 Auto, 2 Autos, 3 Autos« usw. heißt aber nur »0 Koffer, 1 Koffer, 2 Koffer, ...«. Das in Software zu modellieren ist nicht ganz einfach, aber mit `ChoiceFormat` lässt es sich lösen. Dem Konstruktor werden zum Generieren der Ausgabe zwei Felder mitgegeben: Ein Limit-Array kodiert Bereiche, und ein zweites Feld enthält die zugeordneten Elementen für den Bereich.

zB Beispiel

Löse das Problem mit »0 Autos, 1 Auto, 2 Autos« usw.:

```
MessageFormat formatter = new MessageFormat( "Du hast {0} {1} verkauft." );
double[] limits   = { 0.,      1.,      2.      };
String[] formats = { "Autos", "Auto", "Autos" };
ChoiceFormat choices = new ChoiceFormat( limits, formats );
formatter.setFormatByArgumentIndex( 1, choices );
int size = 4;
```

zB

Beispiel (Forts.)

```
Object[] params = { size, size };
System.out.println( formatter.format( params ) ); // Du hast 4 Autos verkauft.
```

Das Feld `{0., 1., 2.}` interpretiert sich so: Liegt der Wert zwischen größer gleich 0 und echt kleiner 1 (also bei Ganzzahlen ist er effektiv 0), wird das erste Element des Feldes `{"Autos", "Auto", "Autos"}`, also »Autos«, gewählt. Liegt es zwischen größer gleich 1 und echt kleiner 2, dann ist es das »Auto«. Alles was echt größer 2 ist, wird auf »Autos« abgebildet. Für die 2 im Feld ist `Math.nextUp(1.)` eine Alternative, und wenn auf »Komma-Autos« gewechselt wird, ist es gleich korrekt, denn es heißt zwar »1 Auto« aber »1,5 Autos«. (Genau genommen sind es sogar »1,0 Autos«, wodurch die ganze Pluralbildung wegfällt.)

4

Die API-Dokumentation der Klasse `ChoiceFormat` gibt weitere Beispiele und zeigt insbesondere, wie sich die Bereiche auch in den Strings selbst kodieren lassen, was für externe Übersetzungsdateien optimal ist.

4.12 Sprachabhängiges Vergleichen und Normalisierung *

Für die deutsche Sprache gilt, dass »ä« zwischen »a« und »b« äquivalent zu »ae« eingesortiert wird und nicht so, wie Unicode das Zeichen einordnet: hinter dem »z«. Ähnliches gilt für das »ß«. Auch das Spanische hat seine Besonderheiten im Alphabet: Hier gelten das »ch« und das »ll« als einzelner Buchstabe, die passend eingesortiert werden müssen.

Damit Java für alle Landessprachen die String-Vergleiche korrekt durchführen kann, bietet die Bibliothek Collator-Klassen.

4.12.1 Die Klasse Collator

Mit den `java.text.Collator`-Objekten ist es möglich, Zeichenketten nach jeweils landesüblichen Kriterien zu vergleichen. So werden die Sprachbesonderheiten jedes Landes beachtet. Ein Collator-Objekt wird vor seiner Benutzung mit `getInstance()` erzeugt.

zB

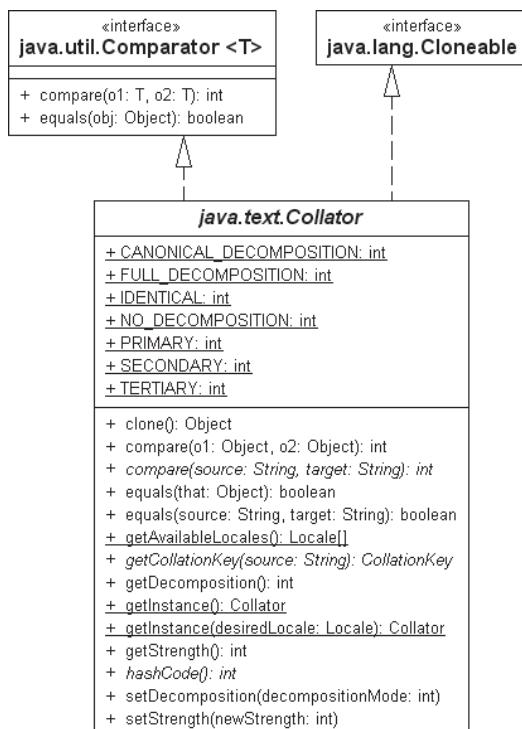
Beispiel

Das »Ä« liegt zwischen »A« und »B«:

zB Beispiel (Forts.)**Listing 4.32:** CollatorDemo.java

```
Collator col = Collator.getInstance();
System.out.println( col.compare( "Armleuchter", "Ätsch" ) ); // -1
System.out.println( col.compare( "Ätsch", "Bätsch" ) ); // -1
```

Die statische Fabrikmethode `getInstance()` nimmt optional einen Ländercode als `Locale`-Objekt an. Explizit setzt `getInstance(Locale.GERMAN)` das Vergleichsverfahren für deutsche Zeichenketten; die Länderbezeichnung ist in diesem Fall eine Konstante der `Locale`-Klasse. Standardmäßig nutzt `getInstance()` die aktuelle Einstellung des Systems.

**Abbildung 4.12:** UML-Diagramm von Collator und Obertypen

```
abstract class java.text.Collator
  implements Comparator<Object>, Cloneable
```

- static Collator getInstance()

Liefert einen Collator für die aktuelle Landessprache.

- static Collator getInstance(Locale desiredLocale)
Liefert einen Collator für die gewünschte Sprache.
- abstract int compare(String source, String target)
Vergleicht die beiden Zeichenketten auf ihre Ordnung. Der Rückgabewert ist entweder <0, 0 oder >0.
- int compare(Object o1, Object o2)
Vergleicht die beiden Argumente auf ihre Ordnung. Ruft compare((String)o1, (String)o2) auf.

Vergleichsarten

Die Collator-Klasse deklariert sinnvolle Methoden, die über die Vergleichsmöglichkeiten der String- und StringBuffer/StringBuilder-Klasse hinausgehen. So ist es über die Methode `setStrength()` möglich, unterschiedliche Vergleichsarten einzustellen. Die Collator-Klasse deklariert vier Strenge-Konstanten:

- PRIMARY: Erkennt Unterschiede im Grundzeichen, sodass »a« kleiner »b« ist. Es gibt keine Unterschiede durch Akzente und Umlaute, sodass »a«, »ä« und »á« gleich sind.
- SECONDARY: Erkennt Zeichen mit Akzenten. So sind »a« und »á« nicht mehr gleich wie bei PRIMARY.
- TERTIARY: Unterscheidet in der Groß- und Kleinschreibung; bei PRIMARY und SECONDARY ist die Schreibweise egal, und »a« ist gleich »A«.
- IDENTICAL: Wirklich alle Unicode-Zeichen sind anders. Während die ersten drei Konstanten nicht sichtbare Buchstaben wie '\u0001' oder '\u0006' gleich behandeln, sind sie unter IDENTICAL wirklich unterschiedlich.

Was die einzelnen Werte für jede Sprache bedeuten, beschreibt der Unicode-Standard präzise. Beispielsweise erkennt der tolerante Vergleich »abc« und »ABC« als gleich. Ohne explizit gesetztes `setStrength()` ist der Standard TERTIARY:

Listing 4.33: CollatorStrengthDemo.java

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

class CollatorStrengthDemo
{
    static void compare( Collator col, String a, String b )
    {

```

```
if ( col.compare( a, b ) < 0 )
    System.out.println( a + " < " + b );

if ( col.compare( a, b ) == 0 )
    System.out.println( a + " = " + b );

if ( col.compare( a, b ) > 0 )
    System.out.println( a + " > " + b );
}

public static void main( String[] args )
{
    Collator col = Collator.getInstance( Locale.GERMAN );

    System.out.println( "Strength = PRIMARY" );
    col.setStrength( Collator.PRIMARY );
    compare( col, "abc", "ABC" );
    compare( col, "Quäken", "Quaken" );
    compare( col, "boß", "boss" );
    compare( col, "boß", "boxen" );

    System.out.printf( "%nStrength = SECONDARY%n" );
    col.setStrength( Collator.SECONDARY );
    compare( col, "abc", "ABC" );
    compare( col, "Quäken", "Quaken" );
    compare( col, "boß", "boss" );
    compare( col, "boß", "boxen" );

    System.out.printf( "%nStrength = TERTIARY%n" );
    col.setStrength( Collator.TERTIARY );
    compare( col, "abc", "ABC" );
    compare( col, "Quäken", "Quaken" );
    compare( col, "boß", "boss" );
    compare( col, "boß", "boxen" );
}
}
```

Die Ausgabe ist folgende:

```
Strength = PRIMARY
abc = ABC
Quäken = Quaken
boß = boss
boß < boxen
Strength = SECONDARY
abc = ABC
Quäken > Quaken
boß = boss
boß < boxen
Strength = TERTIARY
abc < ABC
Quäken > Quaken
boß > boss
boß < boxen
```

4.12.2 Effiziente interne Speicherung für die Sortierung

Obwohl sich mit der Collator-Klasse sprachspezifische Vergleiche korrekt umsetzen lassen, ist die Geschwindigkeit gegenüber einem normalen String-Vergleich geringer. Daher bietet die Collator-Klasse die Objektmethode `getCollationKey()` an, die ein CollationKey-Objekt liefert, das schnellere Vergleiche zulässt.

```
Collator col = Collator.getInstance( Locale.GERMAN );
CollationKey key1 = col.getCollationKey( "ätzend" );
CollationKey key2 = col.getCollationKey( "Bremsspur" );
```

Durch CollationKeys lässt sich die Performance bei Vergleichen zusätzlich verbessern, da der landesspezifische String in einen dazu passenden, normalen Java-String umgewandelt wird, der dann schneller gemäß der internen Unicode-Zeichenkodierung verglichen werden kann. Dies bietet sich zum Beispiel beim Sortieren einer Tabelle an, wo mehrere Vergleiche *mit einem gleichen String* durchgeführt werden müssen. Der Vergleich wird mit `compareTo(CollationKey)` durchgeführt.

zB Beispiel

Der Vergleich von key1 und key2 lässt sich durch folgende Zeile ausdrücken:

```
int comp = key2.compareTo( key1 );
```

Das Ergebnis ist wie bei der `compare()`-Methode bei Collator-Objekten entweder <0, 0 oder >0.

```
final class java.text.CollationKey
implements Comparable<CollationKey>
```

- `int compareTo(CollationKey target)`

Vergleicht zwei CollationKey-Objekte miteinander.

- `int compareTo(Object o)`

Vergleicht den aktuellen CollationKey mit dem angegebenen Objekt. Ruft lediglich `compareTo((CollationKey)o)` auf.

- `byte[] toByteArray()`

Konvertiert den CollationKey in eine Folge von Bytes.

- `boolean equals(Object target)`

Testet die beiden CollationKey-Objekte auf Gleichheit.

- `String getSourceString()`

Liefert den String zum CollationKey.

- `int hashCode()`

Berechnet den Hashcode für den CollationKey.

```
abstract class java.text.Collator
implements Comparator<Object>, Cloneable
```

- `abstract CollationKey getCollationKey(String source)`

Liefert einen CollationKey für den konkreten String.

**Hinweis**

Das `java.text`-Paket hat weitere sehr interessante Schnittstellen. IBM stellt unter <http://www.ibm.com/developerworks/java/library/j-text-searching.html> weitere Typen, wie zum Beispiel `RuleBasedCollator` und `CollationElementIterator`, für das Suchen vor.

4.12.3 Normalisierung

Seit Java 6¹⁸ gibt es die Klasse `java.text.Normalizer`, die eine Unicode-Normalisierung (<http://unicode.org/faq/normalization.html>) ermöglicht. Die Klasse bietet zwei einfache statische Methoden:

- `boolean isNormalized(CharSequence src, Normalizer.Form form)`
- `String normalize(CharSequence src, Normalizer.Form form)`

Der `Normalizer` normalisiert oder testet nach den Vorgaben des Unicode-Standards (<http://www.unicode.org/unicode/reports/tr15/>) einen String nach den Normalisierungsformaten NFC, NFD, NFKC und NFKD.

Beispiel

zB

Normalisiere einen String:

```
String s = Normalizer.normalize( "ääüöñ", Normalizer.Form.NFKD );
System.out.println( s );                                // aa?u?o?n?
System.out.println( Arrays.toString( s.getBytes() ) );
// [97, 97, 63, 117, 63, 111, 63, 110, 63]
```

Die übrig gebliebenen Striche und Punkte lassen sich einfach entfernen, denn so ist das Ergebnis nicht sonderlich nützlich.

Beispiel

zB

Ersetze in einem String alle diakritischen Zeichen:

```
String s = "Müller";
s = Normalizer.normalize( s, Normalizer.Form.NFD );
s = s.replaceAll( "[\p{InCombiningDiacriticalMarks}\p{IsLm}\p{IsSk}]+", "" );
System.out.println( s ); // Muller
```

Die Lösung geht zweistufig vor. Der Normalisierer zerlegt zunächst den String und macht die eigentliche Arbeit. `replaceAll()` entfernt dann übrig gebliebene Punkte, Striche, Kreise und Häkchen.

¹⁸ Die Klasse gab es schon lange vorher, doch war sie in einem Sun-Paket »versteckt«.

4.13 Zum Weiterlesen

Wenn bei der Zeichenkettenverarbeitung sehr große Datenmengen verarbeitet werden, ist die Frage der Optimierung interessant. Die Standardimplementierung vom JDK arbeitet nur mit einem einfachen Suchalgorithmus, der bei großen Mustern und Suchstrings sehr ineffizient ist. Hier hat die Informatik in den letzten Jahrzehnten sehr interessante Ansätze hervorgebracht, wie zum Beispiel den Optimal-Mismatch-Algorithmus. Eine Applet-Visualisierung und kurze Beschreibung der Arbeitsweisen unterschiedlicher Suchalgorithmen bietet <http://www.igm.univ-mlv.fr/~lecroq/string/>. Ein gewisses Problem stellt aber der komplette Unicode-Standard dar, insbesondere Unicode 4.

Perl-Entwickler wachsen mit regulären Ausdrücken auf, während Java-Entwickler sich mit ihnen traditionell schwerer tun. Es lohnt sich auch für uns, sich mit diesem Bereich auseinanderzusetzen und zu üben, üben, üben – etwa an den unter <http://regexlib.com/> gesammelten Ausdrücken. Bei <http://www.rexv.org/> können die Ausdrücke über die Ajax-Technologie direkt im Webbrowser getestet werden. Oracle bietet unter <http://download.oracle.com/javase/tutorial/essential/regex/index.html> auch ein Tutorial an. Für Entwicklungsumgebungen bietet es sich an, ein Plugin zu installieren, mit dem reguläre Ausdrücke einfach eingegeben, getestet und dann in den Java-Editor übernommen werden können. Für Eclipse leistet das zum Beispiel <http://brosinski.com/regex/>.



Kapitel 5

Eigene Klassen schreiben

»Das Gesetz ist der abstrakte Ausdruck des allgemeinen an und für sich seienden Willens.«

– Georg Wilhelm Friedrich Hegel (1770–1831)

5

In den letzten Kapiteln haben wir viel über Objektorientierung erfahren, aber eher von der Benutzerseite her. Wir haben gesehen, wie Objekte aufgebaut werden, und auch einfache Klassen mit statischen Methoden haben wir implementiert. Es wird Zeit, das Wissen um alle objektorientierten Konzepte zu vervollständigen.

5.1 Eigene Klassen mit Eigenschaften deklarieren

Die Deklaration einer Klasse leitet das Schlüsselwort `class` ein. Im Rumpf der Klasse lassen sich deklarieren:

- Attribute (Variablen)
- Methoden
- Konstruktoren
- Klassen- sowie Exemplarinitialisierer
- innere Klassen, innere Schnittstellen und innere Aufzählungen

Eine ganz einfache Klassendeklaration

Wir wollen die Konzepte der Klassen und Schnittstellen an einem kleinen Spiel verdeutlichen. Beginnen wir mit dem Spieler, den die Klasse `Player` repräsentiert:

Listing 5.1: com/tutego/insel/game/v1/Player.java, Player

```
class Player
{
}
```

Die Klasse hat einen vom Compiler generierten Konstruktor, sodass sich ein Exemplar unserer Klasse mit `new Player()` erzeugen lässt.

5.1.1 Attribute deklarieren

Diese Player-Klasse hat bisher keine Attribute und kann bisher nichts. Geben wir dem Spieler zwei Attribute: eines für den Namen und ein zweites für einen Gegenstand, den er trägt. Die Datentypen sollen beide String sein:

Listing 5.2: com/tutego/insel/game/v2/Player.java, Player

```
class Player
{
    String name;
    String item;
}
```



Hinweis

Eine spezielle Namenskonvention für Objektvariablen gibt es nicht. So ist es zwar möglich, zur Unterscheidung von lokalen Variablen ein Präfix wie »f« oder »_« voranzustellen, doch sogar die Eclipse-Macher sind davon abgekommen. Objektvariablen können auch grundsätzlich wie Methoden heißen, doch ist das unüblich, da Variablennamen im Allgemeinen Substantiv und Methoden Verben sind. Da Bezeichner nie so heißen können wie Schlüsselwörter, fallen Variablen wie enum schon raus (das führte in Java 5 zu einigen Quellcodeänderungen, da dort enum als neues Schlüsselwort eingeführt wurde).

Eine zweite Klasse Playground erzeugt in der statischen `main()`-Methode für den mutigen Spieler ein Player-Objekt, schreibt und liest die Attribute:

Listing 5.3: com/tutego/insel/game/v2/Playground.java, Playground

```
class Playground
{
    public static void main( String[] args )
```

```

{
    Player p = new Player();
    p.name = "Mutiger Manfred";
    p.item = "Schlips";

    System.out.printf( "%s nimmt einen %s mit.", p.name, p.item );
}
}

```

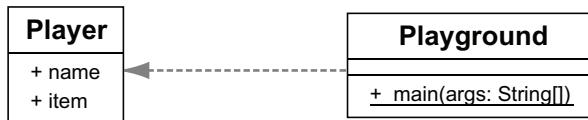


Abbildung 5.1: Das UML-Diagramm zeigt Abhängigkeiten von Playground und Player

Initialisierung von Attributen

Anders als lokale Variablen initialisiert die Laufzeitumgebung alle Attribute mit einem Standardwert:

- 0 bei numerischen Werten und char
- false bei boolean
- null bei Referenzvariablen

Gefällt uns das nicht, lassen sich die Variablen mit einem Wert belegen:

```

class Player
{
    String name = "";
}

```

Gültigkeitsbereich, Sichtbarkeit und Lebensdauer

Lokale Variablen beginnen ihr Leben in dem Moment, in dem sie deklariert und initialisiert werden. Endet der Block, ist die lokale Variable nicht mehr gültig, und sie kann nicht mehr verwendet werden, da sie aus dem Sichtbarkeitsbereich verschwunden ist. Bei Objektvariablen ist das anders. Eine Objektvariable lebt ab dem Moment, zu dem das Objekt mit `new` aufgebaut wurde, und sie lebt so lange, bis der Garbage-Collector das Ob-

ject wegräumt. Sichtbar und gültig ist die Variable aber immer im gesamten Objekt und in allen Blöcken.¹

 Spätestens, wenn zwei Klassen im Editor offen sind, möchten Tastaturjunkies schnell zwischen den Editoren wechseln. Das geht in Eclipse mit **Strg + F6**. Allerdings ist dieses Tastenkürzel in der Windows-Welt unüblich, sodass es umdefiniert werden kann, etwa zu **Strg + ↩**. Das geht so: Unter **WINDOWS • PREFERENCES** aktivieren wir unter **GENERAL • KEYS** das Kommando **NEXT EDITOR**. Im Textfeld **BINDING** lässt sich zunächst das alte Kürzel löschen und einfach **Strg + ↩** drücken. Das Ganze lässt sich auch für **PREVIOUS EDITOR** und **Strg + ⌘ + ↩** wiederholen.

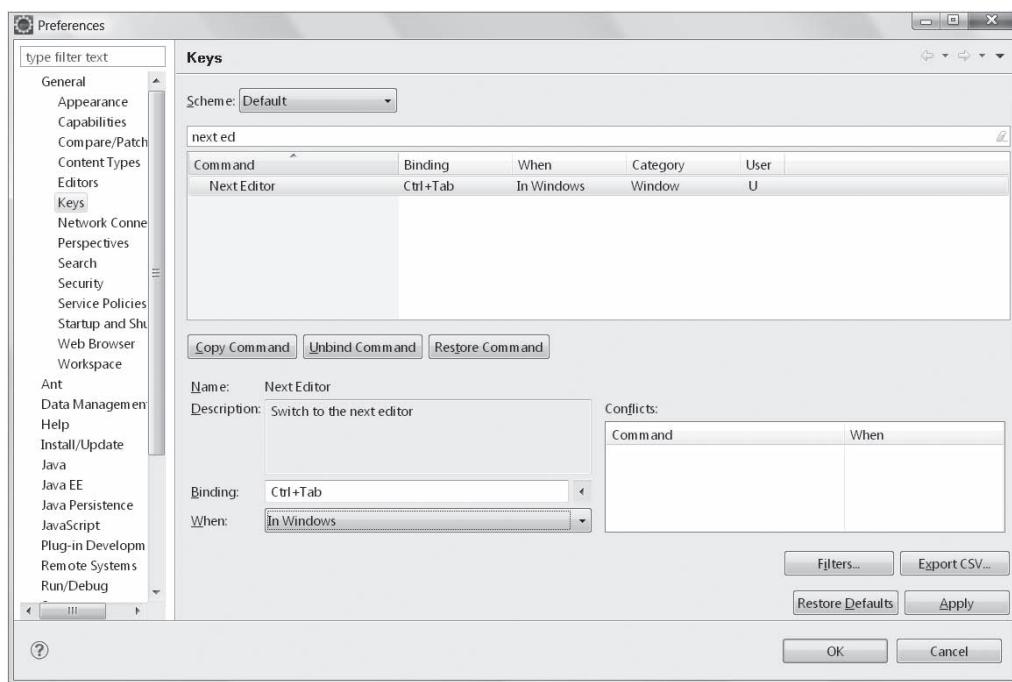


Abbildung 5.2: Tastenkürzel in Eclipse einstellen

5.1.2 Methoden deklarieren

Zu Attributen gesellen sich Methoden, die üblicherweise auf den Objektvariablen arbeiten. Anders als unsere bisherigen statischen Methoden werden diese aber für Player

¹ Das gilt nicht für statische Methoden und statische Initialisierungsblöcke, aber diese werden erst später vorgestellt.

keine Klassenmethoden sein. Geben wir dem Spieler zwei Methoden: `clearName()` soll den Namen auf den Leerstring "" zurücksetzen, und `hasCompoundName()` soll verraten, ob der Spielername aus einem Vor- und Nachnamen zusammengesetzt ist. Der Name »Parry Hotter« ist zum Beispiel zusammengesetzt, »Spuckiman« aber nicht.

Listing 5.4: com/tutego/insel/game/v3/Player.java, Player

```
class Player
{
    String name = "";
    String item = "";

    void clearName()
    {
        name = "";
    }

    boolean hasCompoundName()
    {
        return (name == null) ? false : name.contains( " " );
    }
}
```

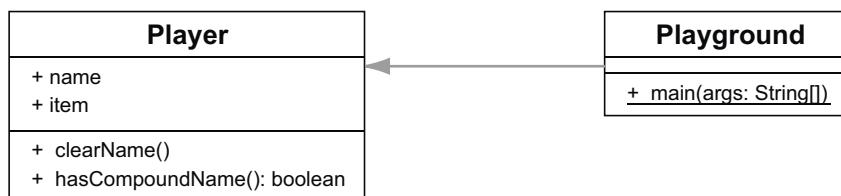


Abbildung 5.3: Das Klassendiagramm von Player zeigt zwei Attribute und zwei Methoden.

Testen wir die Methode mit zwei Spielern:

Listing 5.5: com/tutego/insel/game/v3/Playground.java, main()

```
Player parry = new Player();
parry.name = "Parry Hotter";
System.out.printf( "%s' hat zusammengesetzten Namen: %b%n",
                   parry.name, parry.hasCompoundName() );
```

```
Player spucki = new Player();
spucki.name = "Spuckiman";
System.out.printf( "'%s' hat zusammengesetzten Namen: %b%n",
                   spucki.name, spucki.hasCompoundName() );
spucki.clearName();
System.out.printf( "Spuckis Name ist leer? %b%n", spucki.name.isEmpty() );
```

Wie zu erwarten, ist die Ausgabe:

```
'Parry Hotter' hat zusammengesetzten Namen: true
'Spuckiman' hat zusammengesetzten Namen: false
Spuckis Name ist leer? true
```

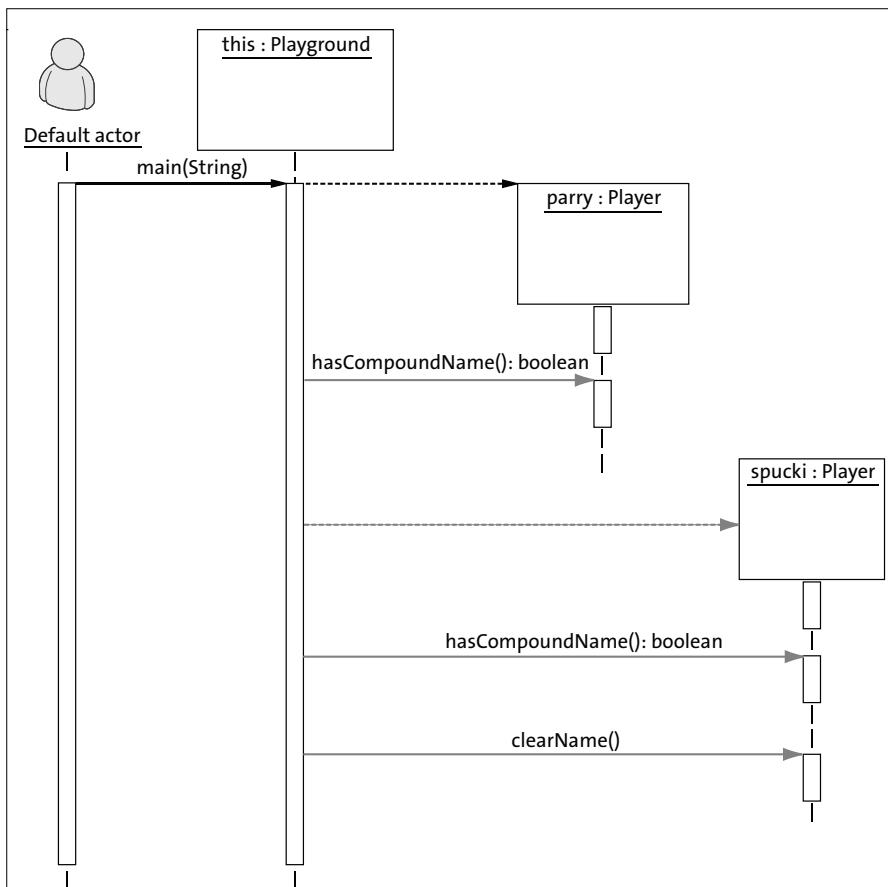
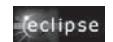


Abbildung 5.4: Ein Sequenzdiagramm stellt nur die Objekterzeugung und Methodenaufrufe, jedoch keine Attributzugriffe dar.



Um schnell von einer Methode (oder Variablen) zur anderen zu navigieren, zeigt **Strg** + **0** ein Outline an (dieselbe Ansicht wie in der Ansicht OUTLINE). Im Unterschied zur Ansicht OUTLINE lässt sich in diesem kleinen gelben Fenster mit den Cursor-Tasten navigieren, und ein **←** befördert uns zur angewählten Methode oder dem angewählten Attribut. Wird in der Ansicht erneut **Strg** + **0** gedrückt, befinden sich dort auch die in den Oberklassen deklarierten Eigenschaften. Sie sind grau, und zusätzlich befinden sich hinter den Eigenschaften die Klassennamen.

5

Methodenaufrufe und Nebeneffekte

Alle Variablen und Methoden einer Klasse sind in der Klasse selbst sichtbar. Das heißt, innerhalb einer Klasse werden die Objektvariablen und Methoden mit ihrem Namen verwendet. Somit greift die Methode `hasCompoundName()` direkt auf das nötige Attribut `name` zu, um die Programmlogik auszuführen. Dies wird oft für Nebeneffekte (Seiteneffekte) genutzt. Die Methode `clearName()` ändert ausdrücklich eine Objektvariable und verändert so den Zustand des Objekts. `hasCompoundName()` liest dagegen nur den Zustand, modifiziert ihn aber nicht. Methoden, die Zustände ändern, sollten das in der API-Beschreibung entsprechend dokumentieren.

Objektorientierte und prozedurale Programmierung im Vergleich

Entwickler aus der prozeduralen Welt haben ein anderes Denkmodell verinnerlicht, so dass wir an dieser Stelle die Besonderheit der Objektorientierung noch einmal verdeutlichen wollen. Während in der guten objektorientierten Modellierung die Objekte immer gleichzeitig Zustand und Verhalten besitzen, gibt es in der prozeduralen Welt nur Speicherbereiche, die referenziert werden; Daten und Verhalten liegen hier nicht zusammen. Problematisch wird es, wenn die prozedurale Denkweise in Java-Programme abgebildet wird. Dazu ein Beispiel: Die Klasse `PlayerData` ist ein reiner Datencontainer für den Zustand, aber Verhalten wird hier nicht deklariert:

Listing 5.6: PlayerData.java

```
class PlayerData
{
    String name = "";
    String item = "";
}
```

Anstatt nun die Methoden ordentlich, wie in unserem ersten Beispiel, mit an die Klasse zu hängen, würde in der prozeduralen Welt ein Unterprogramm genau ein Datenobjekt bekommen und von diesem Zustände erfragen oder ändern:

Listing 5.7: PlayerFunctions.java

```
class PlayerFunctions
{
    static void clearName( PlayerData data )
    {
        data.name = "";
    }

    static boolean hasCompoundName( PlayerData data )
    {
        return (data.name == null) ? false : data.name.contains( " " );
    }
}
```

Da die Unterprogramme nun nicht mehr an Objekte gebunden sind, können sie statisch sein. Genauso falsch wären aber auch Methoden (egal ob statisch oder nicht) in der Klasse PlayerData, wenn sie ein PlayerData-Objekt übergeben bekommen.

Beim Aufruf ist dieser nicht-objektorientierte Ansatz gut zu sehen. Setzen wir links den falschen und rechts den korrekt objektorientiert modellierten Weg ein:

Prozedural	Objektorientiert
PlayerData parry = new PlayerData(); parry.name = "Parry Hotter"; PlayerFunctions.hasCompoundName(parry); PlayerFunctions.clearName(parry);	Player parry = new Player(); parry.name = "Parry Hotter"; parry.hasCompoundName(); parry.clearName();

Tabelle 5.1: Gegenüberstellung prozedurale und objektorientierte Programmierung

Ein Indiz für problematische objektorientierte Modellierung ist also, wenn externen Methoden Objekte übergeben werden, anstatt die Methoden selbst an die Objekte zu setzen.

5.1.3 Die this-Referenz

In jeder Objektmethode und jedem Konstruktor steht eine Referenz mit dem Namen `this` bereit, die auf das eigene Exemplar zeigt. Mit dieser `this`-Referenz lassen sich elegante Lösungen realisieren, wie die folgenden Beispiele zeigen:

- Die `this`-Referenz löst das Problem, wenn Parameter beziehungsweise lokale Variablen Objektvariablen verdecken.
- Liefert eine Methode als Rückgabe die `this`-Referenz auf das aktuelle Objekt, lassen sich Methoden der Klasse einfach hintereinandersetzen.
- Mit der `this`-Referenz lässt sich einer anderen Methode eine Referenz auf uns selbst geben.

Überdeckte Objektvariablen nutzen

Trägt eine lokale Variable den gleichen Namen wie eine Objektvariable, so *verdeckt* sie diese. Das folgende Beispiel deklariert in der `Player`-Klasse eine Objektvariable `name`, und eine Methode `quote()` deklariert in der Parameterliste eine Variable, die ebenfalls `name` heißt. Somit bezieht sich in der Methode jeder Zugriff auf `name` auf die Parametervariable und nicht auf die Objektvariable:

```
class Player
{
    String name;                                // Objektvariable name

    void quote( String name )                  // Lokale Parametervariable name
    {
        System.out.println( "" + name + "" );   // Bezieht sich auf Parametervariable
    }
}
```

Das heißt aber nicht, dass auf die äußere Variable nicht mehr zugegriffen werden kann. Die `this`-Referenz zeigt auf das aktuelle Objekt, und damit ist auch ein Zugriff auf Objekteigenschaften jederzeit möglich:

```
class Player
{
    String name;

    void quote( String name )
```

```

{
    System.out.println( "" + this.name + "" ); // Zugriff auf Objektvariable
    System.out.println( "" + name + "" );      // Zugriff auf Parametervariable
}
}

```

Häufiger Einsatzort für das `this` in Methoden sind Methoden, die Zustände initialisieren. Gerne nennen Entwickler die Parametervariablen so wie die Exemplarvariablen, um damit eine starke Zugehörigkeit auszudrücken. Schreiben wir eine Methode `setName()`:

```

class Player
{
    String name;

    void setName( String name )
    {
        this.name = name;
    }
}

```

Der an `setName()` übergebene Wert soll die Objektvariable `name` initialisieren. So greift `this.name` auf die Objektvariable direkt zu, sodass die Zuweisung `this.name = name;` die Objektvariable mit dem Argument initialisiert.

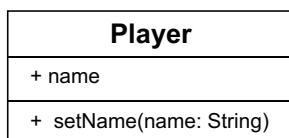


Abbildung 5.5: Klassendiagramm für Player mit Setter

this für kaskadierte Methoden *

Die `append()`-Methoden bei `StringBuilder` liefern die `this`-Referenz, sodass sich Folgendes schreiben lässt:

```

StringBuilder sb = new StringBuilder();
sb.append( "Android oder iPhone" ).append( '?' );

```

Jedes `append()` liefert das `StringBuilder`-Objekt, auf dem es aufgerufen wird. Wir wollen diese Möglichkeit bei einem Spieler programmieren, sodass die Methoden `name()` und `item()` Spielername und Gegenstand zuweisen. Beide Methoden liefern ihr eigenes Player-Objekt über die `this`-Referenz zurück:

Listing 5.8: com/tutego/insel/game/v4/Player.java, Player

```
class Player
{
    String name = "", item = "";

    Player name( String name )
    {
        this.name = name;
        return this;
    }

    String name()
    {
        return name;
    }

    Player item( String item )
    {
        this.item = item;
        return this;
    }

    String item()
    {
        return item;
    }

    String id()
    {
        return name + " hat " + item;
    }
}
```

Player	
name	
item	
+ name(name: String): Player	
+ name(): String	
+ item(item: String): Player	
+ item(): String	
+ id(): String	

Abbildung 5.6: Player-Klasse mit kaskadierbaren Methoden

Erzeugen wir einen Player, und kaskadieren wir einige Methoden:

Listing 5.9: com/tutego/insel/game/v4/Playground.java, main()

```
Player parry = new Player().name( "Parry" ).item( "Helm" );
System.out.println( parry.name() );                                // Parry
System.out.println( parry.id() );                                 // Parry hat Helm
```

Der Ausdruck `new Player()` liefert eine Referenz, die wir sofort für den Methodenaufruf nutzen. Da `name(String)` wiederum eine Objektreferenz vom Typ `Player` liefert, ist dahinter direkt `.item(String)` möglich. Die Verschachtelung von `name(String).item(String)` bewirkt, dass Name und Gegenstand gesetzt werden und der jeweils nächste Methodenaufruf in der Kette über `this` eine Referenz auf dasselbe Objekt, aber mit verändertem internen Zustand bekommt.



Hinweis

Bei Anfragemethoden könnten wir versucht sein, diese praktische Eigenschaft überall zu verwenden. Üblicherweise sollten Objekte jedoch ihre Eigenschaften nach der Java-Beans-Konvention mit `void setXXX()` setzen, und dann liefern sie ausdrücklich keine Rückgabe. Eine mit dem Objekttyp deklarierte Rückgabe `Player setName(String)` verstößt also gegen diese Konvention, sodass die Methode in dem Beispiel einfach `Player name(String)` heißt. Beispiele dieser Bauart sind in der Java-Bibliothek an einigen Stellen zu finden. Sie werden auch *Builder* genannt.

5.2 Privatsphäre und Sichtbarkeit

Innerhalb einer Klasse sind alle Methoden und Attribute für die Methoden sichtbar. Damit die Daten und Methoden einer Klasse vor externem Zugriff geschützt oder ausdrücklich für andere wiederum als öffentlich sichtbar markiert sind, gibt es unterschiedliche Sichtbarkeiten:

- öffentlich
- geschützt
- paketsichtbar
- privat

Für drei Sichtbarkeiten gibt es Schlüsselwörter, die *Sichtbarkeitsmodifizierer*. In diesem Abschnitt sollen die Sichtbarkeiten `public` (öffentlich), `private` (privat) und `protected` (geschützt) erklärt werden; zu `protected` (geschützt) kommen wir in Abschnitt 5.8 beim Thema Vererbung zurück.

5.2.1 Für die Öffentlichkeit: `public`

Der Sichtbarkeitsmodifizierer `public` an Klassen, Konstruktoren, Methoden und sonstigen Klassen-Innereien bestimmt, dass alle diese markierten Elemente von außen für jeden sichtbar sind. Es spielt dabei keine Rolle, ob sich der Nutzer im gleichen oder in einem anderen Paket befindet.

Ist zwar die Klasse `public`, aber eine Eigenschaft `privat`, kann eine fremde Klasse dennoch nicht auf die Eigenschaft zurückgreifen. Und ist eine Eigenschaft `public`, aber die Klasse `privat`, dann kann eine andere Klasse erst gar nicht an diese Eigenschaft »herankommen«.

5.2.2 Kein Public Viewing – Passwörter sind `privat`

Der Sichtbarkeitsmodifizierer `private` verbietet allen von außen zugreifenden Klassen den Zugriff auf Eigenschaften. Das wäre etwa für eine Klasse wichtig, die Passwörter speichern möchte. Dafür wollen wir eine öffentliche Klasse `Password` mit einem privaten Attribut `password` deklarieren. Eine öffentliche Methode `assign()` soll eine Änderung des Passwortes zulassen, wenn das alte Passwort bekannt ist, und eine weitere öffentliche Methode `check()` erlaubt das Prüfen eines Passwortes. Am Anfang ist das Passwort der leere String:

Listing 5.10: Password.java

```

public class Password
{
    private String password = "";

    public void assign( String oldPassword, String newPassword )
    {
        if ( password.equals(oldPassword) && newPassword != null )
        {
            password = newPassword;

            System.out.println( "Passwort gesetzt." );
        }
        else
            System.out.println( "Passwort konnte nicht gesetzt werden." );
    }

    public boolean check( String passwordToCheck )
    {
        return password.equals( passwordToCheck );
    }
}

```

Wir sehen, dass öffentliche Objektmethoden ganz selbstverständlich auf das private-Element ihrer Klasse zugreifen können.

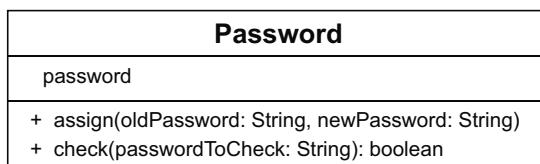


Abbildung 5.7: In der UML werden private Eigenschaften mit einem führenden Minus gekennzeichnet.

Eine zweite Klasse PasswordDemo will nun auf das Passwort von außen zugreifen:

Listing 5.11: PasswordDemo.java, main()

```
>Password pwd = new Password();
pwd.assign( "", "TeutoburgerWald" );
pwd.assign( "TeutoburgerWald", "Doppelkeks" );
pwd.assign( "Dopplerkeks", "panic" );
// System.out.println( pwd.password ); // The field Password.password is not visible
```

Die Klasse Password enthält den privaten String password, und dieser kann nicht referenziert werden. Der Compiler erkennt zur Übersetzungs- beziehungsweise Laufzeit Verstöße und meldet diese.

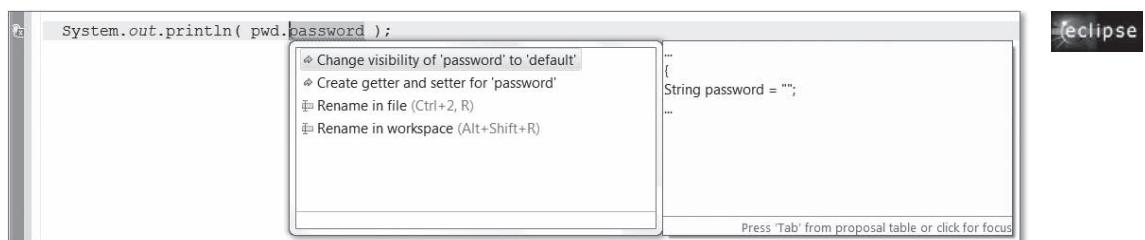


Abbildung 5.8: Bei einem durch die falsche Sichtbarkeit verursachten Fehler bietet Eclipse mit **Strg** + **1** eine Änderung der Sichtbarkeit an.

Allerdings wäre es manchmal besser, wenn der Compiler uns nicht verriete, dass das Element privat ist, sondern einfach nur melden würde, dass es dieses Element nicht gibt.

5.2.3 Wieso nicht freie Methoden und Variablen für alle?

Private Methoden und Variablen dienen in erster Linie dazu, den Klassen Modularisierungsmöglichkeiten zu geben, die von außen nicht sichtbar sein müssen. Zwecks Strukturierung werden Teilaufgaben in Methoden gegliedert, die aber von außen nie allein aufgerufen werden dürfen. Daav die Implementierung versteckt wird und der Programmierer vielleicht nur eine Zugriffsmethode sieht, wird auch der Terminus »Data Hiding« verwendet. Wer wird schon einem Fremden die Geheimzahl der Kreditkarte geben oder verraten, mit wem er die letzte Nacht verbracht hat? Oder nehmen wir zum

Beispiel ein Radio: Von außen bietet es die Methoden `an()`, `aus()`, `lauter()` und `leiser()` an, aber welche physikalischen Vorgänge ein Radio dazu bringen, Musik zu spielen, das ist eine ganz andere Frage, für die wir uns als gewöhnliche Benutzer eines Radios nicht interessieren.

5.2.4 Privat ist nicht ganz privat: Es kommt darauf an, wer's sieht *

Private Eigenschaften sind nur für andere Klassen privat, aber nicht für die eigene, auch wenn die Objekte unterschiedlich sind. Das ist eine Art Spezialfall, dass über eine Referenzvariable der Zugriff auf eine private Eigenschaft eines anderen Objekts erlaubt ist:

Listing 5.12: Key.java

```
public class Key
{
    private int id;

    public Key( int id )
    {
        this.id = id;
    }

    public boolean compare( Key that )
    {
        return this.id == that.id;
    }
}
```

Die Methode `compare()` der Klasse `Key` vergleicht das eigene Attribut `this.id` mit dem Attribut des als Argument übergebenen Objekts `that.id`. Zwar wäre `this` nicht nötig, doch verdeutlicht es schön das eigene und das fremde Objekt. An dieser Stelle sehen wir, dass der Zugriff auf `that.id` zulässig ist, obwohl `id` privat ist. Dieser Zugriff ist aber erlaubt, da die `compare()`-Methode in der `Key`-Klasse deklariert und der Parameter ebenfalls vom Typ `Key` ist. Mit Unterklassen (siehe Abschnitt 5.8.1, »Vererbung in Java«) funktioniert das schon nicht mehr. Private Attribute und Methoden sind also gegen Angriffe von außerhalb der deklarierenden Klasse geschützt.

5.2.5 Zugriffsmethoden für Attribute deklarieren

Attribute sind eine tolle und notwendige Sache. Allerdings ist zu überlegen, ob der Nutzer eines Objekts immer direkt auf die Attribute zugreifen sollte oder ob dies problematisch ist:

- Bei manchen Variablen gibt es Wertebereiche, die einzuhalten sind. Das Alter eines Spielers kann nicht kleiner null sein, und Menschen, die älter als zweihundert Jahre sind, werden nur in der Bibel genannt. Wenn wir das Alter privat machen, kann eine Zugriffsmethode wie `setAge(int)` mithilfe einer Bereichsprüfung nur bestimmte Werte in die Objektvariable übertragen und den Rest ablehnen. Die öffentliche Methode `getAge()` gibt dann Zugriff auf die Variable.
- Mit einigen Variablen sind Abhängigkeiten verbunden. Wenn zum Beispiel ein Spieler ein Alter hat, kann der Spieler gleichzeitig eine Wahrheitsvariable für die Volljährigkeit deklarieren. Natürlich gibt es nun eine Abhängigkeit. Ist der Spieler älter als 18, soll die Wahrheitsvariable auf `true` stehen. Diese Abhängigkeit lässt sich mit zwei öffentlichen Variablen nicht wirklich erzwingen. Eine Methode `setAge(int)` kann jedoch bei privaten Attributen diese Konsistenz einhalten.
- Gibt es Zugriffsmethoden, so lassen sich dort leicht Debug-Breakpoints setzen. Auch lassen sich die Methoden so erweitern, dass der Zugriff geloggt (protokolliert) wird oder die Rechte des Aufrufers geprüft werden.
- Bei Klassen gilt das Geheimnisprinzip. Obwohl es vorrangig für Methoden gilt, sollte es auch für Variablen gelten. Möchten Entwickler etwa ihr internes Attribut von `int` auf `BigInteger` ändern und damit beliebig große Ganzzahlen verwalten, hätten wir ein beträchtliches Problem, da an jeder Stelle des Vorkommens ein Objekt eingesetzt werden müsste. Wollten wir zwei Variablen einführen – ein `int`, damit die alte, derzeit benutzte Software ohne Änderung auskommt, und ein neues `BigInteger` – hätten wir ein Konsistenzproblem.
- Nicht immer müssen dem Aufrufer haarklein alle Eigenschaften angeboten werden. Der Nutzer möchte sogenannte *höherwertige Dienste* vom Objekt angeboten bekommen, sodass der Zugriff auf die unteren Attribute vielleicht gar nicht nötig ist.

5.2.6 Setter und Getter nach der JavaBeans-Spezifikation

Wir sehen an diesen Beispielen, dass es gute Gründe dafür gibt, Attribute zu privatisieren und öffentliche Methoden zum Lesen und Schreiben anzubieten. Weil diese Methoden auf die Attribute zugreifen, nennen sie sich auch *Zugriffsmethoden*. Für jedes Attribut wird eine Schreib- und Lesemethode deklariert, für die es auch ein festes

Namensschema laut JavaBeans-Konvention gibt. Die Zugriffsmethoden machen dabei eine *Property* zugänglich. Für eine Property »name« und den Typ `String` gilt zum Beispiel:

- `String getName():` Die Methode besitzt keine Parameterliste, und der Rückgabetyp ist `String`.
- `void setName(String name):` Die Methode hat keine Rückgabe, aber genau einen Parameter.

Die `getXXX()`-Methoden heißen *Getter*, die `setXXX()`-Methoden *Setter*. Die Methoden sind öffentlich, und der Typ der Getter-Rückgabe muss der gleiche wie der Parametertyp vom Setter sein.

Bei `boolean`-Attributen darf es (und muss es in manchen Fällen auch) statt `getXXX()` alternativ `isXXX()` heißen. Da die Programmierung in der Regel mit englischen Bezeichnernamen erfolgt, kommt es nicht zu unschönen Bezeichnernamen à la `getGegenstand()` oder `isVolljährig()`.



Sprachenvergleich

Während in Java Methoden, die ein `boolean` liefern, immer mit dem Präfix `is` beginnen, schließt bei Ruby ein Fragezeichen den Methodennamen ab. So liefert `".empty?"` wahr und `"java".equal? "ruby"` falsch.² Das ist eine übliche Konvention in Ruby und zeigt außerdem, dass andere Programmiersprachen mit den erlaubten Bezeichnern großzügiger sind.

Zugriffsmethoden für den Spieler

Das folgende Beispiel soll einen Spieler umsetzen, bei dem der Name und der Gegenstand privat und hübsch durch Zugriffsmethoden abgesichert sind. Eine Konsistenzprüfung soll verhindern, dass ein `String null` oder leer ist:

Listing 5.13: com/tutego/insel/game/v5/Player.java, Player

```
public class Player
{
```

² Klammern bei Methodenaufrufen sind optional. Weiterhin gilt: In Java testet der `==`-Operator die Identität und `equals()` die Gleichheit. In Ruby ist das genau anders herum: `==` testet auf Gleichheit und `equal?` auf Identität. Und dazu kommt noch die Methode `eq?`, die testet, ob zwei Objekte die gleichen Werte haben und vom gleichen Typ sind.

```

private String name = "";
private String item = "";

public String getName()
{
    return name;
}

public void setName( String name )
{
    if ( name != null && !name.trim().isEmpty() )
        this.name = name;
}

public String getItem()
{
    return item;
}

public void setItem( String item )
{
    if ( item != null && !item.trim().isEmpty() )
        this.item = item;
}

```

Es muss keine gute Idee sein, sich bei ungültigen Werten taub zu stellen; eine Alternative besteht darin, zu loggen oder etwa in Form einer Ausnahme (Exception) unerwünschte Werte zu melden.

Player	
name	
item	
+ getName(): String + setName(name: String) + getItem(): String + setItem(item: String)	

Abbildung 5.9: Player-Klasse mit Setter/Getter

Der Nutzer der Klasse muss wegen der Methodenaufrufe etwas mehr schreiben:

Listing 5.14: com/tutego/insel/game/v5/Playground.java

```
Player spongebob = new Player();
spongebob.setName( "Spongebob" );
spongebob.setItem( "Schnecke" );
```

Wird später bei der Weiterentwicklung des Programms eine Änderung notwendig, wenn etwa die Gegenstände anders gespeichert werden sollen, kann der Typ der internen Variablen geändert werden, und die Welt draußen bekommt davon nichts mit. Lediglich intern in den Gettern und Settern ändert sich etwas, aber nicht an der Schnittstelle.



Eclipse bietet zwei Möglichkeiten, Setter und Getter automatisch zu generieren. Das Kontextmenü unter SOURCE • GENERATE GETTERS AND SETTERS... fördert ein Dialogfenster zutage, mit dem Eclipse automatisch die setXXX()- und getXXX()-Methoden einfügen kann. Die Attribute, für die eine Zugriffsmethode gewünscht ist, werden selektiert. Die zweite Möglichkeit funktioniert nur für genau ein Attribut: Steht der Cursor auf der Variablen, liefert REFACTOR • ENCAPSULATE FIELD... einen Dialog, mit dem zum einen Setter und Getter generiert werden und zum anderen die direkten Zugriffe auf das Attribut in Methodenaufrufe umgewandelt werden.

5.2.7 Paketsichtbar

Die Sichtbarkeiten `public` und `private` sind Extreme. Steht kein ausdrücklicher Sichtbarkeitsmodifizierer an den Eigenschaften, gilt die *Paketsichtbarkeit*. Sie sagt aus, dass die paketsichtbaren Klassen nur von anderen Klassen im gleichen Paket gesehen werden können. Für die Eigenschaften gilt das Gleiche: Nur Typen im gleichen Paket sehen die paketsichtbaren Eigenschaften.

Dazu einige Beispiele: Zwei Klassen, A und B, befinden sich in unterschiedlichen Paketen. Die Klasse A ist nicht öffentlich:



Abbildung 5.10: Vier Klassen, verteilt auf zwei Pakete, für das Beispiel

Listing 5.15: com/tutego/insel/protecteda/A.java

```
package com.tutego.insel.protecteda;  
class A  
{  
}
```

Die Klasse B versucht, sich auf A zu beziehen, doch funktioniert das wegen der »Unsichtbarkeit« von A nicht – es gibt einen Compilerfehler:

Listing 5.16: com/tutego/insel/protectedb/B.java

```
package com.tutego.insel.protectedb;  
class B  
{  
    A a;           // ☹ A cannot be resolved to a type  
}
```

Ist eine Klasse C öffentlich, aber die Klasse deklariert ein nur paketsichtbares Attribut c, dann kann eine Klasse in einem anderen Paket das Attribut nicht sehen, auch wenn sie die Klasse selbst sehen kann:

Listing 5.17: com/tutego/insel/protecteda/C.java

```
package com.tutego.insel.protecteda;  
public class C  
{  
    static int c;  
}
```

Und dies ist die Klasse D:

Listing 5.18: com/tutego/insel/protectedb/D.java

```
package com.tutego.insel.protectedb;  
import com.tutego.insel.protecteda.C;  
class D  
{  
    int d = C.c;    // ☹ The field C.c is not visible  
}
```

Paketsichtbare Eigenschaften sind sehr nützlich, weil sich damit Gruppen von Typen bilden lassen, die gegenseitig Teile ihres Innenlebens kennen. Von außerhalb des Pakets ist der Zugriff auf diese Teile dann untersagt, analog zu `private`. Dazu ein Beispiel für unsere Spielerklasse: Nutzt der Player zum Beispiel intern eine Hilfsklasse, etwa zur Speicherung der Gegenstände, so kann diese paketsichtbare Speicherklasse für Außenstehende unsichtbar bleiben.

5.2.8 Zusammenfassung zur Sichtbarkeit

In Java gibt es vier Sichtbarkeiten und dafür drei Sichtbarkeitsmodifizierer:

- Öffentliche Typen und Eigenschaften deklariert der Modifizierer `public`. Die Typen sind überall sichtbar, also kann jede Klasse und Unterklasse aus einem beliebigen anderen Paket auf öffentliche Eigenschaften zugreifen. Die mit `public` deklarierten Methoden und Variablen sind überall dort sichtbar, wo auch die Klasse sichtbar ist. Bei einer unsichtbaren Klasse sind auch die Eigenschaften unsichtbar.
- Der Modifizierer `private` ist bei Typdeklarationen seltener, da er sich nur dann einsetzen lässt, wenn in einer Kompilationseinheit (also einer Datei) mehrere Typen deklariert werden. Derjenige Typ, der den Dateinamen bestimmt, kann nicht privat sein, doch andere Typen (und innere Klassen) dürfen unsichtbar sein – nur der sichtbare Typ kann sie dann verwenden. Die mit `private` deklarierten Methoden und Variablen sind nur innerhalb der eigenen Klasse sichtbar. Eine Ausnahme bilden innere Klassen, die auch auf private Eigenschaften der äußeren Klasse zugreifen können. Wenn eine Klasse erweitert wird, sind die privaten Elemente für Unterklassen nicht sichtbar.
- Während `private` und `public` Extreme darstellen, liegt die Paketsichtbarkeit dazwischen. Sie ist die Standard-Sichtbarkeit und kommt ohne Modifizierer aus. Paketsichtbare Typen und Eigenschaften sind nur für die Klassen aus dem gleichen Paket sichtbar, also weder für Klassen noch für Unterklassen aus anderen Paketen.
- Der Sichtbarkeitsmodifizierer `protected` hat eine Doppelfunktion: Zum einen hat er die gleiche Bedeutung wie Paketsichtbarkeit, und zum anderen gibt er die Elemente für Unterklassen frei. Dabei ist es egal, ob die Unterklassen aus dem eigenen Paket stammen (hier würde ja die Standard-Sichtbarkeit reichen) oder aus einem anderen Paket. Eine Kombination aus `private protected` wäre wünschenswert, um die Eigen-

schaften nur für die Unterklassen sichtbar zu machen und nicht gleich für die Klassen aus dem gleichen Paket.³

Eigenschaft ist	Sieht eigene Klasse	Sieht Klasse im gleichen Paket	Sieht UnterkLASSE im anderen Paket	Sieht Klasse in anderem Paket
public	ja	ja	ja	ja
protected	ja	ja	ja	nein
paketsichtbar	ja	ja	nein	nein
private	ja	nein	nein	nein

Tabelle 5.2: Wer sieht welche Eigenschaften bei welcher Sichtbarkeit?

Der Einsatz der Sichtbarkeitsstufen über die Schlüsselwörter `public`, `private` und `protected` und den Standard »paket-sichtbar« ohne explizites Schlüsselwort sollte überlegt erfolgen. Die objektorientierte Programmierung zeichnet sich durch überlegten Einsatz von Klassen und deren Beziehungen aus. Am besten ist die restriktivste Beschreibung; also nie mehr Öffentlichkeit als notwendig. Das hilft, die Abhängigkeiten zu minimieren und später Inneres einfacher zu verändern.

Sichtbarkeit in der UML *

Für die Sichtbarkeit von Attributen und Operationen sieht die UML diverse Symbole vor, die vor die jeweilige Eigenschaft gesetzt werden:

Symbol	Sichtbarkeit
+	Öffentlich
-	Privat

Tabelle 5.3: UML-Symbole für die Sichtbarkeit von Attributen und Operationen

3 Die Java Programmers' FAQ (http://groups.google.com/group/comp.lang.java.programmer/browse_thread/thread/5f5ae8700cc63291/977b7eddd7064217) führt aus: »It first appeared in JDK 1.0 FCS (it had not been in the Betas). Then it was removed in JDK 1.0.1. It was an ugly hack syntax-wise, and it didn't fit consistently with the other access modifiers. It never worked properly: in the versions of the JDK before it was removed, calls to private protected methods were not dynamically bound, as they should have been. It added very little capability to the language. It's always a bad idea to reuse existing keywords with a different meaning. Using two of them together only compounds the sin. The official story is that it was a bug. That's not the full story. Private protected was put in because it was championed by a strong advocate. It was pulled out when he was overruled by popular acclamation.«

Symbol	Sichtbarkeit
#	geschützt (protected)
~	Paketsichtbar

Tabelle 5.3: UML-Symbole für die Sichtbarkeit von Attributen und Operationen (Forts.)

**Hinweis**

Wenn in der UML kein Sichtbarkeitsmodifizierer steht, so bedeutet das nicht »paket-sichtbar«. Es heißt nur, dass dies noch nicht definiert ist.

Reihenfolge der Eigenschaften in Klassen *

Verschiedene Elemente einer Klasse müssen in einer Klasse untergebracht werden. Eine verbreitete Reihenfolge ist die Aufteilung in Sektionen:

- Klassenvariablen
- Objektvariablen
- Konstruktoren
- statische Methoden
- Setter/Getter
- beliebige Objektmethoden

Innerhalb eines Blocks werden die Informationen oft auch bezüglich ihrer Zugriffsrechte sortiert. Am Anfang stehen sichtbare Eigenschaften und tiefer private. Der öffentliche Teil befindet sich deswegen am Anfang, da wir uns auf diese Weise schnell einen Überblick verschaffen können. Der zweite Teil ist dann nur noch für die erbenden Klassen interessant, und der letzte Teil beschreibt allein geschützte Informationen für die Entwickler. Die Reihenfolge kann aber problemlos gebrochen werden, indem private Methoden hinter öffentlichen stehen, um zusammenhängende Teile auch zusammenzuhalten.

**Codestil**

Quellcode sollte immer mit Leerzeichen statt mit Tabulatoren eingerückt werden. Zwei oder vier Leerzeichen sind oft anzutreffen. Viele Entwickler setzen die öffnende geschweifte Klammer für den Beginn eines Blocks gerne alleinstehend in die nächste Zeile. In Eclipse und NetBeans kann ein Code-Formatierer den Quellcode automatisch korrigieren.

5.3 Statische Methoden und statische Attribute

Exemplarvariablen sind eng mit ihrem Objekt verbunden. Wird ein Objekt geschaffen, erhält es einen eigenen Satz von Exemplarvariablen, die zusammen den Zustand des Objekts repräsentieren. Ändert eine Objektmethode den Wert einer Exemplarvariablen in einem Objekt, so hat dies keine Auswirkungen auf die Daten der anderen Objekte; jedes Objekt speichert eine individuelle Belegung. Es gibt jedoch auch Situationen, in denen Eigenschaften oder Methoden nicht direkt einem individuellen Objekt zugeordnet werden. Dazu gehören zum Beispiel die statischen Methoden:

- `Math.max()` liefert das Maximum zweier Zahlen.
- `Math.sin()` bestimmt den Sinus.
- `Integer.parseInt()` konvertiert einen String in eine Ganzzahl.
- `JOptionPane.showInputDialog()` zeigt einen Eingabedialog.
- `Color.HSBtoRGB()`⁴ konvertiert Farben vom HSB-Farbraum in den RGB-Farbraum.

Dazu gesellen sich Zustände, die nicht an ein individuelles Objekt gebunden sind:

- `Integer.MAX_VALUE` ist die größte darstellbare `int`-Ganzzahl.
- `Math.PI` bestimmt die Zahl 3,1415...
- `Font.MONOSPACED` steht für einen Zeichensatz mit fester Breite.
- `MediaSize.ISO.A4` definiert die Größe einer DIN-A4-Seite mit 210 mm × 297 mm.

Diese genannten Eigenschaften sind keinem konkreten Objekt zugeordnet, sondern vielmehr der Klasse. Diese Art von Zugehörigkeit wird in Java durch *statische Eigenschaften* unterstützt. Da sie zu keinem Objekt gehören (wie Objekteigenschaften), nennen wir sie auch *Klasseneigenschaften*. Die Sinus-Methode ist ein Beispiel für eine statische Methode der `Math`-Klasse, und `MAX_INTEGER` ist ein statisches Attribut der Klasse `Integer`.

5.3.1 Warum statische Eigenschaften sinnvoll sind

Statische Eigenschaften haben gegenüber Objekteigenschaften den Vorteil, dass sie im Programm ausdrücken, keinen Zustand vom Objekt zu nutzen. Betrachten wir noch einmal die statischen Methoden aus der Klasse `Math`. Wenn sie Objektmethoden wären, so würden sie in der Regel mit einem Objektzustand arbeiten. Die statischen Methoden

⁴ Ja, die Methode beginnt unüblicherweise mit einem Großbuchstaben.

hätten keine Parameter und nähmen ihre Arbeitswerte nicht aus den Argumenten, sondern aus dem internen Zustand des Objekts. Das macht aber keine Math-Methode. Um den Sinus eines Winkels zu berechnen, benötigen wir kein spezifisches Mathe-Objekt. Andersherum könnte eine Methode wie `setName()` eines Spielers nicht statisch sein, da der Name ganz individuell für einen Spieler gesetzt werden soll und nicht alle Spieler-Objekte immer den gleichen Namen tragen sollten.

Statische Methoden sind aus diesem Grund häufiger als statische Variablen, da sie ihre Arbeitswerte ausschließlich aus den Parametern ziehen. Statische Variablen werden in erster Linie als Konstanten verwendet.

5.3.2 Statische Eigenschaften mit static

Um statische Eigenschaften in Java umzusetzen, fügen wir vor der Deklaration einer Variablen oder einer Methode das Schlüsselwort `static` hinzu. Für den Zugriff verwenden wir statt der Referenzvariablen einfach den Klassennamen. In der UML sind statische Eigenschaften unterstrichen gekennzeichnet.

Deklarieren wir eine statische Methode und eine statische Variable für eine Klasse `GameUtils`. Die Methode soll testen, ob Bezeichner, die im Spiel etwa für die Gegenstände verwendet werden, korrekt sind; ein korrekter Bezeichner ist nicht zu lang und enthält kein Sonderzeichen. Die Konstante `MAX_ID_LEN` steht für die maximale Bezeichnerlänge. Die Variable ist mit dem Modifizierer `final` versehen, da `MAX_ID_LEN` eine Konstante ist, deren Wert später nicht mehr verändert werden soll:

Listing 5.19: GameUtils.java

```
public class GameUtils
{
    public static final int MAX_ID_LEN = 20 /* chars */;

    public static boolean isGameIdentifier( String name )
    {
        if ( name == null )
            return false;

        return name.length() <= MAX_ID_LEN && name.matches( "\\\w+" );
    }
}
```

GameUtils
+ MAX_ID_LEN: int
+ isGameIdentifier(String): boolean

Abbildung 5.11: Statische Eigenschaften werden in der UML unterstrichen.

Die statischen Eigenschaften werden mit dem Klassennamen GameUtils angesprochen:

Listing 5.20: GameUtilsDemo.java, main() Ausschnitt

```
System.out.println( GameUtils.isGameIdentifier( "Superpig" ) ); // true
System.out.println( GameUtils.isGameIdentifier( "Superpig II" ) ); // false
```

Tipp

Falls eine Klasse nur statische Eigenschaften deklariert, spricht nichts dagegen, einen privaten Konstruktor anzugeben – das verhindert den äußeren Aufbau von Objekten. Eigene Konstruktoren werden etwas später vorgestellt, sodass unsere Klasse diese Möglichkeit noch nicht nutzt.



Abschnitt 5.5.1, »Konstruktoren schreiben«, erklärt Konstruktoren genauer und erläutert auch weitere Anwendungsfälle für private Konstruktoren.

Gültigkeitsbereich, Sichtbarkeit und Lebensdauer

Bei statischen und nicht-statischen Variablen können wir deutliche Unterschiede in der Lebensdauer festmachen. Eine Objektvariable beginnt ihr Leben mit dem new, und sie endet mit dem GC. Eine statische Variable dagegen beginnt ihr Leben in dem Moment, in dem die Laufzeitumgebung die Klasse lädt und initialisiert. Das Leben der statischen Variable endet, wenn die JVM die Klasse entfernt und aufräumt. Der Zugriff auf statische Variablen ist immer in allen Blöcken gestattet, da ja auch in Objektmethoden die statische Variable »schon eher da war« als das Objekt selbst, denn ein new setzt ja die geladene Klassendefinition, die die statischen Variablen vorbereitet, voraus.

5.3.3 Statische Eigenschaften über Referenzen nutzen? *

Besitzt eine Klasse eine Klasseneigenschaft, so kann sie auch wie ein Objektattribut über die Referenz angesprochen werden. Dies bedeutet, dass es prinzipiell zwei Möglichkeiten gibt, wenn ein Objektexemplar existiert und die Klasse ein statisches Attribut hat.

Bleiben wir bei unserem obigen Beispiel mit der Klasse GameUtils. Wir können für den Zugriff auf MAX_ID_LEN Folgendes schreiben:

Listing 5.21: GameUtilsDemo.java, main() Ausschnitt

```
System.out.println( GameUtils.MAX_ID_LEN );           // Genau richtig
GameUtils ut = new GameUtils();
System.out.println( ut.MAX_ID_LEN );                  // Nicht gut
```

Zugriffe auf statische Eigenschaften sollten wir nie über die Objektreferenz schreiben, denn dem Leser ist sonst nicht klar, ob die Eigenschaft statisch oder nicht-statisch ist. Das zu wissen, ist aber wichtig. Aus diesem Grund sollten wir immer statische Eigenschaften über ihren Klassennamen ansprechen; Eclipse gibt hier auch eine Meldung aus, wenn wir es nicht so machen.



Hinweis

Bei statischen Zugriffen spielt die Referenz keine Rolle, und sie kann auch null sein.

Wer im Wettbewerb um das schlechteste Java-Programm weit vorne sein möchte, der schreibt:

```
System.out.println( ((GameUtils) null).MAX_ID_LEN );
```

5.3.4 Warum die Groß- und Kleinschreibung wichtig ist *

Die Vorgabe der Namenskonvention besagt: Klassennamen sind mit Großbuchstaben zu vergeben und Variablennamen mit Kleinbuchstaben. Treffen wir auf eine Anweisung wie Math.max(a, b), so wissen wir sofort, dass max() eine statische Methode sein muss, weil davor ein Bezeichner steht, der großgeschrieben ist. Dieser kennzeichnet also keine Referenz, sondern einen Klassennamen. Daher sollten wir in unseren Programmen großgeschriebene Objektnamen meiden.

Das folgende Beispiel demonstriert anschaulich, warum Referenzvariablen mit Kleinbuchstaben und Klassennamen mit Großbuchstaben beginnen sollten:

```
String StringModifier = "What is the Matrix?";
String t = StringModifier.trim();
```

Die trim()-Methode ist nicht statisch, wie die Anweisung durch die Großschreibung der Variable suggeriert.

Das gleiche Problem haben wir, wenn wir Klassen mit Kleinbuchstaben benennen. Auch dies kann irritieren:

```
class player
{
    static void move() { }
}
```

Jetzt könnte jemand `player.move()` schreiben, und der Leser nähme an, dass `player` wegen seiner Kleinschreibung eine Referenzvariable ist und `move()` eine Objektmethode. Wir sehen an diesem Beispiel, dass es wichtig ist, sich an die Groß-/Kleinschreibung zu halten.

5.3.5 Statische Variablen zum Datenaustausch *

Der Wert einer statischen Variable wird bei dem Klassenobjekt gespeichert und nicht bei einem Exemplar der Klasse. Wie wir aber gesehen haben, kann jedes Exemplar einer Klasse auch auf die statischen Variablen der Klasse zugreifen. Da eine statische Variable aber nur einmal pro Klasse vorliegt, führt dies dazu, dass mehrere Objekte sich eine Variable teilen. Ist etwa das Attribut `PI` statisch und `size` ein Objektattribut, so ergibt sich bei zwei Exemplaren folgendes Bild:

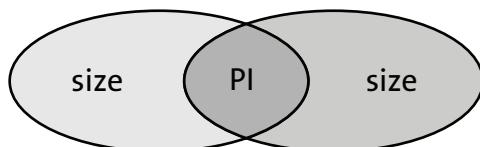


Abbildung 5.12: Zwei Klassen teilen sich das statische Attribut PI.

Während also beide Exemplare das gleiche `PI` nutzen, können beide Exemplare `size` völlig unterschiedlich belegen.

Mit diesem Wissen wird es möglich, einen Austausch von Informationen über die Objektgrenze hinaus zu erlauben:

Listing 5.22: ShareData.java

```
public class ShareData
{
    private static int data;
```

```
public void memorize( int data )
{
    ShareData.data = data;
}

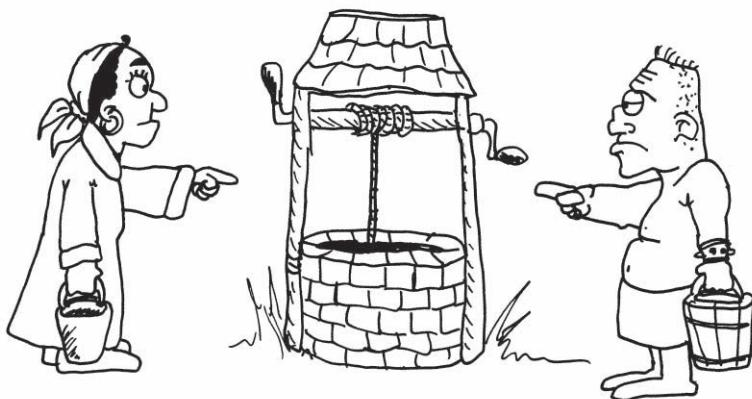
public int retrieve ()
{
    return data;
}

public static void main( String[] args )
{
    ShareData s1 = new ShareData();
    ShareData s2 = new ShareData();
    s1.memorize( 2 );
    System.out.println( s2.retrieve() );    // 2
}
}
```



Hinweis

Bei nebenläufigen Zugriffen auf statische Variablen kann es zu Problemen kommen. Deshalb müssen wir spezielle Synchronisationsmechanismen nutzen – die das Beispiel allerdings nicht verwendet. Statische Variablen können auch schnell zu Speicherproblemen führen, da Objektreferenzen sehr lange gehalten werden. Der Einsatz muss wohl-durchdacht sein.



5.3.6 Statische Eigenschaften und Objekteigenschaften *

Wie wir oben gesehen haben, können wir über eine Objektreferenz auch statische Eigenschaften nutzen. Wir wollen uns aber noch einmal vergewissern, wie Objekteigenschaften und statische Eigenschaften gemischt werden können. Erinnern wir uns daran, dass unsere ersten Programme aus der statischen `main()`-Methode bestanden, aber unsere anderen Methoden auch `static` sein mussten. Dies ist sinnvoll, da eine statische Methode – ohne explizite Angabe eines aufrufenden Objekts – nur andere statische Methoden aufrufen kann. Wie sollte auch eine statische Methode eine Objektmethode aufrufen können, wenn es kein zugehöriges Objekt gibt? Andersherum kann aber jede Objektmethode eine beliebige statische Methode direkt aufrufen. Genauso verhält es sich mit Attributen. Eine statische Methode kann keine Objektattribute nutzen, da es kein implizites Objekt gibt, auf dessen Eigenschaften zugegriffen werden könnte.

»this«-Referenzen und statische Eigenschaften

Auch der Einsatz der `this`-Referenz ist bei statischen Eigenschaften nicht möglich. Eine statische Methode kann also keine `this`-Referenz verwenden:

Listing 5.23: InStaticNoThis.java

```
class InStaticNoThis
{
    String name;

    static void setName()
    {
        name = "Amanda";           // ☹ Compilerfehler
        this.name = "Amanda";      // ☹ Compilerfehler
    }
}
```

5.4 Konstanten und Aufzählungen

In Programmen gibt es Variablen, die sich ändern (wie zum Beispiel ein Schleifenzähler), aber auch andere, die sich beim Ablauf eines Programms nicht ändern. Dazu gehören etwa die Startzeit der Tagesschau oder die Ausmaße einer DIN-A4-Seite⁵. Die Werte

⁵ Ein DIN-A4-Blatt ist 29,7 cm hoch und 21,0 cm breit.

sollten nicht wiederholt im Quellcode stehen, sondern über ihre Namen angesprochen werden. Dazu werden Variablen deklariert, denen genau der konstante Wert zugewiesen wird; die Konstanten heißen dann *symbolische Konstanten*.

In Java gibt es zur Deklaration von Konstanten zwei Möglichkeiten:

- Öffentliche statische finale Variablen nehmen konstante Werte auf.
- Aufzählungen über ein `enum` (die intern aber auch nur öffentliche finale statische Werte sind).

5.4.1 Konstanten über öffentliche statische finale Variablen

Statische Variablen werden auch verwendet, um symbolische Konstanten zu deklarieren. Damit die Variablen unveränderlich bleiben, gesellt sich der Modifizierer `final` hinzu. Dem Compiler wird auf diese Weise mitgeteilt, dass dieser Variable nur einmal ein Wert zugewiesen werden darf. Für Variablen bedeutet dies: Es sind Konstanten; jeder spätere Schreibzugriff wäre ein Fehler.

Konstante Werte haben wir schon bei `GameUtils` eingesetzt:

Listing 5.24: GameUtils, Ausschnitt

```
public class GameUtils
{
    public static final int MAX_ID_LEN = 20 /* chars */;

    ...
}
```

Da im Quellcode das Vorkommen von Zahlen wie der 20 undurchsichtig wäre, sind symbolische Namen zwingend. Stehen dennoch Zahlen ohne offensichtliche Bedeutung im Quellcode, so werden sie *magische Zahlen* (engl. *magic numbers*) genannt. Es gilt, diese Werte in Konstanten zu fassen und sinnvoll zu benennen.



Tipp

Es ist eine gute Idee, die Namen von Konstanten durchgehend großzuschreiben, um ihre Bedeutung hervorzuheben.

Der Zugriff auf die Variablen sieht genauso aus wie ein Zugriff auf andere statische Variablen.

zB

5

Beispiel

Greife auf Konstanten zurück:

```
System.out.println( Math.PI );
int len = GameUtils.MAX_ID_LEN;
if ( s.length() > GameUtils.MAX_ID_LEN )
    System.out.println( "Zu lang!" );
```

5.4.2 Typ(un)sichere Aufzählungen *

Konstanten sind eine wertvolle Möglichkeit, den Quellcode aussagekräftiger zu machen. Das gilt auch für Aufzählungen, also diverse Konstanten, die für unterschiedliche Ausprägungen stehen. Eine Klasse `Materials` soll zum Beispiel Konstanten für die Beschaffenheit eines Materials deklarieren:

Listing 5.25: Materials.java

```
public class Materials
{
    public static final int SOFT      = 0;
    public static final int HARD     = 1;
    public static final int DRY      = 2;
    public static final int WET      = 3;
    public static final int SMOOTH   = 4;
    public static final int ROUGH   = SMOOTH + 1;
}
```

Für ihre Belegungen ist es günstig, die Konstanten relativ zum Vorgänger zu wählen, um das Einfügen in der Mitte zu vereinfachen. Das sehen wir bei der letzten Variablen, `ROUGH`.

Materials
+ SOFT: int
+ HARD: int
+ DRY: int
+ WET: int
+ SMOOTH: int
+ ROUGH: int

Abbildung 5.13: Die Klasse `Materials` mit Konstanten

Einfache Konstantentypen – wie bei uns `int` – bringen jedoch den Nachteil mit sich, dass die Konstanten nicht unbedingt von jedem angewendet werden müssen und ein Programmierer die Zahlen oder Zeichenketten eventuell direkt einsetzt. Dieses Problem ergibt sich zum Beispiel dann, wenn ein `Font`-Objekt für die grafische Oberfläche angelegt werden soll, aber unser Gedächtnis versagt, in welcher Reihenfolge die Parameter zu füllen sind. Ein Fallbeispiel:

```
Font f = new Font( "Dialog" , 12 , Font.BOLD );
```

Leider ist dies falsch, denn die Argumente für die Zeichensatzgröße und den Schriftstil sind vertauscht: Es müsste `new Font("Dialog", Font.BOLD, 12)` heißen. Das Problem ist, dass die Konstanten nur Namen für Werte eines frei zugänglichen Grundtyps (auch hier `int`) sind und nur die Variablenbelegung, also der Wert, an den Konstruktor übergeben wird. Niemand kann verbieten, dass die Werte direkt eingetragen werden. Das führt dann zu Fehlern wie im oberen Fall. In diesem ist `12` die Ganzzahl für den Schriftstil, obwohl es dafür nur die Werte `0, 1, 2` geben sollte. Mit Zeichenketten als Werten der Konstanten kommen wir der Lösung auch nicht näher.



Hinweis

Ganzzahlen haben aber durchaus ihren Vorteil, wenn es Mischungen von Aufzählungen gibt, also etwa ein hartes und ein raues Material. Das lässt sich durch `Materials.HARD + Materials.ROUGH` darstellen – was aber nur dann gut funktioniert, wenn jede Konstante ein Bit im Wort einnimmt, wenn also die Werte der Konstanten `1, 2, 4, 8, 16, ...` lauten.

Eine gute Möglichkeit, von Ganzzahlen wegzukommen, besteht darin, Objekte einer Klasse als Konstanten einzusetzen. Java bietet seit der Version 5 über das neue Schlüsselwort `enum` ein Sprachkonstrukt für richtige Aufzählungen.



Geschichte

Sun hat zu Beginn der Entwicklung von Java diverse Schlüsselwörter reserviert, aber `enum` war nicht seit Beginn dabei. Als dann in Java 5 plötzlich ein neues Schlüsselwort hinzukam, mussten Entwickler viel Quellcode anpassen und Variablennamen ändern, denn für den Variablentyp `java.util.Enumeration` wurde gern der Variablenname »`enum`« gewählt.

5.4.3 Aufzählungen mit enum

Die Schreibweise für Aufzählungen erinnert ein wenig an die Deklaration von Klassen, nur dass das Schlüsselwort `enum` statt `class` gebraucht wird. Aufzählungen für Wochentage sind ein gutes Beispiel:

Listing 5.26: com/tutego/weekday/Weekday.java, Weekday

```
public enum Weekday
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Die Konstantennamen werden wie üblich großgeschrieben.



Abbildung 5.14: Eine Aufzählung in UML

Enums nutzen

Um zu verstehen, wie sich Enums nutzen lassen, ist es hilfreich, zu wissen, wie der Compiler sie umsetzt. Intern erstellt der Compiler eine normale Klasse, in unserem Fall `Weekday`. Alle Aufzählungselemente sind dann statische Variablen (Konstanten) vom Typ des Enums:

```
public class Weekday
{
    public static final Weekday MONDAY;
    public static final Weekday TUESDAY;
    ...
}
```

Jetzt ist es einfach, diese Werte zu nutzen, da sie wie jede andere statische Variable angesprochen werden:

```
Weekday day = Weekday.SATURDAY;
```

Hinter den Aufzählungen stehen Objekte, die sich – wie alle anderen – weiterverarbeitet lassen.

```
if ( day == Weekday.MONDAY )
    System.out.println( "'I hate Mondays' (Garfield)" );
```

Enum-Vergleiche mit ==

Wie die Umsetzung der `Enum`-Typen zeigt, wird für jede Konstante ein Objekt konstruiert, und das sind `Singletons`, also Objekte, die nur einmal erzeugt werden. Eigene neue `Enum`-Objekte können wir nicht aufbauen, da die Klasse nur einen privaten Konstruktor deklariert. Der Zugriff auf dieses Objekt ist wie ein Zugriff auf eine statische Variable. Der Vergleich zweier Konstanten läuft somit auf den Vergleich von statischen Referenzvariablen hinaus, wofür der Vergleich mit `==` völlig korrekt ist. Ein `equals()` ist nicht nötig.

zB Beispiel

Eine Methode soll entscheiden, ob ein Tag das Wochenende einläutet:

```
public static boolean isWeekEnd( Weekday day )
{
    return day == Weekday.SATURDAY || day == Weekday.SUNDAY;
```

enum-Konstanten in switch

`enum`-Konstanten sind in `switch`-Anweisungen möglich. Das ist möglich, da sie intern über eine Ganzzahl als Identifizierer verfügen, den der Compiler für die Aufzählung einsetzt. Das ist ein ähnliches Konzept, wie es der Compiler ab Java 7 auch bei Strings verfolgt.

Initialisieren wir eine Variable vom Typ `Weekday`, und nutzen wir eine Fallunterscheidung mit der Aufzählung für einen Test auf das Wochenende:

Listing 5.27: WeekdayDemo.java, Ausschnitt main()

```
Weekday day = Weekday.MONDAY;
switch ( day )
{
    case SATURDAY: // nicht Weekday.SATURDAY!
    case SUNDAY: System.out.println( "Wochenende. Party!" );
}
```

Dass `case Weekday.SATURDAY` nicht möglich ist, erklärt sich dadurch, dass mit `switch (day)` schon der Typ `Weekday` über die Variable `day` bestimmt ist. Es ist nicht möglich, dass der Typ der `switch`-Variablen vom Typ der Variablen in `case` abweicht.

Referenzen auf Enum-Objekte können null sein

Dass die Aufzählungen nur Objekte sind, hat eine wichtige Konsequenz. Blicken wir zunächst auf eine Variablen Deklaration vom Typ eines `enum`, die mit einem Wochentag initialisiert ist:

```
Weekday day = Weekday.MONDAY;
```

Die Variable `day` speichert einen Verweis auf das `Weekday.MONDAY`-Objekt. Das Unschöne an Referenz-Variablen ist allerdings, dass sie auch mit `null` belegt werden können, was so gesehen kein Element der Aufzählung ist:

```
Weekday day = null;
```

Wenn solch eine `null`-Referenz in einem `switch` landet, gibt es eine `NullPointerException`, da versteckt im `switch` ein Zugriff auf die im Enum-Objekt gespeicherte Ordinalzahl stattfindet.

Methoden, die Elemente einer Aufzählung, also Objektverweise, entgegennehmen, sollten im Allgemeinen auf `null` testen und eine Ausnahme auslösen, um diesen fehlerhaften Teil anzuzeigen:

```
public void setWeekday( Weekday day )
{
    if ( day == null )
        throw new IllegalArgumentException( "null is not a valid argument!" );
    this.day = day;
}
```

Aufzählungen als inneren Typ deklarieren *

Es gibt »normale« Aufzählungen, die an normale Klassen erinnern, und auch »innere« Aufzählungen, die an innere Klassen erinnern. Mit anderen Worten: `Weekday` kann auch innerhalb einer anderen Klasse deklariert werden. Ist die innere Aufzählung öffentlich, kann jeder sie nutzen. Sie folgt aber den gleichen Sichtbarkeiten wie Klassen, da Aufzählungen ja nichts anderes als Klassen sind, die der Compiler generiert.

Statische Imports von Aufzählungen *

Die Aufzählung Weekday hatten wir in das Paket com.tutego.weekday gesetzt. Um auf eine Konstante wie MONDAY zugreifen zu können, wollen wir unterschiedliche import-Varianten nutzen.

Import-Anweisung	Zugriff
import com.tutego.weekday.Weekday;	Weekday.MONDAY
import com.tutego.weekday.*;	Weekday.MONDAY
import static com.tutego.weekday.Weekday.*;	MONDAY

Tabelle 5.4: Welche Zugriffe sind mit welchen import-Deklarationen möglich?

Nehmen wir im Paket als zweites Beispiel eine innere Aufzählung der Klasse Week hinzu:

Listing 5.28: com/tutego/weekday/Week.java

```
package com.tutego.weekday;
```

```
public class Week
{
    public enum Weekday
    {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
    }
}
```

Import-Anweisung	Zugriff
import com.tutego.weekday.Week;	Week.Weekday.MONDAY
import com.tutego.weekday.Week.Weekday;	Weekday.MONDAY
import static com.tutego.weekday.Week.Weekday.*	MONDAY

Tabelle 5.5: Welche Zugriffe sind mit welchen import-Deklarationen möglich?

Standard-Methoden der Enums *

Die erzeugten Enum-Objekte bekommen standardmäßig eine Reihe von zusätzlichen Eigenschaften. Wir überschreiben sinnvoll `toString()`, `hashCode()` und `equals()` aus `Object` und implementieren zusätzlich `Serializable` und `Comparable`,⁶ aber nicht `Clone-`

⁶ Die Ordnung der Konstanten ist die Reihenfolge, in der sie geschrieben sind.

able, da Aufzählungsobjekte nicht geklont werden können. Die Methode `toString()` liefert den Namen der Konstante, sodass `Weekday.SUNDAY.toString().equals("SUNDAY")` wahr ist. Zusätzlich erbt jedes Aufzählungsobjekt von der Spezialklasse `Enum`, die in Abschnitt 8.5, »Die Spezial-Oberklasse `Enum`«, näher erklärt wird.

5.5 Objekte anlegen und zerstören

Wenn Objekte mit dem `new`-Operator angelegt werden, reserviert die Speicherverwaltung des Laufzeitsystems auf dem System-Heap Speicher. Wird das Objekt nicht mehr referenziert, so räumt der *Garbage-Collector* (GC) in bestimmten Abständen auf und gibt den Speicher an das Laufzeitsystem zurück.

5.5.1 Konstruktoren schreiben

Wenn der `new`-Operator ein Objekt anlegt, wird ein Konstruktor der Klasse automatisch aufgerufen. Mit einem eigenen Konstruktor lässt sich erreichen, dass ein Objekt nach seiner Erzeugung einen sinnvollen Anfangszustand aufweist. Dies kann bei Klassen, die Variablen beinhalten, notwendig sein, weil sie ohne vorherige Zuweisung beziehungsweise Initialisierung keinen Sinn ergäben.

Konstruktorendeklärungen

Konstruktorendeklärungen sehen ähnlich wie Methodendeklarationen aus – so gibt es auch Sichtbarkeiten und Überladung –, doch bestehen zwei deutliche Unterschiede:

- Konstruktoren tragen immer denselben Namen wie die Klasse.
- Konstruktorendeklärungen besitzen keinen Rückgabetyp, also noch nicht einmal `void`.

Sollte eine Klasse `Dungeon` einen Konstruktor bekommen, schreiben wir:

```
class Dungeon
{
    Dungeon()                  // Konstruktor der Klasse Dungeon
    {
    }
}
```

Ein Konstruktor, der keinen Parameter besitzt, nennt sich *Standard-Konstruktor, parameterloser Konstruktor* oder auch auf Englisch *no-arg-constructor* beziehungsweise *nullary constructor*.

Aufrufreihenfolge

Dass der Konstruktor während der Initialisierung und damit vor einem äußeren Methodenaufruf aufgerufen wird, soll ein kleines Beispiel zeigen:

Listing 5.29: Dungeon.java

```
public class Dungeon
{
    public Dungeon()
    {
        System.out.println( "2. Konstruktor" );
    }

    public void play()
    {
        System.out.println( "4. Spielen" );
    }

    public static void main( String[] args )
    {
        System.out.println( "1. Vor dem Konstruktor" );
        Dungeon d = new Dungeon();
        System.out.println( "3. Nach dem Konstruktor" );
        d.play();
    }
}
```

Die Aufrufreihenfolge auf dem Bildschirm ist:

1. Vor dem Konstruktor
2. Konstruktor
3. Nach dem Konstruktor
4. Spielen



Hinweis

UML kennt zwar Attribute und Operationen, aber keine Konstruktoren im Java-Sinne. In einem UML-Diagramm werden Konstruktoren wie Operationen gekennzeichnet, die eben nur so heißen wie die Klasse.

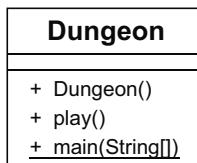


Abbildung 5.15: Die Klasse Dungeon mit einem Konstruktor und zwei Methoden

5.5.2 Der vorgegebene Konstruktor (default constructor)

Wenn wir in unserer Klasse überhaupt keinen Konstruktor angeben, legt der Compiler automatisch einen an. Diesen Konstruktor nennt die Java Sprachdefinition (JLS) *default constructor*, was wir als *vorgegebener Konstruktor* (selten auch *Vorgabekonstruktor*) eindeutschen wollen.

Schreiben wir nur

```
class Player
{ }
```

dann macht der Compiler daraus immer automatisch:

```
class Player
{
    Player() { }
}
```

Der vorgegebene Konstruktor hat immer die gleiche Sichtbarkeit wie die Klasse. Ist also die Klasse public/private/protected, wird auch der automatisch eingeführte Konstruktor public/private/protected sein. Ist die Klasse paketsichtbar, ist es auch der Konstruktor.

Vorgegebener und expliziter Standard-Konstruktor

Ob ein parameterloser Konstruktor vom Compiler oder Entwickler angelegt wurde, ist ein Implementierungsdetail, das für Nutzer der Klasse irrelevant ist. Daher ist es im Grunde egal, ob wir einen Standard-Konstruktor selbst anlegen oder ob wir uns einen vorgegebenen Konstruktor vom Compiler generieren lassen: Im Bytecode lässt sich das nicht mehr unterscheiden. Selbst eine generierte JavaDoc-API-Dokumentation von einer `public class C1 {}` und `public class C2 { public C2(){} }` wäre strukturell gleich.

<code>CAFE BABE 0000 0032 0010 0700 0201 0002 Ép‰...2.....</code>	<code>CAFE BABE 0000 0032 0010 0700 0201 0002 Ép‰...2.....</code>
<code>4331 0700 0401 0010 6A61 7661 2F6C 616E C1.....java/lan</code>	<code>4332 0700 0401 0010 6A61 7661 2F6C 616E C2.....java/lan</code>
<code>672F 4F62 6A65 6374 0100 063C 696E 6974 g/Object...<init</code>	<code>672F 4F62 6A65 6374 0100 063C 696E 6974 g/Object...<init</code>
<code>3E01 0003 2829 5601 0004 436F 6465 0A00 >...()V...Code..</code>	<code>3E01 0003 2829 5601 0004 436F 6465 0A00 >...()V...Code..</code>
<code>0300 090C 0005 0006 0100 0F4C 696E 654ELineN</code>	<code>0300 090C 0005 0006 0100 0F4C 696E 654ELineN</code>
<code>756D 6265 7254 6162 6C65 0100 124C 6F63 umberTable...Loc</code>	<code>756D 6265 7254 6162 6C65 0100 124C 6F63 umberTable...Loc</code>
<code>616C 5661 7269 6162 6C65 5461 626C 6501 alVariableTable..</code>	<code>616C 5661 7269 6162 6C65 5461 626C 6501 alVariableTable..</code>
<code>0004 7468 6973 0100 044C 4331 3B01 000A ..this..LC1;...</code>	<code>0004 7468 6973 0100 044C 4332 3B01 000A ..this..LC2;...</code>
<code>536F 7572 6365 4669 6C65 0100 0743 312E SourceFile...C1..</code>	<code>536F 7572 6365 4669 6C65 0100 0743 322E SourceFile...C2..</code>
<code>6A61 7661 0021 0001 0003 0000 0000 0001 java.!.....</code>	<code>6A61 7661 0021 0001 0003 0000 0000 0001 java.!.....</code>
<code>0001 0005 0006 0001 0007 0000 002F 0001/...</code>	<code>0001 0005 0006 0001 0007 0000 002F 0001/...</code>
<code>0001 0000 0005 2AB7 0008 B100 0000 0200*..±....</code>	<code>0001 0000 0005 2AB7 0008 B100 0000 0200*..±....</code>
<code>0A00 0000 0600 0100 0000 0100 0B00 0000*....</code>	<code>0A00 0000 0600 0100 0000 0100 0B00 0000*....</code>
<code>0C00 0100 0000 0500 0C00 0D00 0000 0100*....</code>	<code>0C00 0100 0000 0500 0C00 0D00 0000 0100*....</code>
<code>0E00 0000 0200 0F[</code>	<code>0E00 0000 0200 0F[</code>

Abbildung 5.16: Es gibt keinen Unterschied im Bytecode bezüglich der Konstruktoren in den Klassen C1 und C2

In der Begriffswelt der Insel heißt ein parameterloser Konstruktor immer *Standard-Konstruktor*, was natürlich den Unterschied verschwinden lässt, ob der Standard-Konstruktor von Hand angelegt wurde oder als (impliziter) vorgegebener Konstruktor vom Compiler eingeführt wurde. Um das noch klarer zu unterscheiden, können wir diesen Umstand mit *vorgegebener (Standard-)Konstruktor* und *expliziter Standard-Konstruktor* weiter präzisieren.

Auch wenn der Compiler einen vorgegebenen Konstruktor anlegt, ist es oft sinnvoll, einen eigenen Standard-Konstruktor anzugeben, auch wenn der Rumpf leer ist. Ein Grund ist, ihn mit JavaDoc zu dokumentieren, ein anderer Grund ist, die Sichtbarkeit explizit zu wählen, etwa wenn die Klasse `public` ist, aber der Konstruktor nur die Paketsichtbarkeit haben soll.



Begrifflichkeit I

In der *Java Language Specification* gibt es bei den Konstruktoren nur die Trennungen in *no-arg-constructor* (parameterloser Konstruktor) und *default constructor* (vorgegebener Konstruktor), aber den Begriff »standard constructor« gibt es nicht. Viele Autoren übersetzen die englische Bezeichnung »default constructor« (unseren vorgegebenen Konstruktor) einfach nur mit »Standard-Konstruktor«.

Begrifflichkeit II

!

Einige Autoren nennen nur den vom Entwickler explizit geschriebenen parameterlosen Konstruktor »Standard-Konstruktor« und trennen dies sprachlich von dem Konstruktor, den der Compiler generiert hat, den sie weiterhin »Default-Konstruktor« nennen. Beide werden dann zusammengefasst einfach »parameterlose Konstruktoren« genannt.

Wenn also etwa die Frage gestellt wird, ob die Deklaration `class C { }` einen Standard-Konstruktor enthält, ist die Begrifflichkeit des Autors zu prüfen. Wenn der Autor nur den ausprogrammierten parameterlosen Konstruktor »Standard-Konstruktor« genannt hat, so hätte die Klasse C nach seiner Definition *keinen* »Standard-Konstruktor«. Nach der Insel-Definition hätte die Klasse zwar einen vorgegebenen (Standard-)Konstruktor, aber keinen expliziten Standard-Konstruktor.

5.5.3 Parametrisierte und überladene Konstruktoren

Der Standard-Konstruktor hatte keine Parameter, und daher hatten wir ihn auch *parameterlosen Konstruktor* genannt. Ein Konstruktor kann aber wie eine Methode auch eine Parameterliste besitzen: Er heißt dann *parametrisierter Konstruktor* oder *allgemeiner Konstruktor*. Konstruktoren können wie Methoden überladen, also mit unterschiedlichen Parameterlisten deklariert sein. Dies soll auch für den Spieler gelten:

- `Player(String name)`
- `Player(String name, String item)`

Der Player soll sich mit einem Namen und alternativ auch mit einem Gegenstand initialisieren lassen:

Listing 5.30: com/tutego/insel/game/v6/Player.java, Player

```
public class Player
{
    public String name;
    public String item;

    public Player( String name )
    {
        this.name = name;
    }
}
```

```
public Player( String name, String item )
{
    this.name = name;
    this.item = item;
}
```

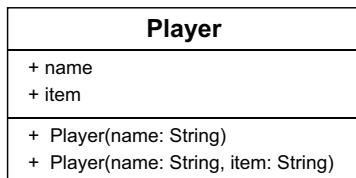


Abbildung 5.17: UML-Diagramm für Player mit zwei Konstruktoren

Die Nutzung kann so aussehen:

Listing 5.31: com/tutego/insel/game/v6/Playground.java, main()

```
Player spuderman = new Player( "Spuderman" );
System.out.println( spuderman.name ); // Spuderman
System.out.println( spuderman.item ); // null

Player holk      = new Player( "Holk", "green color" );
System.out.println( holk.name );      // Holk
System.out.println( holk.item );      // green color
```

Wann der Compiler keinen vorgegebenen Konstruktor einfügt

Wenn es mindestens einen ausprogrammierten Konstruktor gibt, gibt der Compiler keinen eigenen Standard-Konstruktor mehr vor. Wenn wir also nur parametrisierte Konstruktoren haben – wie in unserem obigen Beispiel – führt der Versuch, bei unserer Spieler-Klasse ein Objekt einfach mit dem Standard-Konstruktor über new Player() zu erzeugen, zu einem Übersetzungsfehler, da es eben keinen vom Compiler generierten Standard-Konstruktor gibt:

```
Player p = new Player();           // ☹ The constructor Player() is undefined
```

Dass der Compiler keinen vorgegebenen Konstruktor anlegt, hat seinen guten Grund: Es ließe sich sonst ein Objekt anlegen, ohne dass vielleicht wichtige Variablen initialisiert worden wären. So ist das bei unserem Spieler. Die parametrisierten Konstrukto-

erzwingen, dass beim Erzeugen ein Spielername angegeben werden muss, sodass nach dem Aufbau auf jeden Fall ein Spielername vorhanden ist. Wenn wir es ermöglichen wollen, dass Entwickler neben den parametrisierten Konstruktoren auch einen parameterlosen Standard-Konstruktor nutzen können, müssten wir diesen per Hand hinzufügen.

Wie ein nützlicher Konstruktor aussehen kann

Besitzt ein Objekt eine Reihe von Attributen, so wird ein Konstruktor in der Regel diese Attribute initialisieren wollen. Wenn wir eine Unmenge von Attributen in einer Klasse haben, sollten wir dann auch endlos viele Konstruktoren schreiben? Besitzt eine Klasse Attribute, die durch setXXX()-Methoden gesetzt und durch getXXX()-Methoden gelesen werden, so ist es nicht unbedingt nötig, dass diese Attribute im Konstruktor gesetzt werden. Ein Standard-Konstruktor, der das Objekt in einen Initialzustand setzt, ist angebracht; anschließend können die Zustände mit den Zugriffsmethoden verändert werden. Das sagt auch die JavaBean-Konvention. Praktisch sind sicherlich auch Konstruktoren, die die häufigsten Initialisierungsszenarien abdecken. Das Punkt-Objekt der Klasse `java.awt.Point` lässt sich mit dem Standard-Konstruktor erzeugen, aber auch mit einem parametrisierten, der gleich die Koordinatenwerte entgegennimmt.

Wenn ein Objekt Attribute besitzt, die nicht über setXXX()-Methoden modifiziert werden können, diese Werte aber bei der Objekterzeugung wichtig sind, so bleibt uns nichts anderes übrig, als die Werte im Konstruktor einzufügen (eine setXXX()-Methode, die nur einmalig eine Schreiboperation zulässt, ist nicht wirklich schön). So arbeiten zum Beispiel Wertebobjekte, die einmal im Konstruktor einen Wert bekommen und ihn beibehalten. In der Java-Bibliothek gibt es eine Reihe solcher Klassen, die keinen Standard-Konstruktor besitzen, und nur einige parametrisierte, die Werte erwarten. Die im Konstruktor übergebenen Werte initialisieren das Objekt, und es behält diese Werte sein ganzes Leben lang. Zu den Klassen gehören zum Beispiel `Integer`, `Double`, `Color`, `File` oder `Font`.

5.5.4 Copy-Konstruktor

Ein Konstruktor ist außerordentlich praktisch, wenn er ein typgleiches Objekt über seinen Parameter entgegennimmt und aus diesem Objekt die Startwerte für seinen eigenen Zustand nimmt. Ein solcher Konstruktor heißt *Copy-Konstruktor*.

Dazu ein Beispiel: Die Klasse `Player` bekommt einen Konstruktor, der einen anderen Spieler als Parameter entgegennimmt. Auf diese Weise lässt sich ein schon initialisierter

Spieler als Vorlage für die Attributwerte nutzen. Alle Eigenschaften des existierenden Spielers können so auf den neuen Spieler übertragen werden. Die Implementierung kann so aussehen:

Listing 5.32: com/tutego/insel/game/v7/Player.java, Player

```
public class Player
{
    public String name;
    public String item;

    public Player()
    {
    }

    public Player( Player player )
    {
        name = player.name;
        item = player.item;
    }
}
```

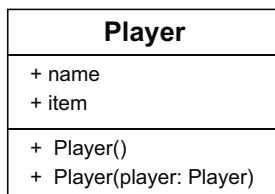


Abbildung 5.18: Klassendiagramm für Player mit Standard- und parametrisierten Konstruktoren

Die statische `main()`-Methode soll jetzt einen neuen Spieler `patric` erzeugen und anschließend wiederum einen neuen Spieler `tryk` mit den Werten von `patric` initialisieren:

Listing 5.33: com/tutego/insel/game/v7/Playground.java, main()

```
Player patric = new Player();
patric.name = "Patric Circle";
patric.item = "Knoten";
```

```
Player tryk = new Player( patric );
System.out.println( tryk.name ); // Patric Circle
System.out.println( tryk.item ); // Knoten
```

Hinweis

5

Wenn die Klasse Player neben dem parametrisierten Konstruktor Player(Player) einen zweiten, Player(Object), deklarieren würde, käme es bei einer Verwendung durch new Player(patric) auf den ersten Blick zu einem Konflikt, denn beide Konstruktoren würden passen. Der Java-Compiler löst das so, dass er immer den spezifischsten Konstruktor aufruft, also Player(Player) und nicht Player(Object). Das gilt auch für new Player(null) – auch hier wird der Konstruktor Player(Player) bemüht. Während diese Frage für den Alltag nicht so bedeutend ist, müssen sich Kandidaten der Java-Zertifizierung *Sun Certified Java Programmer* auf eine solche Frage einstellen. Im Übrigen gilt bei den Methoden das gleiche Prinzip.

5.5.5 Einen anderen Konstruktor der gleichen Klasse mit this() aufrufen

Mitunter werden zwar verschiedene Konstruktoren angeboten, aber nur in einem Konstruktor verbirgt sich die tatsächliche Initialisierung des Objekts. Nehmen wir unser Beispiel mit dem Konstruktor, der einen Spieler als Vorlage über einen Parameter entgegennimmt, aber auch einen anderen Konstruktor, der den Namen und den Gegenstand direkt entgegennimmt:

Listing 5.34: com/tutego/insel/game/v8/Player.java, main()

```
public class Player
{
    public String name;
    public String item;

    public Player( Player player )
    {
        name = player.name;
        item = player.item;
    }
}
```

```
public Player( String name, String item )
{
    this.name = name;
    this.item = item;
}
```

Zu erkennen ist, dass beide Konstruktoren die Objektvariablen initialisieren und letztlich das Gleiche machen. Schlauer ist es, wenn der Konstruktor `Player(Player)` den Konstruktor `Player(String, String)` der eigenen Klasse aufruft. Dann muss nicht gleicher Programmcode für die Initialisierung mehrfach ausprogrammiert werden. Java lässt eine solche Konstruktorverkettung mit dem Schlüsselwort `this` zu:

Listing 5.35: com/tutego/insel/game/v9/Player.java, Player

```
public class Player
{
    public String name;
    public String item;

    public Player()
    {
        this( "", "" );
    }

    public Player( Player player )
    {
        this( player.name, player.item );
    }

    public Player( String name, String item )
    {
        this.name = name;
        this.item = item;
    }
}
```

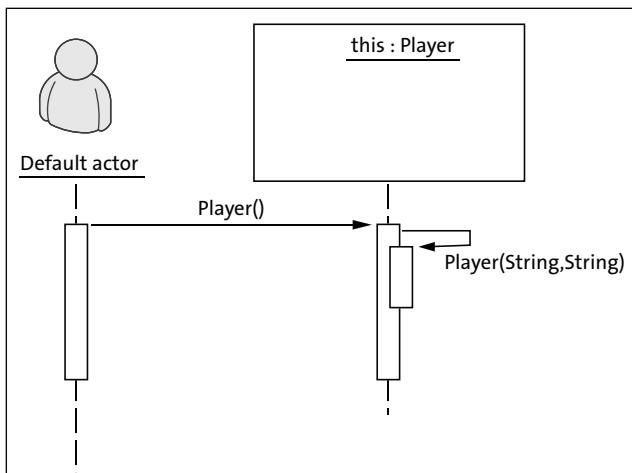


Abbildung 5.19: Das Sequenzdiagramm mit dem Aufruf von zwei Konstruktoren

Der Gewinn gegenüber der vorherigen Lösung ist, dass es nur eine zentrale Stelle gibt, die im Fall von Änderungen angefasst werden müsste. Nehmen wir an, wir hätten zehn Konstruktoren für alle erdenklichen Fälle in genau diesem Stil implementiert. Tritt der unerwünschte Fall ein, dass wir auf einmal in jedem Konstruktor etwas initialisieren müssen, so muss der Programmcode – etwa ein Aufruf der Methode `init()` – in jeden der Konstruktoren eingefügt werden. Dieses Problem umgehen wir einfach, indem wir die Arbeit auf einen speziellen Konstruktor verschieben. Ändert sich nun das Programm in der Weise, dass beim Initialisieren überall zusätzlicher Programmcode ausgeführt werden muss, dann ändern wir eine Zeile in dem konkreten, von allen benutzten Konstruktor. Damit fällt für uns wenig Änderungsarbeit an – unter softwaretechnischen Gesichtspunkten ein großer Vorteil. Überall in den Java-Bibliotheken lässt sich diese Technik wiedererkennen.

Hinweis

Das Schlüsselwort `this` ist in Java mit zwei Funktionen belegt: Zum einen »zeigt« es als Referenz auf das aktuelle Objekt, und zum anderen formt es einen Aufruf zu einem anderen Konstruktor der gleichen Klasse.



Einschränkungen von `this()` *

Beim Aufruf eines anderen Konstruktors mittels `this()` gibt es zwei wichtige Beschränkungen:

- Der Aufruf von `this()` muss die erste Anweisung des Konstruktors sein.
- Vor dem Aufruf von `this()` im Konstruktor können keine Objekteigenschaften ange- sprochen werden. Das heißt, es darf weder eine Objektvariable als Argument an `this()` übergeben werden noch darf eine andere Objektmethode der Klasse aufgerufen werden, die etwa das Argument berechnen möchte. Erlaubt ist nur der Zugriff auf statische Variablen (etwa finale Variablen, die Konstanten sind) oder der Aufruf von statischen Methoden.

Die erste Einschränkung besagt, dass das Erzeugen eines Objekts immer das Erste ist, was ein Konstruktor leisten muss. Nichts darf vor der Initialisierung ausgeführt werden. Die zweite Einschränkung hat damit zu tun, dass die Objektvariablen erst *nach* dem Aufruf von `this()` initialisiert werden, sodass ein Zugriff unsinnig wäre – die Werte wären im Allgemeinen null:

Listing 5.36: Stereo.java

```
public class Stereo
{
    static      final int STANDARD = 1000;
    /*non-static*/ final int standard = 1000;
    public int watt;

    public Stereo()
    {
        // this( standard );      // ☹ Führt auskommentiert zum Compilerfehler:
        // ^ Cannot refer to an instance field standard while
        // explicitly invoking a constructor

        this( STANDARD );
    }

    public Stereo( int watt )
    {
        this.watt = watt;
    }
}
```

Da Objektvariablen bis zu einem bestimmten Punkt noch nicht initialisiert sind (was der nächste Abschnitt erklärt), lässt uns der Compiler nicht darauf zugreifen – nur statische Variablen sind als Übergabeparameter erlaubt. Daher ist der Aufruf `this(standard)` nicht gültig, da `standard` eine Objektvariable ist; `this(STANDARD)` ist jedoch in Ordnung, weil `STANDARD` eine statische Variable ist.

5.5.6 Ihr fehlt uns nicht – der Garbage-Collector

Glücklicherweise werden wir beim Programmieren von der lästigen Aufgabe befreit, Speicher von Objekten freizugeben. Wird ein Objekt nicht mehr referenziert, findet der Garbage-Collector⁷ dieses Objekt und kümmert sich um alles Weitere – der Entwicklungsprozess wird dadurch natürlich vereinfacht. Der Einsatz eines GC verhindert zwei große Probleme:

- Ein Objekt kann gelöscht werden, aber die Referenz existiert noch (engl. *dangling pointer*).
- Kein Zeiger verweist auf ein bestimmtes Objekt, dieses existiert aber noch im Speicher (engl. *memory leak*).

Hinweis

Konstruktoren sind besondere Anweisungsblöcke, die die Laufzeitumgebung immer im Zuge der Objekterzeugung aufruft. Sprachen wie C++ kennen auch das Konzept eines Dekonstruktors, also eines besonderen Anweisungsblocks, der immer dann aufgerufen wird, wenn die Laufzeitumgebung erkennt, dass das Objekt nicht mehr benötigt wird. Allgemeine Dekonstruktoren kennt Java nicht. Es gibt jedoch mit der `finalize()`-Methode (sie wird in Abschnitt 8.3.7, »Aufräumen mit `finalize()`«, vorgestellt) eine Möglichkeit, die an einen Dekonstruktor erinnert, allerdings mit einigen Einschränkungen.

Prinzipielle Arbeitsweise des Müllauftsammlers

Der Garbage-Collector (GC) erscheint hier als ominöses Ding, das die Objekte clever verwaltet. Doch was ist der GC? Implementiert wird er als unabhängiger Thread mit niedriger Priorität. Er verwaltet die Wurzelobjekte, von denen aus das gesamte Geflecht der

⁷ Eine lange Tradition hat der Garbage-Collector unter LISP und unter Smalltalk, aber auch Visual Basic benutzt einen GC, und selbst das C64-BASIC nutzte Garbage-Collection für nicht mehr benötigte Zeichenketten.

lebendigen Objekte (der sogenannte Objektgraph) erreicht werden kann. Dazu gehören die Wurzel des Thread-Gruppen-Baums und die lokalen Variablen aller aktiven Methodenaufrufe (Stack aller Threads). In regelmäßigen Abständen markiert der GC nicht benötigte Objekte und entfernt sie.

Dank der *HotSpot*-Technologie geschieht das Anlegen von Objekten unter der Java VM von Oracle sehr schnell. HotSpot verwendet einen generationenorientierten GC, der den Umstand ausnutzt, dass zwei Gruppen von Objekten mit deutlich unterschiedlicher Lebensdauer existieren. Die meisten Objekte sterben sehr jung, die wenigen überlebenden Objekte werden hingegen sehr alt. Die Strategie dabei ist, dass Objekte im »Kinderergarten« erzeugt werden, der sehr oft nach toten Objekten durchsucht wird und in der Größe beschränkt ist. Überlebende Objekte kommen nach einiger Zeit aus dem Kinderergarten in eine andere Generation, die nur selten vom GC durchsucht wird. Damit folgt der GC der Philosophie von Auffenberg, der meinte: »Verbesserungen müssen zeitig glücken; im Sturm kann man nicht mehr die Segel flicken.« Das heißt, der GC arbeitet ununterbrochen und räumt auf. Er beginnt nicht erst mit der Arbeit, wenn es zu spät und der Speicher schon voll ist.

Die manuelle Nullung und Speicherlecks

Im folgenden Szenario wird der GC das nicht mehr benötigte Objekt hinter der Referenzvariablen `ref` entfernen, wenn die Laufzeitumgebung den inneren Block verlässt:

```
{
{
    StringBuffer ref = new StringBuffer();
}
// StringBuffer ref ist frei für den GC
}
```

In fremden Programmen sind mitunter Anweisungen wie die folgende zu lesen:

```
ref = null;
```

Oftmals sind sie unnötig, denn wie im Fall unseres Blocks weiß der GC, wann der letzte Verweis vom Objekt genommen wurde. Anders sieht das aus, wenn die Lebensdauer der Variablen größer ist, etwa bei einer Objekt- oder sogar bei einer statischen Variablen, oder wenn sie in einem Feld referenziert wird. Wenn dann das referenzierte Objekt nicht mehr benötigt wird, sollte die Variable (oder der Feldeintrag) mit `null` belegt werden, da andernfalls der GC das Objekt aufgrund der starken Referenzierung nicht weg räumen würde. Zwar findet der GC jedes nicht mehr referenzierte Objekt, aber die Fähig

keit zur Divination⁸, Speicherlecks durch unbenutzte, aber referenzierte Objekte aufzuspüren, hat er nicht.

5.5.7 Private Konstruktoren, Utility-Klassen, Singleton, Fabriken

Ein Konstruktor kann privat sein, was verhindert, dass von außen ein Exemplar dieser Klasse gebildet werden kann. Was auf den ersten Blick ziemlich beschränkt erscheint, erweist sich als ziemlich clever, wenn damit die Exemplarbildung bewusst verhindert werden soll. Sinnvoll ist das etwa bei den sogenannten *Utility-Klassen*. Das sind Klassen, die nur statische Methoden besitzen, also Hilfsklassen sind. Beispiele für diese Hilfsklassen gibt es zur Genüge, zum Beispiel Math. Warum sollte es hier Exemplare geben? Für den Aufruf von `max()` ist das nicht nötig. Also wird die Bildung von Objekten erfolgreich mit einem privaten Konstruktor unterbunden.

Wenn ein Konstruktor privat ist, bedeutet das noch lange nicht, dass keine Exemplare mehr erzeugt werden können. Ein privater Konstruktor besagt nur, dass er von außen nicht sichtbar ist – aber die Klasse selbst kann ihn ebenso wie private Methoden »sehen« und zur Objekterzeugung nutzen. Objektmethoden kommen dafür nicht in Frage, da ähnlich wie beim Henne-Ei-Problem ja vorher ein Objekt nötig wäre. Es bleiben somit die statischen Methoden als Erzeuger.

Singleton

Ein *Singleton* stellt sicher, dass es von einer Klasse nur ein Exemplar gibt.⁹ Nützlich ist das für Dinge, die es nur genau einmal in einer Applikation geben soll, etwa einen Logger für Protokollierungen:

Listing 5.37: Logger.java

```
public final class Logger
{
    private static Logger logger;
    private Logger()
    {
    }
}
```

⁸ Wahrsagen

⁹ Nicht in der gesamten JVM, das ist nicht möglich. Zu diesem Problem kommen wir noch in Abschnitt 11.5, »Klassenlader (Class Loader)«.

```

public static synchronized Logger getInstance()
{
    if ( logger == null )
        logger = new Logger();
    return logger;
}

public void log( String s )
{
    System.out.println( s );
}
}

```

Interessant sind einmal der private Konstruktor und zum anderen die statische Anfrage-Methode `getInstance()`. Gibt es noch kein Exemplar des Loggers, erzeugt `getInstance()` eines und weist es der Klassenvariablen zu. `synchronized` schützt bei parallelen Zugriffen, sodass nur ein Thread die Methode betreten kann und ein potenziell anderer Thread so lange warten muss, bis der erste Thread die Methode wieder verlassen hat:

Listing 5.38: LoggerUser.java

```

public class LoggerUser
{
    public static void main( String[] args )
    {
        Logger.getInstance().log( "Log mich!" );
    }
}

```

Fabrikmethoden

Eine *Fabrikmethode* geht noch einen Schritt weiter als ein Singleton. Sie erzeugt nicht exakt ein Exemplar, sondern unter Umständen auch mehrere. Die grundlegende Idee jedoch ist, dass der Anwender nicht über einen Konstruktor ein Exemplar erzeugt, sondern im Allgemeinen über eine statische Methode. Dies hat den Vorteil, dass die statische Fabrikmethode

- alte Objekte aus einem Cache wiedergeben kann,
- den Erzeugungsprozess auf Unterklassen verschieben kann und
- null zurückgeben darf.

Ein Konstruktor erzeugt immer ein Exemplar der eigenen Klasse. Eine Rückgabe wie null kann ein Konstruktor nicht liefern, denn bei new wird immer ein neues Objekt gebaut. Fehler könnten nur über eine Exception angezeigt werden.

In der Java-Bibliothek gibt es eine Unmenge an Beispielen für Fabrikmethoden. Durch eine Namenskonvention sind sie leicht zu erkennen: Meistens heißen sie getInstance(). Eine Suche in der API-Dokumentation fördert gleich 90 solcher Methoden zutage. Viele sind parametrisiert, um genau anzugeben, was die Fabrik für Objekte erzeugen soll. Nehmen wir zum Beispiel die statischen Fabrikmethoden vom java.util.Calendar:

- Calendar.getInstance()
- Calendar.getInstance(java.util.Locale)
- Calendar.getInstance(java.util.TimeZone)

Die nicht parametrisierte Methode gibt ein Standard-Calendar-Objekt zurück. Calendar ist aber selbst eine abstrakte Basisklasse. Innerhalb der getInstance()-Methode befindet sich Quellcode wie der folgende:

```
static Calendar getInstance()
{
    return new GregorianCalendar();
}
```

Im Rumpf der Erzeugermethode getInstance() wird bewusst die Unterklasse GregorianCalendar ausgewählt, die Calendar erweitert. Das ist möglich, da durch Vererbung eine Ist-eine-Art-von-Beziehung gilt und GregorianCalendar ein Calendar ist. Der Aufrufer von getInstance() bekommt das nicht mit, und er empfängt wie gewünscht ein Calendar-Objekt. Mit dieser Möglichkeit kann getInstance() testen, in welchem Land die JVM läuft, und abhängig davon die passende Calendar-Implementierung auswählen.

5.6 Klassen- und Objektinitialisierung *

Eine wichtige Eigenschaft guter Programmiersprachen ist ihre Fähigkeit, keine uninitialisierten Zustände zu erzeugen. Bei lokalen Variablen achtet der Compiler auf die Belegung, also darauf, ob vor dem ersten Lesezugriff schon ein Wert zugewiesen ist. Bei

Objektvariablen und Klassenvariablen haben wir bisher festgestellt, dass die Variablen automatisch mit 0, null oder false oder mit einem eigenen Wert belegt werden. Wir wollen jetzt sehen, wie dies genau funktioniert.

5.6.1 Initialisierung von Objektvariablen

Wenn der Compiler eine Klasse mit Objekt- oder Klassenvariablen sieht, dann müssen diese Variablen an irgendeiner Stelle initialisiert werden. Werden sie einfach deklariert und nicht mit einem Wert initialisiert, so regelt die virtuelle Maschine die Vorbelegung. Spannender ist der Fall, wenn den Variablen explizit ein Wert zugewiesen wird (der auch 0 sein kann). Dann erzeugt der Compiler automatisch einige zusätzliche Zeilen.

Betrachten wir dies zuerst für eine Objektvariable:

Listing 5.39: InitObjectVariable.java

```
class InitObjectVariable
{
    int j = 1;

    InitObjectVariable()
    {
    }

    InitObjectVariable( int j )
    {
        this.j = j;
    }

    InitObjectVariable( int x, int y )
    {
    }
}
```

Die Variable j wird mit 1 belegt. Es ist wichtig, zu wissen, an welcher Stelle Variablen ihre Werte bekommen. So erstaunlich es klingt, aber die Zuweisung findet im Konstruktor statt. Das heißt, der Compiler wandelt das Programm bei der Übersetzung eigenmächtig wie folgt um:

```

class InitObjectVariable
{
    int j;

    InitObjectVariable()
    {
        j = 1;
    }

    InitObjectVariable( int j )
    {
        this.j = 1;
        this.j = j;
    }

    InitObjectVariable( int x, int y )
    {
        j = 1;
    }
}

```

Wir erkennen, dass die Variable wirklich nur beim Aufruf des Konstruktors initialisiert wird. Die Zuweisung steht dabei in der ersten Zeile. Dies kann sich als Falle erweisen, denn problematisch ist etwa die Reihenfolge der Belegung.

Manuelle Nullung

Genau genommen initialisiert die Laufzeitumgebung jede Objekt- und Klassenvariable zunächst mit 0, null oder false und später mit einem Wert. Daher ist die Nullung von Hand nicht nötig:

```

class NeedlessInitNull
{
    int    i = 0;           // unnötig
    String s = null;       // unnötig
}

```

Der Compiler würde nur zusätzlich in jeden Konstruktor die Initialisierung `i = 0`, `s = null` einsetzen.¹⁰ Aus diesem Grund ist auch Folgendes nicht meisterhaft:

```
class NeedlessInitNull
{
    int i = 0;
    NeedlessInitNull( int i ) { this.i = i; }
}
```

Die Belegung für `i` wird sowieso überschrieben.

5.6.2 Statische Blöcke als Klasseninitialisierer

Eine Art Konstruktor für das Klassenobjekt selbst (und nicht für das Exemplar der Klasse) ist ein `static`-Block, der einmal oder mehrmals in eine Klasse gesetzt werden kann. Jeder Block wird genau dann ausgeführt, wenn die Klasse vom Klassenlader in die virtuelle Maschine geladen wird.¹¹ Der Block heißt *Klasseninitialisierer* oder *statischer Initialisierungsblock*:

Listing 5.40: StaticBlock.java

```
class StaticBlock
{
    static
    {
        System.out.println( "Eins" );
    }

    public static void main( String[] args )
    {
        System.out.println( "Jetzt geht's los." );
    }
}
```

¹⁰ Wir wollen hier den Fall, dass der Konstruktor der Oberklasse `i` einen Wert ungleich 0 setzt, nicht betrachten.

¹¹ In der Regel geschieht dies nur einmal während eines Programmlaufs. Unter gewissen Umständen – es gibt einen eigenen Klassenlader für die Klasse – kann jedoch eine Klasse auch aus dem Speicher entfernt und dann mit einem anderen Klassenlader wieder neu geladen werden. Dann werden die `static`-Blöcke neu ausgeführt.

```
static
{
    System.out.println( "Zwei" );
}
}
```

Lädt der Klassenlader die Klasse StaticBlock, so führt er zuerst den ersten Block mit der Ausgabe »Eins« aus und dann den Block mit der Ausgabe »Zwei«. Da die Klasse StaticBlock auch das `main()` besitzt, führt die virtuelle Maschine anschließend die Startmethode aus.

Java-Programme ohne `main()`

Lädt der Klassenlader eine Klasse, so führt er als Allererstes die statischen Blöcke aus. Mit dieser Eigenschaft lassen sich Programme ohne statische `main()`-Methode schreiben. In den statischen Block wird einfach das Hauptprogramm geschrieben. Da die virtuelle Maschine aber immer noch nach dem `main()` sucht, müssen wir die Laufzeitumgebung schon vorher beenden. Dies geschieht dadurch, dass mit `System.exit()` die Bearbeitung abgebrochen wird:

Listing 5.41: StaticNowMain.java

```
class StaticNowMain
{
    static
    {
        System.out.println( "Jetzt bin ich das Hauptprogramm" );
        System.exit( 0 );
    }
}
```

Nicht jede Laufzeitumgebung nimmt das jedoch ohne Murren hin. Mit diesem Vorgehen ist der Nachteil verbunden, dass bei Ausnahmen im versteckten Hauptprogramm manche virtuellen Maschinen unsinnige Fehler melden – etwa den, dass die Klasse `StaticNowMain` nicht gefunden wurde, oder auch einen `ExceptionInInitializerError` meldet, der anstelle einer vernünftigen Exception kommt.

5.6.3 Initialisierung von Klassenvariablen

Abschließend bleibt die Frage, wo Klassenvariablen initialisiert werden. Im Konstruktor ergibt dies keinen Sinn, da für Klassenvariablen keine Objekte angelegt werden müssen. Dafür gibt es den `static{}`-Block. Dieser wird immer dann ausgeführt, wenn der Klassenlader eine Klasse in die Laufzeitumgebung geladen hat. Für eine statische Initialisierung wird also wieder der Compiler etwas einfügen:

Was wir schreiben	Was der Compiler generiert
<pre>public class InitStaticVariable { static int staticInt = 2; }</pre>	<pre>public class InitStaticVariable { static int staticInt; static { staticInt = 2; } }</pre>

5.6.4 Eincompilierte Belegungen der Klassenvariablen

Finale Klassenvariablen können in der Entwicklung mit einer größeren Anzahl von Klassen zu einem Problem werden. Das liegt an der Eigenschaft der finalen Werte, dass sie sich nicht ändern können und sich daher sicher an der Stelle einsetzen lassen, wo sie gebraucht werden. Ein Beispiel:

```
public class Finance
{
    public static final int TAX = 19;
}
```

Greift eine andere Klasse auf die Variable `TAX` zu, ist das im Quellcode nicht als direkter Variablenzugriff `Finance.TAX` kodiert, sondern der Compiler hat das Literal 19 direkt an jeder Aufrufstelle eingesetzt. Dies ist eine Optimierung des Compilers, die er laut Java-Spezifikation vornehmen kann.

Das ist zwar nett, bringt aber gewaltige Probleme mit sich, etwa dann, wenn sich die Konstante einmal ändert. Dann muss nämlich auch jede Klasse übersetzt werden, die Bezug auf die Konstante hatte. Werden die abhängigen Klassen nicht neu übersetzt, ist in ihnen immer noch der alte Wert eincompiliert.

Die Lösung ist, die bezugnehmenden Klassen neu zu übersetzen und sich am besten anzugewöhnen, bei einer Änderung einer Konstante gleich alles neu zu compilieren. Ein anderer Weg transformiert die finale Variable in eine später initialisierte Form:

```
public class Finance
{
    public static final int TAX = Integer.valueOf( 19 );
}
```

Die Initialisierung findet im statischen Initialisierer statt, und die Konstante mit dem Literal 19 ist zunächst einmal verschwunden. Der Compiler wird also beim Zugriff auf Finance.TAX keine Konstante 19 vorfinden und daher das Literal an den Aufrufstellen nicht einbauen können. In der Klassendatei wird der Bezug Finance.TAX vorhanden sein, und eine Änderung der Konstanten erzwingt keine neue Übersetzung der Klassen.

5.6.5 Exemplarinitialisierer (Instanzinitialisierer)

Neben den Konstruktoren haben die Sprachschöpfer eine weitere Möglichkeit vorgesehen, um Objekte zu initialisieren. Diese Möglichkeit wird insbesondere bei anonymen inneren Klassen wichtig, also bei Klassen, die sich in einer anderen Klasse befinden.

Ein Exemplarinitialisierer ist ein Konstruktor ohne Namen. Er besteht in einer Klassendeklaration nur aus einem Paar geschweifter Klammern und gleicht einem statischen Initialisierungsblock ohne das Schlüsselwort `static`:

Listing 5.42: JavaInitializers.java

```
public class JavaInitializers
{
    static
    {
        System.out.println( "Statischer Initialisierer" );
    }

    {
        System.out.println( "Exemplarinitialisierer" );
    }

    JavaInitializers()
}
```

```
{  
    System.out.println( "Konstruktor" );  
}  
  
public static void main( String[] args )  
{  
    new JavaInitializers();  
    new JavaInitializers();  
}  
}
```

Die Ausgabe ist:

```
Statischer Initialisierer  
Exemplarinitialisierer  
Konstruktor  
Exemplarinitialisierer  
Konstruktor
```

Der statische Initialisierer wird nur einmal abgearbeitet: genau dann, wenn die Klasse geladen wird. Konstruktor und Exemplarinitialisierer werden pro Aufbau eines Exemplars abgearbeitet. Der Programmcode vom Exemplarinitialisierer wird dabei vor dem eigentlichen Programmcode im Konstruktor abgearbeitet.

Mit Exemplarinitialisierern Konstruktoren vereinfachen

Die Exemplarinitialisierer können gut dazu verwendet werden, Initialisierungsarbeit bei der Objekterzeugung auszuführen. In den Blöcken lässt sich Programmcode setzen, der sonst in jeden Konstruktor kopiert oder andernfalls in einer gesonderten Methode zentralisiert werden müsste. Mit dem Exemplarinitialisierer lässt sich der Programmcode vereinfachen, denn der gemeinsame Teil kann in diesen Block gelegt werden, und wir haben eine Quellcode-Duplizierung im Quellcode vermieden. Allerdings hat die Technik gegenüber einer langweiligen Initialisierungsmethode auch Nachteile:

- Zwar ist im Quellcode die Duplizierung nicht mehr vorhanden, aber in der Klassendatei steht sie wieder. Das liegt daran, dass der Compiler alle Anweisungen des Exemplarinitialisierers in jeden Konstruktor kopiert.
- Exemplarinitialisierer können schnell übersehen werden. Ein Blick auf den Konstruktor verrät uns dann nicht mehr, was er alles macht, da verstreute Exemplarinitialisie-

rer Initialisierungen ändern oder hinzufügen können. Die Initialisierung trägt damit nicht zur Übersichtlichkeit bei.

- Ein weiteres Manko ist, dass die Initialisierung nur bei neuen Objekten, also mit `new()`, durchgeführt wird. Wenn Objekte wiederverwendet werden sollen, ist eine private Methode wie `initialize()`, die das Objekt wie frisch erzeugt initialisiert, gar nicht so schlecht. Eine Methode lässt sich immer aufrufen, und damit sind die Objektzustände wie neu.
- Die API-Dokumentation führt Exemplarinitialisierer nicht auf; die Konstruktoren müssen also die Aufgabe erklären.

Mehrere Exemplarinitialisierer

In einer Klasse können mehrere Exemplarinitialisierer auftauchen. Sie werden der Reihe nach durchlaufen, und zwar vor dem eigentlichen Konstruktor. Der Grund liegt in der Realisierung der Umsetzung: Der Programmcode der Exemplarinitialisierer wird an den Anfang aller Konstruktoren gesetzt. Objektvariablen wurden schon initialisiert. Ein Programmcode wie der folgende:

Listing 5.43: WhoIsAustin.java

```
class WhoIsAustin
{
    String austinPowers = "Mike Myers";

    {
        System.out.println( "1 " + austinPowers );
    }

    WhoIsAustin()
    {
        System.out.println( "2 " + austinPowers );
    }
}
```

wird vom Compiler also umgebaut zu:

```
class WhoIsAustin
{
    String austinPowers;
```

```

WhoIsAustin()
{
    austinPowers = "Mike Myers";
    System.out.println( "1 " + austinPowers );
    System.out.println( "2 " + austinPowers );
}
}

```

Wichtig ist, abschließend zu sagen, dass vor dem Zugriff auf eine Objektvariable im Exemplarinitialisierer diese Variable vorher im Programm deklariert sein muss und somit dem Compiler bekannt sein muss. Korrekt ist:

```

class WhoIsDrEvil
{
    String drEvil = "Mike Myers";

    {
        System.out.println( drEvil );
    }
}

```

Während Folgendes zu einem Fehler führt:

```

class WhoIsDrEvil
{
    {
        System.out.println( drEvil );      // ☠ Compilerfehler
    }

    String drEvil = "Mike Myers";
}

```

Das ist eher ungewöhnlich, denn würden wir die print-Anweisung in einen Konstruktor setzen, wäre das erlaubt.

Hinweis

Exemplarinitialisierer ersetzen keine Konstruktoren! Sie sind selten im Einsatz und eher für innere Klassen gedacht, ein Konzept, das später in Kapitel 7, »Äußere.innere Klassen«, vorgestellt wird.

5.6.6 Finale Werte im Konstruktor und in statischen Blöcken setzen

Wie die Beispiele im vorangegangenen Abschnitt zeigen, werden Objektvariablen erst im Konstruktor gesetzt und statische Variablen in einem static-Block. Diese Tatsache müssen wir jetzt mit finalen Variablen zusammenbringen, was uns dahinführt, dass auch sie in Konstruktoren beziehungsweise in Initialisierungsblöcken zugewiesen werden. Im Unterschied zu nicht-finalen Variablen müssen finale Variablen auf jeden Fall gesetzt werden, und nur genau ein Schreibzugriff ist möglich.

Finale Werte aus dem Konstruktor belegen

Eine finale Variable darf nur einmal belegt werden. Das bedeutet nicht zwingend, dass sie am Deklarationsort mit einem Wert belegt werden muss, sondern es ist möglich, das auch später vorzunehmen. Der Konstruktor darf zum Beispiel finale Objektvariablen beschreiben. Das Paar aus finaler Variable und initialisierendem Konstruktor ist ein häufig genutztes Idiom, wenn Variablenwerte später nicht mehr geändert werden sollen. So ist im Folgenden die Variable `pattern` final, da sie nur einmalig über den Konstruktor gesetzt und anschließend nur noch gelesen wird:

Listing 5.44: Pattern.java

```
public class Pattern
{
    private final String pattern;

    public Pattern( String pattern )
    {
        this.pattern = pattern;
    }

    public String getPattern()
    {
```

```

        return pattern;
    }
}

```



Java-Stil

Immer dann, wenn sich bis auf die direkte Initialisierung vor Ort oder im Konstruktor die Belegung nicht mehr ändert, sollten Entwickler finale Variablen verwenden.

Konstante mit Dateiinhalt initialisieren

Mit diesem Vorgehen lassen sich auch »variable« Konstanten angeben, deren Belegung sich erst zur Laufzeit ergibt. Im nächsten Beispiel soll eine Datei eine Konstante enthalten, die Hubble-Konstante¹²:

Listing 5.45: hubble-constant.txt

77

Die Hubble-Konstante bestimmt die Expansionsgeschwindigkeit des Universums und ist eine zentrale Größe in der Kosmologie. Dummerweise ist die genaue Bestimmung schwer und der Name *Konstante* eigentlich unpassend. Damit eine Änderung des Werts nicht zur Neuübersetzung des Java-Programms führen muss, legen wir den Wert in eine Datei und belegen gerade nicht direkt die finale statische Konstantenvariable. Die Klasse liest in einem static-Block den Wert aus der Datei und belegt die finale statische Konstante:

Listing 5.46: LateConstant.java

```

public class LateConstant
{
    public static final int      HUBBLE;
    public      final String    ISBN;

    static
    {
        HUBBLE = new java.util.Scanner(
            LateConstant.class.getResourceAsStream("hubble-constant.txt")).nextInt();
    }
}

```

¹² <http://de.wikipedia.org/wiki/Hubble-Konstante>

```
}

public LateConstant()
{
    ISBN = "3572100100";
}

public static void main( String[] args )
{
    System.out.println( HUBBLE );                      // 77
    System.out.println( new LateConstant().ISBN );      // 3572100100
}
}
```

5

Im Beispiel arbeiten mehrere Klassen zusammen, um eine Zahl einzulesen. Am Anfang steht das `Class`-Objekt, das Zugriff auf den Klassenlader liefert, der einen Zugang zur Datei ermöglicht. `LateConstant.class` ist die Schreibweise, um das `Class`-Objekt unserer eigenen Klasse zu beziehen. Die Methode `getResourceAsStream()` ist eine Objektmethode des `Class`-Objekts und gibt einen Datenstrom zum Dateiinhalt, den die Klasse `Scanner` als Eingabequelle zum Lesen nutzt. Die Objektmethode `nextInt()` liest anschließend eine Ganzzahl aus der Datei aus.

5.7 Assoziationen zwischen Objekten

Eine wichtige Eigenschaft objektorientierter Systeme ist der Austausch von Nachrichten untereinander. Dazu »kennt« ein Objekt andere Objekte und kann Anforderungen weitergeben. Diese Verbindung nennt sich *Assoziation* und ist das wichtigste Werkzeug bei der Bildung von Objektverbänden.

Assoziationstypen

Bei Assoziationen ist zu unterscheiden, ob nur eine Seite die andere kennt oder ob eine Navigation in beiden Richtungen möglich ist:

- Eine *unidirektionale Beziehung* geht nur in eine Richtung (ein Fan kennt seine Band, aber nicht umgekehrt).

- Eine *bidirektionale Beziehung* geht in beide Richtungen (Raum kennt Spieler und Spieler kennt Raum). Eine bidirektionale Beziehung ist natürlich ein großer Vorteil, da die Anwendung die Assoziation in beliebiger Richtung ablaufen kann.

Daneben gibt es bei Beziehungen die *Multiplizität*, auch *Kardinalität* genannt. Sie sagt aus, mit wie vielen Objekten eine Seite eine Beziehung haben kann. Übliche Beziehungen sind 1:1 und 1:n.

5.7.1 Unidirektionale 1:1-Beziehung

Damit ein Spieler sich in einem Raum befinden kann, lässt sich in Player eine Referenzvariable vom Typ Room anlegen. In Java sähe das in etwa so aus:

Listing 5.47: com/tutego/insel/game/va/Player.java, Player

```
public class Player
{
    public Room room;
}
```

Listing 5.48: com/tutego/insel/game/va/Room.java, Room

```
public class Room
{
}
```

Zur Laufzeit müssen natürlich noch die Verweise gesetzt werden:

Listing 5.49: com/tutego/insel/game/va/Playground.java, main()

```
Player buster = new Player();
Room tower = new Room();
buster.room = tower;           // Buster kommt in den Tower
```

Assoziationen in der UML

Die UML stellt Assoziationen durch eine Linie zwischen den beteiligten Klassen dar. Hat eine Assoziation eine Richtung, zeigt ein Pfeil am Ende der Assoziation diese an. Wenn es keine Pfeile gibt, heißt das nur, dass die Richtung noch nicht genauer spezifiziert ist, und nicht automatisch, dass die Beziehung bidirektional ist.



Abbildung 5.20: Gerichtete Assoziation im UML-Diagramm

Die Multiplizität wird angeben als »untere Grenze..obere Grenze«, etwa 1..4. Außerdem lässt sich in UML über eine Rolle angeben, welche Aufgabe die Beziehung für eine Seite hat. Die Rollen sind wichtig für *reflexive Assoziationen* (auch *zirkuläre* oder *rekursive Assoziationen* genannt), wenn ein Typ auf sich selbst zeigt. Ein beliebtes Beispiel ist der Typ Person mit den Rollen Chef und Mitarbeiter.

5.7.2 Bidirektionale 1:1-Beziehungen

Diese gerichteten Assoziationen sind in Java sehr einfach umzusetzen, wie wir im Beispiel gesehen haben. Beidseitige Assoziationen erscheinen auf den ersten Blick auch einfach, da nur die Gegenseite um eine Verweisvariable erweitert werden muss. Beginnen wir mit dem Szenario, dass der Spieler seinen Raum und der Raum seinen Spieler kennen soll:

Listing 5.50: com/tutego/insel/game/vb/Player.java, Player

```

public class Player {
    public Room room;
}
  
```

Listing 5.51: com/tutego/insel/game/vb/Room.java, Room

```

public class Room {
    public Player player;
}
  
```



Abbildung 5.21: Bei bidirektionalen Beziehungen gibt es im UML-Diagramm zwei Pfeile.

Verbinden wir das:

Listing 5.52: com/tutego/insel/game/vb/Playground.java, main()

```
Player buster = new Player();
Room tower = new Room();
buster.room = tower;
tower.player = buster;
```

So einfach ist es aber nicht! Bidirektionale Beziehungen erfordern etwas mehr Programmieraufwand, da sichergestellt sein muss, dass beide Seiten eine gültige Referenz besitzen. Denn wird die Assoziation auf einer Seite aufgekündigt, etwa durch Setzen der Referenz auf `null`, muss auch die andere Seite die Referenz lösen:

```
buster.room = null;           // Spieler will nicht mehr im Raum sein
```

Auch kann es passieren, dass zwei Räume angeben, einen Spieler zu besitzen, doch der Spieler kennt von der Modellierung her nur genau einen Raum:

Listing 5.53: com/tutego/insel/game/vb/InvalidPlayground.java, main()

```
Player buster = new Player();
Room tower = new Room();
buster.room = tower;
tower.player = buster;
Room toilet = new Room();
toilet.player = buster;
System.out.println( buster );           // com.tutego.insel.game.vb.Player@aaaaaa
System.out.println( tower );           // com.tutego.insel.game.vb.Room@444444
System.out.println( toilet );          // com.tutego.insel.game.vb.Room@999999
System.out.println( buster.room );      // com.tutego.insel.game.vb.Room@444444
System.out.println( tower.player );     // com.tutego.insel.game.vb.Player@aaaaaa
System.out.println( toilet.player );    // com.tutego.insel.game.vb.Player@aaaaaa
```

An der Ausgabe ist abzulesen, dass sich Buster im Tower befindet, aber auch die Toilette sagt, dass Buster dort ist (die Kennungen hinter @ sind für das Buch durch gut unterscheidbare Zeichenketten ersetzt worden. Sie sind bei jedem Aufruf anders).

Die Wurzel des Übels liegt in den Variablen. Variablen können keine Konsistenzbedingungen aufrechterhalten, Methoden können wie in einer Transaktion aber mehrere

Operationen durchführen und von einem korrekten Zustand in den nächsten überführen. Daher erfolgt diese Kontrolle am besten mit Zugriffsmethoden, etwa wie `setRoom()` und `setPlayer()`.

5.7.3 Unidirektionale 1:n-Beziehung

Immer dann, wenn ein Objekt mehrere andere Objekte referenzieren muss, reicht eine einfache Referenzvariable vom Typ der anderen Seite nicht mehr aus. Dann sind Datenstrukturen gefragt, die mehrere Referenzen aufnehmen können, etwa dann, wenn sich in einem Raum mehrere Spieler befinden können oder wenn ein Spieler mehrere Gegenstände mit sich trägt. Wir müssen auf der 1-Seite eine Datenstruktur verwenden, die entweder eine feste oder eine dynamische Anzahl anderer Objekte aufnimmt. Eine Handy-Tastatur hat beispielsweise nur eine feste Anzahl von Tasten und ein Tisch nur eine feste Anzahl von Beinen. Bei Sammlungen dieser Art ist ein Array gut geeignet. Bei anderen Beziehungen, wo die Anzahl referenzierter Objekte dynamisch ist, ist ein Array wenig elegant, da die manuellen Vergrößerungen oder Verkleinerungen mühevoll sind.

Dynamische Datenstruktur `ArrayList`

Wollen wir zum Beispiel erlauben, dass ein Spieler mehrere Gegenstände tragen kann oder eine unbekannte Anzahl Spieler sich in einem Raum befinden können, ist eine dynamische Datenstruktur wie `java.util.ArrayList` sinnvoller. Genauer wollen wir uns zwar erst in Kapitel 13, »Einführung in Datenstrukturen und Algorithmen«, mit besagten Datenstrukturen und Algorithmen beschäftigen, doch seien an dieser Stelle schon drei Methoden der `ArrayList` vorgestellt, die Elemente in einer Liste (Sequenz) hält:

- `boolean add(E o)` fügt ein Objekt vom Typ `E` der Liste hinzu.
- `int size()` liefert die Anzahl der Elemente in der Liste.
- `E get(int index)` liefert das Element an der Stelle `index`.

Mit diesem Wissen wollen wir dem Raum Methoden geben, sodass er beliebig viele Spieler aufnehmen kann. Für den unidirektionalen Fall ist die `Player`-Klasse wieder einfach:

Listing 5.54: com/tutego/insel/game/vc/Player.java, Player

```
public class Player
{
    public String name;
```

```
public Player( String name )
{
    this.name = name;
}
```

Der Raum bekommt ein internes Attribut players vom Typ der ArrayList:

```
private ArrayList<Player> players = new ArrayList<Player>();
```

Dass Angaben in spitzen Klammern hinter dem Typ stehen, liegt an den Java Generics – sie besagen, dass die ArrayList nur Player aufnehmen wird und keine anderen Dinge (wie Geister). Die Raum-Klasse wird dann zu:

Listing 5.55: com/tutego/insel/game/vc/Room.java, Room

```
import java.util.ArrayList;

public class Room
{
    private ArrayList<Player> players = new ArrayList<Player>();

    public void addPlayer( Player player )
    {
        players.add( player );
    }

    public void listPlayers()
    {
        for ( Player player : players )
            System.out.println( player.name );
    }
}
```

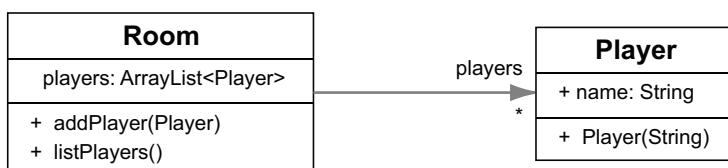


Abbildung 5.22: Room referenziert Player

Die Datenstruktur selbst ist privat, und die `addPlayer()`-Methode fügt einen Spieler in die `ArrayList` ein. Eine Besonderheit bietet die Methode `listPlayers()`, denn sie nutzt das erweiterte `for` zum Durchlaufen aller Spieler. Beim erweiterten `for` ist rechts vom Doppelpunkt nicht nur ein Array erlaubt, sondern auch eine Datenstruktur wie die Liste. Nachdem also zwei Spieler mit `addPlayer()` hinzugefügt wurden, wird `listPlayers()` die beiden Spielernamen ausgeben:

Listing 5.56: com/tutego/insel/game/vc/Playground.java, main()

```
Room oceanLiner = new Room();
oceanLiner.addPlayer( new Player( "Tim" ) );
oceanLiner.addPlayer( new Player( "Jorry" ) );
oceanLiner.listPlayers();                                // Tim Jorry
```

5.8 Vererbung

Schon von Kindheit an lernen wir, Objekte in Beziehung zu setzen. Assoziationen bilden dabei die Hat-Beziehung zwischen Objekten ab: Ein Teddy hat (nach dem Kauf) zwei Arme, der Tisch hat vier Beine, der Wauwau hat ein Fell. Neben der Assoziation von Objekten gibt es eine weitere Form der Beziehung, die Ist-eine-Art-von-Beziehung¹³. Apfel und Birne sind Obstsorten, Lotad, Seedot und Wingull sind verschiedene Pokémons, und »Berg« ist der Sammelbegriff und die Kategorie für K2 und Mount Everest.

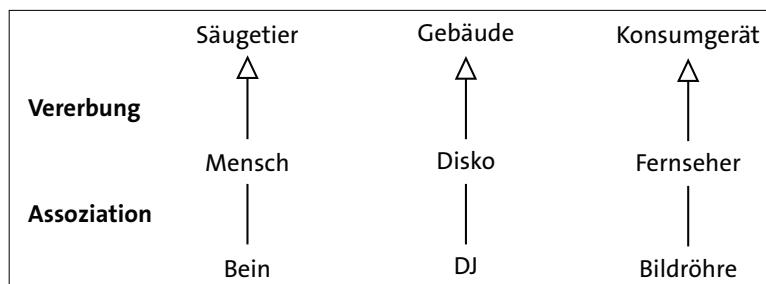


Abbildung 5.23: Vererbung und Assoziation

Das Besondere bei der Ist-eine-Art-von-Beziehung ist die Tatsache, dass die Gruppe gewisse Merkmale für alle Elemente der Gruppe vorgibt.¹⁴ Bei Obst haben wir eine intui-

¹³ So etwas gibt es auch in der Linguistik; dort heißt der Oberbegriff eines Begriffs *Hyperonym* und der Unterbegriff eines Begriffs *Hyponym*.

tive Vorstellung, und jeder Berg hat eine Höhe und einen Namen sowie eine Reihe von Besteigern.

Programmiersprachen drücken Gruppierung und Hierarchiebildung über die Vererbung aus. Vererbung basiert auf der Vorstellung, dass Eltern ihren Kindern Eigenschaften mitgeben. Vererbung bindet die Klassen sehr dicht aneinander. Mittels dieser engen Verbindung können wir später sehen, dass Klassen in gewisser Weise austauschbar sind. Ein Programm kann ausdrücken: Gib mir irgendein Obststück, und es bekommt dann vielleicht einen Apfel oder eine Birne.

5.8.1 Vererbung in Java

Java ordnet Klassen in hierarchischen Relationen an, in der sie Ist-eine-Art-von-Beziehungen bilden. Eine neu deklarierte Klasse erweitert durch das Schlüsselwort `extends` eine andere Klasse. Sie wird dann zur *Unterklasse* (auch *Subklasse*, *Kindklasse* oder *Erweiterungsklasse* genannt). Die Klasse, von der die Unterklasse erbt, heißt *Oberklasse* (auch *Superklasse* oder *Elternklasse*). Durch den Vererbungsmechanismus werden alle sichtbaren Eigenschaften der Oberklasse auf die Unterklasse übertragen. Eine Oberklasse vererbt also Eigenschaften, und die Unterklasse erbt sie.



Hinweis

In Java können nur Untertypen von Klassen deklariert werden. Einschränkungen von primitiven Typen – etwa im Wertebereich oder in der Anzahl der Nachkommastellen – sind nicht möglich. Die Programmiersprache Ada erlaubt das zum Beispiel, und Untertypen sind beim XML-Schema üblich, wo etwa `xs:short` oder `xs:unsignedByte` Untertypen von `xs:integer` sind.

5.8.2 Spielobjekte modellieren

Wir wollen nun eine *Klassenhierarchie* für Objekte in unserem Spiel aufbauen. Bisher haben wir Spieler, Schlüssel und Räume, aber andere Objekte kommen später noch hinzu. Eine Gemeinsamkeit der Objekte ist, dass sie Spielobjekte sind und alle im Spiel

14 Semantische Netzwerke sind in der kognitiven Psychologie ein Erklärungsmodell zur Wissensrepräsentation. Eigenschaften gehören zu Kategorien, die durch Ist-eine-Art-von-Beziehungen hierarchisch verbunden sind. Informationen, die nicht bei einem speziellen Konzept abgespeichert sind, lassen sich von einem übergeordneten Konzept abrufen.

einen Namen haben: Der Raum heißt etwa »Knochenbrecherburg«, der Spieler »James Blond« und der Schlüssel »Magic Wand«.

All diese Objekte sind Spielobjekte und durch ihre Eigenschaft, dass sie alle denselben Namen haben, miteinander verwandt. Die Ist-eine-Art-von-Hierarchie muss aber nicht auf einer Ebene aufhören. Wir könnten uns einen privilegierten Spieler als Spezialisierung vom Spieler vorstellen. Der privilegierte Spieler darf zusätzlich Dinge tun, die ein normaler Spieler nicht tun darf. Damit ist ein normaler Spieler eine Art von Spielobjekt, ein privilegierter Spieler ist eine Art von Spieler, und transitiv gilt weiterhin, dass ein privilegierter Spieler eine Art von Spielobjekt ist.

Schreiben wir die Hierarchie für zwei Spielobjekte auf, für den Spieler und den Raum. Der Raum hat zusätzlich eine Größe. Die Basisklasse (Oberklasse) soll `GameObject` sein:

Listing 5.57: com/tutego/insel/game/vd/GameObject.java, `GameObject`

```
public class GameObject
{
    public String name;
}
```

Der Player soll einfach nur das `GameObject` erweitern und nichts hinzufügen:

Listing 5.58: com/tutego/insel/game/vd/Player.java, `Player`

```
public class Player extends GameObject
{}
```

Syntaktisch wird die Vererbung durch das Schlüsselwort `extends` beschrieben. Die Deklaration der Klasse `Player` trägt den Anhang `extends GameObject` und erbt somit alle sichtbaren Eigenschaften der Oberklasse, also das Attribut `name`. Die vererbten Eigenschaften behalten ihre Sichtbarkeit, sodass eine Eigenschaft `public` weiterhin `public` bleibt. Private Eigenschaften sind für andere Klassen nicht sichtbar, also auch nicht für die Unterklassen; sie erben somit private Eigenschaften nicht.

Der Raum soll neben dem geerbten Namen noch eine Größe besitzen:

Listing 5.59: com/tutego/insel/game/vd/Room.java, `Room`

```
public class Room extends GameObject
{
    public int size;
```

Die Klasse `Room` kann die geerbten Eigenschaften nutzen, also etwa auf die Variable `name` zurückgreifen. Wenn sich in der Oberklasse der Typ der Variablen oder die Implementierung einer Methode ändert, wird auch die Unterklassse diese Änderung zu spüren bekommen. Daher ist die Kopplung mittels Vererbung sehr eng, denn die Unterklassen sind Änderungen der Oberklassen ausgeliefert, da ja Oberklassen nichts von Unterklassen wissen.

Damit ergibt sich das nachfolgende UML-Diagramm. Die Vererbung ist durch einen Pfeil in Richtung der Oberklasse angegeben.

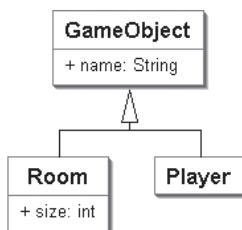


Abbildung 5.24: Room und Player sind zwei Unterklassen von GameObject

Die Unterklassen `Room` und `Player` besitzen alle sichtbaren Eigenschaften der Oberklasse und zusätzlich ihre hinzugefügten:

Listing 5.60: com/tutego/insel/game/vd/Playground.java, Ausschnitt

```

Room clinic = new Room();
clinic.name = "Clinic";           // Geerbtes Attribut
clinic.size = 120000;             // Eigenes Attribut

Player theDoc = new Player();
theDoc.name = "Dr. Schuwibscho";   // Geerbtes Attribut
  
```

5.8.3 Die implizite Basisklasse `java.lang.Object`

Steht keine ausdrückliche `extends`-Anweisung hinter einem Klassennamen – wie in dem Beispiel `GameObject` –, erbt die Klasse automatisch von `Object`, einer impliziten Basisklasse. Steht also keine ausdrückliche Oberklasse, wie bei

```
class GameObject
```

so ist das gleichwertig zu:

```
class GameObject extends Object
```

Alle Klassen haben somit direkt oder indirekt die Klasse `java.lang.Object` als Basisklasse und erben so eine Reihe von Methoden, wie `toString()`.

5.8.4 Einfach- und Mehrfachvererbung *

In Java ist auf direktem Weg nur die *Einfachvererbung* (engl. *single inheritance*) erlaubt, sodass hinter dem Schlüsselwort `extends` lediglich eine einzige Klasse steht. Andere objektorientierte Programmiersprachen, wie C++¹⁵, Python, Perl oder Eiffel, erlauben Mehrfachvererbung und können mehrere Klassen zu einer neuen verbinden. Doch warum bietet Java neben anderen Sprachen wie C#, Objective-C, Simula, Ruby oder Delphi keine Mehrfachvererbung auf Klassenebene?

Nehmen wir an, die Klassen `O1` und `O2` deklarieren beide eine öffentliche Methode `f()`, und `U` ist eine Klasse, die von `O1` und `O2` erbt. Steht in `U` ein Methodenaufruf `f()`, ist nicht klar, welche der beiden Methoden gemeint ist. In C++ löst der Scope-Operator (`::`) das Problem, indem der Entwickler immer angibt, aus welcher Oberklasse die Funktion anzusprechen ist.

Dazu gesellt sich das *Diamanten-Problem* (auch Rauten-Problem genannt). Zwei Klassen, `K1` und `K2`, erben von einer Oberklasse `O` eine Eigenschaft `x`. Eine Unterklass `U` erbt von den Klassen `K1` und `K2`. Lässt sich in `U` auf die Eigenschaft `x` zugreifen? Eigentlich existiert die Eigenschaft ja nur einmal und dürfte kein Grund zur Sorge sein. Dennoch stellt dieses Szenario ein Problem dar, weil der Compiler »vergessen« hat, dass sich `x` in den Unterklassen `K1` und `K2` nicht verändert hat. Mit der Einfachvererbung kommt es erst gar nicht zu diesem Dilemma.

Immer wieder wird diskutiert, ob das Fehlen der Mehrfachvererbung Java einschränkt. Die Antwort ist zu verneinen. Java erlaubt zwar keine multiplen Oberklassen, es erlaubt aber immer noch, mehrere Schnittstellen (Interfaces) zu implementieren und so unterschiedliche Typen anzunehmen.

5.8.5 Die Sichtbarkeit `protected`

Eine Unterklass erbt alle sichtbaren Eigenschaften. Dazu gehören alle `public`-Elemente und, falls sich Unterklass und Oberklasse im gleichen Paket befinden, auch die `paket-`

¹⁵ Bjarne Stroustrup hat Mehrfachvererbung erst in C++ 2.0 (1985–1987) eingeführt.

sichtbaren Eigenschaften. Die Vererbung kann durch `private` eingeschränkt werden, dann sieht keine andere Klasse die Eigenschaften, weder fremde noch Unterklassen.

Neben diesen drei Sichtbarkeiten kommt eine vierte hinzu: `protected`. Diese Sichtbarkeit umfasst (seltsamerweise) zwei Eigenschaften:

- `protected`-Eigenschaften werden an alle Unterklassen vererbt.
- Klassen, die sich im gleichen Paket befinden, können alle `protected`-Eigenschaften sehen, denn `protected` ist eine Erweiterung der Paketsichtbarkeit.

Sind also weitere Klassen im gleichen Paket und Eigenschaften `protected`, ist die Sichtbarkeit für sie `public`. Für andere Nicht-Unterklassen in anderen Paketen sind die `protected`-Eigenschaften `private`. Damit lassen sich die Sichtbarkeiten so ordnen:

`public > protected > paketsichtbar > private`

5.8.6 Konstruktoren in der Vererbung und `super()`

Obwohl Konstruktoren Ähnlichkeit mit Methoden haben, etwa in der Eigenschaft, dass sie überladen werden oder Ausnahmen erzeugen können, werden sie im Gegensatz zu Methoden nicht vererbt. Das heißt, eine UnterkLASSE muss ganz neue Konstruktoren angeben, denn mit den Konstruktoren der OberKLASSE kann ein Objekt der UnterkLASSE nicht erzeugt werden. Ob das nun reine Objektorientierung ist – darüber lässt sich streiten; in der Skriptsprache Python etwa werden auch Konstruktoren vererbt. In Java gehöREN Konstruktoren eigentlich zum statischen Teil einer Klasse. Die Klasse selbst weiß, wie neue Objekte konstruiert werden. Würden wir Konstruktoren eher als Initialisierungsmethoden ansehen, läge es natürlich näher, sie wie Objektmethoden zu behandeln. Dagegen spricht jedoch, dass eine UnterkLASSE mehr Eigenschaften hat und der Konstruktor der OberKLASSE dann nur einen Teil initialisieren würde.

In Java sammelt eine UnterkLASSE zwar automatisch alle sichtbaren Eigenschaften der OberKLASSE, aber die Initialisierung der einzelnen Eigenschaften pro Hierarchie ist immer noch Aufgabe der jeweiligen Konstruktoren in der Hierarchie. Um diese Initialisierung sicherzustellen, ruft Java im Konstruktor einer jeden Klasse (ausgenommen `java.lang.Object`) automatisch den Standard-Konstruktor der OberKLASSE auf, damit die OberKLASSE »ihre« Attribute initialisieren kann. Es ist dabei egal, ob der Konstruktor in der UnterkLASSE parametrisiert ist oder nicht; jeder Konstruktor der UnterkLASSE muss einen Konstruktor der OberKLASSE aufrufen.

Ein Beispiel mit Konstruktorweiterleitung

Sehen wir uns noch einmal die Konstruktorverkettung an:

```
class GameObject
{
}

class Player extends GameObject
{
}
```

Da wir keine expliziten Konstruktoren haben, fügt der Compiler diese ein, und da `GameObject` von `java.lang.Object` erbt, sieht die Laufzeitumgebung die Klassen so:

```
class GameObject
{
    GameObject() { }
}

class Player extends GameObject
{
    Player() { }
}
```

Deutschland sucht den `super()`-Aufruf

Dass automatisch jeder Konstruktor einer Klasse den Standard-Konstruktor der Oberklasse aufruft, lässt sich auch explizit formulieren – das nötige Schlüsselwort ist `super` und formt den Aufruf `super()`. Da der Compiler automatisch `super()` als erste Anweisung in den Konstruktor einfügt, müssen wir das nicht manuell hinschreiben und sollten es uns auch sparen – unsere Fingerkraft ist für andere Dinge wichtig! Ob wir also nun von Hand `super()` im Konstruktor platzieren oder es vom Compiler einsetzen lassen, für die Laufzeitumgebung ist die vorangehende Schreibweise oder die folgende völlig gleich:

```
class GameObject extends Object
{
    GameObject()
{}
```

```
super();           // Ruft Standard-Konstruktor von Object auf
}
}

class Player extends GameObject
{
    Player()
    {
        super();           // Ruft Standard-Konstruktor von GameObject auf
    }
}
```



Hinweis

super() muss immer die erste Anweisung im Konstruktor sein. Beim Aufbau neuer Objekte läuft die Laufzeitumgebung daher als Erstes die Hierarchie nach java.lang.Object ab und beginnt dort von oben nach unten mit der Initialisierung. Kommt der eigene Konstruktor an die Reihe, konnten die Konstruktoren der Oberklasse ihre Werte schon initialisieren.

super() auch bei parametrisierten Konstruktoren

Nicht nur die Standard-Konstruktoren rufen mit super() den Standard-Konstruktor der Oberklasse auf, sondern auch immer die parametrisierten Konstruktoren. Nehmen wir eine Klasse für Außerirdische mit einem parametrisierten Konstruktor für den Namen des Planeten an:

Listing 5.61: com/tutego/insel/game/vd/Alien.java, Alien

```
public class Alien extends GameObject
{
    public String planet;

    public Alien( String planet )
    {
        this.planet = planet;
    }
}
```

Auch wenn es hier keinen Standard-Konstruktor gibt, sondern nur einen parametrisierten, ruft auch dieser automatisch den Standard-Konstruktor der Basisklasse `GameObject` auf. Explizit ausgeschrieben heißt das:

```
public Alien( String planet )
{
    super();          // Ruft automatisch den Standard-Konstruktor von GameObject auf
    this.planet = planet;
}
```

Natürlich muss `super()` wieder als Erstes stehen.

super() mit Argumenten füllen

Mitunter ist es nötig, aus der Unterklassie nicht nur den Standard-Konstruktor anzu-steuern, sondern einen anderen (parametrisierten) Konstruktor der Oberklassie anzu-sprechen. Dazu gibt es das `super()` mit Argumenten.

Der Aufruf von `super()` kann parametrisiert erfolgen, sodass nicht der Standard-Kon-struktor, sondern ein parametrisierter Konstruktor aufgerufen wird. Gründe dafür könnten sein:

- Ein parametrisierter Konstruktor der Unterklassie leitet die Argumente an die Ober-klasse weiter; es soll nicht der Standard-Konstruktor aufgerufen werden, da der Ober-klassen-Konstruktor das Attribut annehmen und verarbeiten soll.
- Wenn wir keinen Standard-Konstruktor in der Oberklassie vorfinden, müssen wir in der Unterklassie mittels `super(Argument ...)` einen speziellen, parametrisierten Kon-struktor aufrufen.

Gehen wir Schritt für Schritt eine Vererbungshierarchie durch, um zu verstehen, dass ein `super()` mit Parameter nötig ist.

Beginnen wir mit einer Klasse `Alien`, die in einem parametrisierten Konstruktor den Pla-netennamen erwartet:

Listing 5.62: Alien.java

```
public class Alien
{
    public String planet;
    public Alien( String planet ) { this.planet = planet; }
}
```

Erweitert eine Klasse `Grob` für eine besondere Art von Außerirdischen die Klasse `Alien`, kommt es zu einem Compilerfehler:

```
public class Grob extends Alien { } // ☹ Compilerfehler
```

Der Fehler vom Eclipse-Compiler ist: »Implicit super constructor `Alien()` is undefined. Must explicitly invoke another constructor.«

Der Grund ist simpel: `Grob` enthält einen vom Compiler generierten vorgegebenen Konstruktor, der mit `super()` nach einem Standard-Konstruktor in `Alien` sucht – den gibt es aber nicht. Wir müssen daher entweder einen Standard-Konstruktor in der Oberklasse anlegen (was bei nicht modifizierbaren Klassen natürlich nicht geht) oder das `super()` in `Grob` so einsetzen, dass es mit einem Argument den parametrisierten Konstruktor der Oberklasse aufruft. Das kann so aussehen:

Listing 5.63: `Grob.java`

```
public class Grob extends Alien
{
    public Grob()
    {
        super( "Locutus" ); // Alle Grobs leben auf Locutus
    }
}
```

Es spielt dabei keine Rolle, ob `Grob` einen Standard-Konstruktor oder einen parametrisierten Konstruktor besitzt: In beiden Fällen müssen wir mit `super()` einen Wert an den Konstruktor der Basisklasse übergeben. Oftmals leiten Unterklassen einfach nur den Konstruktorwert an die Oberklasse weiter:

```
public class Grob extends Alien
{
    public Grob( String planet )
    {
        super( planet );
    }
}
```

Der this()-und-super()-Konflikt *

this() und super() haben eine Gemeinsamkeit: Beide wollen die erste Anweisung eines Konstruktors sein. Es kommt vor, dass es mit super() einen parametrisierten Aufruf des Konstruktors der Basisklasse gibt, aber gleichzeitig auch ein this() mit Parametern, um in einem zentralen Konstruktor alle Initialisierungen vornehmen zu können. Beides geht aber leider nicht. Die Lösung besteht darin, auf das this() zu verzichten und den gemeinsamen Programmcode in eine private Methode zu setzen. Das kann so aussehen:

Listing 5.64: ColoredLabel.java

```
import java.awt.Color;
import javax.swing.JLabel;

public class ColoredLabel extends JLabel
{
    public ColoredLabel()
    {
        initialize( Color.BLACK );
    }

    public ColoredLabel( String label )
    {
        super( label );
        initialize( Color.BLACK );
    }

    public ColoredLabel( String label, Color color )
    {
        super( label );
        initialize( color );
    }

    private void initialize( Color color )
    {
        setForeground( color );
    }
}
```

Die farbige Beschriftung `ColoredLabel` ist ein spezielles `JLabel`. Es kann auf drei Arten initialisiert werden, wobei bei allen Herangehensweisen die Aufgabe gleich ist, dass eine Farbe gespeichert werden muss. Das übernimmt die Methode `initialize()`, die alle Konstruktoren aufrufen. Hier wird dann Beliebiges platziert, was alle Konstruktoren gerne initialisieren wollen.

Zusammenfassung: Konstruktoren und Methoden

Methoden und Konstruktoren haben einige Gemeinsamkeiten in der Signatur, weisen aber auch einige wichtige Unterschiede auf, wie den Rückgabewert oder den Gebrauch von `this` und `super`. Tabelle 5.5 fasst die Unterschiede und Gemeinsamkeiten zusammen:¹⁶

Benutzung	Konstruktoren	Methoden
Modifizierer	Sichtbarkeit <code>public</code> , <code>protected</code> , <code>paketsichtbar</code> und <code>private</code> . Können <i>nicht</i> <code>abstract</code> , <code>final</code> , <code>native</code> , <code>static</code> oder <code>synchronized</code> sein.	Sichtbarkeit <code>public</code> , <code>protected</code> , <code>paketsichtbar</code> und <code>private</code> . Können <code>abstract</code> , <code>final</code> , <code>native</code> , <code>static</code> oder <code>synchronized</code> sein.
Rückgabewert	kein Rückgabewert, auch nicht <code>void</code>	Rückgabetyp oder <code>void</code>
Bezeichner-name	Gleicher Name wie die Klasse. Beginnt mit einem Großbuchstaben.	Beliebig. Beginnt mit einem Kleinbuchstaben.
<code>this</code>	<p><code>this</code> ist eine Referenz in Objektmethoden und Konstruktoren, die sich auf das aktuelle Exemplar bezieht.</p> <p><code>this()</code> bezieht sich auf einen anderen Konstruktor der gleichen Klasse. Wird <code>this()</code> benutzt, muss es in der ersten Zeile stehen.</p>	
<code>super</code>	<p><code>super</code> ist eine Referenz mit dem Namensraum der Oberklasse. Damit lassen sich überschriebene Objektmethoden aufrufen.</p> <p><code>super()</code> ruft einen Konstruktor der Oberklasse auf. Wird es benutzt, muss es die erste Anweisung sein.</p>	
Vererbung	Konstruktoren werden nicht vererbt.	Sichtbare Methoden werden vererbt.

Tabelle 5.6: Gegenüberstellung von Konstruktoren und Methoden

¹⁶ Schon seltsam, dass `synchronized` nicht erlaubt ist, aber ein Konstruktor ist implizit `synchronized`.

5.9 Typen in Hierarchien

Die Vererbung bringt einiges Neues in Bezug auf Kompatibilität von Typen mit. Dieser Abschnitt beschäftigt sich mit den Fragen, welche Typen kompatibel sind und wie sich ein Typ zur Laufzeit testen lässt.

5.9.1 Automatische und explizite Typanpassung

Die Klassen Room und Player haben wir als Unterklassen von GameObject modelliert. Die eigene Oberklasse GameObject erweitert selbst keine explizite Oberklasse, sodass implizit java.lang.Object die Oberklasse ist. In GameObject gibt es das Attribut name, das Player und Room erben, und der Raum hat zusätzlich size für die Raumgröße.

Ist-eine-Art-von-Beziehung und die automatische Typanpassung

Mit der Ist-eine-Art-von-Beziehung ist eine interessante Eigenschaft verbunden, die wir bemerken, wenn wir die Zusammenhänge zwischen den Typen beachten:

- Ein Raum ist ein Spielobjekt.
- Ein Spieler ist ein Spielobjekt.
- Ein Spielobjekt ist ein java.lang.Object.
- Ein Spieler ist ein java.lang.Object.
- Ein Raum ist ein java.lang.Object.
- Ein Raum ist ein Raum.
- Ein Spieler ist ein Spieler.

Kodieren wir das in Java:

Listing 5.65: com/tutego/insel/game/vd/TypeSuptype.java, main()

```
Player    playerIsPlayer    = new Player();
GameObject gameObjectIsPlayer = new Player();
Object    objectIsPlayer    = new Player();
Room     roomIsRoom        = new Room();
GameObject gameObjectIsRoom = new Room();
Object    objectIsRoom     = new Room();
```

Es gilt also, dass immer dann, wenn ein Typ gefordert ist, auch ein Untertyp erlaubt ist. Der Compiler führt eine implizite Typanpassung durch. Wir werden uns dieses sogenannte liskovsche Substitutionsprinzip im folgenden Abschnitt anschauen.

Was wissen Compiler und Laufzeitumgebung über unser Programm?

Wichtig ist, zu beobachten, dass Compiler und Laufzeitumgebung unterschiedliche Dinge wissen. Durch den new-Operator gibt es zur Laufzeit nur zwei Arten von Objekten: Player und Room. Auch dann, wenn es

```
GameObject gameObjectIsRoom = new Room();
```

heißt, referenziert gameObjectIsRoom zur Laufzeit ein Room-Objekt. Der Compiler aber »vergisst« dies und glaubt, gameObjectIsRoom wäre nur ein einfaches GameObject. In der Klasse GameObject ist jedoch nur name deklariert, aber kein Attribut size, obwohl das tatsächliche Room-Objekt natürlich eine size kennt. Auf size können wir aber erst einmal nicht zugreifen:

```
println( gameObjectIsRoom.name );
println( gameObjectIsRoom.size ); // ☹ gameObjectIsRoom.size cannot
                                //      be resolved or is not a field
```

Schreiben wir noch einschränkender

```
Object objectIsRoom = new Room();
println( objectIsRoom.name ); // ☹ objectIsRoom.name cannot be
                            //      resolved or is not a field
println( objectIsRoom.size ); // ☹ objectIsRoom.size cannot be
                            //      resolved or is not a field
```

so steht hinter der Referenzvariablen objectIsRoom ein vollständiges Room-Objekt, aber weder size noch name sind nutzbar; es bleiben nur die Fähigkeiten aus java.lang.Object.



Begrifflichkeit

Um den Compiler-Typ vom JVM-Typ zu unterscheiden, nutzen einige Buchautoren die Begriffe *Referenztyp* und *Objekttyp*. Im Fall von `GameObject p = new Player();` ist GameObject der Referenztyp und Player der Objekttyp. Der Compiler sieht nur den Referenztyp, aber nicht den Objekttyp.

Explizite Typanpassung

Diese Typeinschränkung gilt auch an anderer Stelle. Ist eine Variable vom Typ Room deklariert, können wir die Variable nicht mit einem »kleineren« Typ initialisieren:

```
GameObject go          = new Room();    // Raum zur Laufzeit
Room      cubbyhole  = go;           // ☹ Type mismatch: cannot convert from
                                    // GameObject to Room
```

Auch wenn zur Laufzeit go ein Room referenziert, können wir cubbyhole nicht damit initialisieren. Der Compiler kennt go nur unter dem »kleineren« Typ GameObject, und das reicht nicht zur Initialisierung des »größeren« Typs Room.

Es ist aber möglich, das Objekt hinter go durch eine explizite Typumwandlung für den Compiler wieder zu einem vollwertigen Room mit Größe zu machen:

```
Room      cubbyhole = (Room) go;
System.out.println( cubbyhole.size ); // Room hat das Attribut size
```

Unmögliche Anpassung und ClassCastException

Dies funktioniert aber lediglich dann, wenn go auch wirklich einen Raum referenziert. Dem Compiler ist das in dem Moment relativ egal, sodass auch Folgendes ohne Fehler compiliert wird:

Listing 5.66: com/tutego/insel/game/vd/ClassCastExceptionDemo.java, main()

```
GameObject go          = new Player();
Room      cubbyhole  = (Room) go;     // ☹ ClassCastException
System.out.println( cubbyhole.size );
```

Zur Laufzeit kommt es bei diesem Kuckucksobjekt zu einer ClassCastException:

```
Exception in thread "main" java.lang.ClassCastException: com.tutego.insel.game.vd.
Player cannot be cast to com.tutego.insel.game.vd.Room
at com.tutego.insel.game.vd.ClassCastExceptionDemo.main(ClassCastExceptionDemo.java:8)
```

5.9.2 Das Substitutionsprinzip

Stellen wir uns vor, Bekannte kommen ausgehungert von einer Wandertour zurück und fragen: »Haste was zu essen?« Die Frage zielt wohl darauf ab, dass es bei Hunger ziemlich egal ist, was wir anbieten, wichtig ist nur etwas Essbares. Daher können wir Eis, aber auch Frittierzett und gegrillte Heuschrecken anbieten.

Diese Ausgangslage führt uns zu einem wichtigen Konzept in der Objektorientierung: »Wer wenig will, kann viel bekommen.« Genauer gesagt: Wenn Unterklassen wie Player

oder Room die Oberklasse GameObject erweitern, können wir überall, wo GameObject gefordert wird, auch einen Player oder Room übergeben, da beide ja vom Typ GameObject sind und wir mit der Unterklassie nur spezieller werden. Auch können wir weitere Unterklassen von Player und Room übergeben, da auch die Unterklassie weiterhin zusätzlich das »Gen« GameObject in sich trägt. Alle diese Dinge wären vom Typ GameObject und daher typkompatibel. Wenn nun etwa eine Methode eine Übergabe vom Typ GameObject erwartet, kann sie alle Eigenschaften von GameObject nutzen, also das Attribut name, da ja alle Unterklassen die Eigenschaften erben und Unterklassen die Eigenschaften nicht »wegzaubern« können. Derjenige, dem wir »mehr« übergeben, kann zwar nichts mit den Erweiterungen anfangen, ablehnen wird er das Objekt aber nicht, weil es alle geforderten Eigenschaften aufweist.



Weil anstelle eines Objekts auch ein Objekt der Unterklassie auftauchen kann, sprechen wir von *Substitution*. Das Prinzip wurde von der Professorin Barbara Liskov¹⁷ formuliert und heißt daher auch *liskovsches Substitutionsprinzip*.

Die folgende Klasse `QuoteNameFromGameObject` nutzt diese Eigenschaft. Sie fordert in der Methode `quote()` irgendein GameObject, von dem bekannt ist, dass es ein Attribut `name` hat. Im Hauptprogramm kann `quote()` ein Spieler oder Raum übergeben werden:

¹⁷ Die Zeitschrift »Discover« zählt sie zu den 50 wichtigsten Frauen in der Wissenschaft.

Listing 5.67: com/tutego/insel/game/vd/QuoteNameFromGameObject.java,
QuoteNameFromGameObject

```
public class QuoteNameFromGameObject
{
    public static void quote( GameObject go )
    {
        System.out.println( "'" + go.name + "' " );
    }

    public static void main( String[] args )
    {
        GameObject player = new Player();
        player.name = "Godman";
        quote( player );           // 'Godman'

        GameObject room = new Room();
        room.name = "Hogwurz";
        quote( room );           // 'Hogwurz'
    }
}
```

Mit `GameObject` haben wir eine Basisklasse geschaffen, die verschiedenen Unterklassen Grundfunktionalität beibringt, in unserem Fall das Attribut `name`. So liefert die Basisklasse einen gemeinsamen Nenner, etwa gemeinsame Attribute oder Methoden, die jede Unterklasse besitzen wird.

In der Java-Bibliothek finden sich zahllose weitere Beispiele. Die `println(Object)`-Methode ist so ein Beispiel. Die Methode nimmt beliebige Objekte entgegen, denn der ParameterTyp ist `Object`. Die Substitution besagt, dass wir alle Objekte dort einsetzen können, da alle Klassen von `Object` abgeleitet sind.

5.9.3 Typen mit dem instanceof-Operator testen

Der relationale Operator `instanceof` hilft dabei, Exemplare auf ihre Verwandtschaft mit einem Referenztyp zu prüfen. Er stellt zur Laufzeit fest, ob eine Referenz ungleich `null` und von einem bestimmten Typ ist. Der Operator ist binär, hat also zwei Operanden:

Listing 5.68: com/tutego/insel/game/vd/InstanceOfDemo.java, main()

```
System.out.println( "Toll" instanceof String );      // true
System.out.println( "Toll" instanceof Object );      // true
System.out.println( new Player() instanceof Object ); // true
```

Alles in doppelten Anführungsstrichen ist ein String, sodass instanceof String wahr ergibt. Für den zweiten und dritten Fall gilt: Alle Objekte gehen irgendwie aus Object hervor und sind somit logischerweise Erweiterungen.



Hinweis

Der Operator instanceof testet ein Objekt auf seine Hierarchie. So ist zum Beispiel o instanceof Object für jedes Objekt o wahr, denn jedes Objekt ist immer Kind von java.lang.Object. Die Programmiersprache Smalltalk unterscheidet hier mit zwei Nachrichten isMemberOf (exakt) und isKindOf (wie Javas instanceof). Um den exakten Typ zu testen, lässt sich mit dem Class-Objekt arbeiten, etwa wie im Ausdruck o.getClass().equals(Object.class), der testet, ob o genau ein Object-Objekt ist.

Die bisherigen Beziehungen hätte der Compiler bereits herausfinden können. Vervollständigen wir das, um zu sehen, dass instanceof wirklich zur Laufzeit den Test durchführen muss. In allen Fällen ist das Objekt zur Laufzeit ein Raum:

```
Room      go1 = new Room();
System.out.println( go1 instanceof Room );      // true
System.out.println( go1 instanceof GameObject ); // true
System.out.println( go1 instanceof Object );     // true

GameObject go2 = new Room();
System.out.println( go2 instanceof Room );      // true
System.out.println( go2 instanceof GameObject ); // true
System.out.println( go2 instanceof Object );     // true
System.out.println( go2 instanceof Player );    // false

Object      go3 = new Room();
System.out.println( go3 instanceof Room );      // true
System.out.println( go3 instanceof GameObject ); // true
System.out.println( go3 instanceof Object );     // true
```

```
System.out.println( go3 instanceof Player );      // false
System.out.println( go3 instanceof String );      // false
```

Der Compiler lässt aber nicht alles durch. Liegen zwei Typen überhaupt nicht in der Typ-hierarchie, lehnt der Compiler den Test ab, da die Vererbungsbeziehungen schon inkompatibel sind:

```
System.out.println( "Toll" instanceof StringBuilder );
// ✗ Incompatible conditional operand types String and StringBuilder
```

Der Ausdruck ist falsch, da `StringBuilder` keine Basisklasse für `String` ist.

Zum Schluss:

```
Object ref1 = new int[ 100 ];
System.out.println( ref1 instanceof String );
System.out.println( new int[100] instanceof String ); // ✗ Compilerfehler
```

Hinweis

Mit `instanceof` lässt sich der Programmfluss aufgrund der tatsächlichen Typen steuern, etwa mit Anweisungen wie `if(reference instanceof Typ) A else B`. In der Regel zeigt Kontrolllogik dieser Art aber tendenziell ein Designproblem an und kann oft anders gelöst werden. Das dynamische Binden ist so eine Lösung; sie wird später vorgestellt.

instanceof und null

Ein `instanceof`-Test mit einer Referenz, die `null` ist, gibt immer `false` zurück:

```
Object ref2 = null;
System.out.println( ref2 instanceof String );      // null
```

Das leuchtet ein, denn `null` entspricht ja keinem konkreten Objekt.

Tipp

Da `instanceof` einen `null`-Test enthält, sollte statt etwa

```
if ( s != null && s instanceof String )
```

immer vereinfacht so geschrieben werden:

```
if ( s instanceof String )
```



5.10 Methoden überschreiben

Wir haben gesehen, dass eine Unterklasse durch Vererbung die sichtbaren Eigenschaften ihrer Oberklasse erbt. Die Unterklasse kann nun wiederum Methoden hinzufügen. Dabei zählen überladene Methoden – also Methoden, die den gleichen Namen wie eine andere Methode aus einer Oberklasse tragen, aber eine andere Parameteranzahl oder andere Parametertypen haben – zu ganz normalen, hinzugefügten Methoden.

5.10.1 Methoden in Unterklassen mit neuem Verhalten ausstatten

Besitzt eine Unterklasse eine Methode mit dem gleichen Methodenamen und der exakten Parameterliste (also der gleichen Signatur) wie schon die Oberklasse, so *überschreibt* die Unterklasse die Methode der Oberklasse. Ist der Rückgabetyp `void` oder ein primitiver Typ, so muss er in der überschreibenden Methode der gleiche sein. Bei Referenztypen kann der Rückgabetyp etwas variieren, doch das werden wir später genauer sehen.

Implementiert die Unterklasse die Methode neu, so sagt sie auf diese Weise: »Ich kann's besser.« Die *überschreibende Methode* der Unterklasse kann demnach den Programmcode spezialisieren und Eigenschaften nutzen, die in der Oberklasse nicht bekannt sind. Die *überschriebene Methode* der Oberklasse ist dann erst einmal aus dem Rennen, und ein Methodenaufruf auf einem Objekt der Unterklasse würde sich in der überschriebenen Methode verfangen.



Hinweis

Wir sprechen nur von überschriebenen Methoden und nicht von überschriebenen Attributnen, da Attribute nicht überschrieben, sondern nur *überlagert* werden. Attribute werden auch nicht dynamisch gebunden – eine Eigenschaft, die später in Abschnitt »Eine letzte Spielerei mit Javas dynamischer Bindung und überschatteten Attributen*« genauer erklärt wird.

Überschreiben von `toString()`

Aus der absoluten Basisklasse `java.lang.Object` bekommen alle Unterklassen eine Methode `toString()` vererbt, die meist zu Debug-Zwecken eine Objektkennung ausgibt:

Listing 5.69: java/lang/Object.java, `toString()`

```
public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Die Methode liefert den Namen der Klasse, gefolgt von einem "@" und einer hexadezimalen Kennung. Die Klasse `GameObject` ohne eigenes `toString()` soll die Wirkung testen:

Listing 5.70: com/tutego/insel/game/ve/GameObject.java, `GameObject`

```
public class GameObject
{
    private String name;

    public String getName()
    {
        return name;
    }

    public void setName( String name )
    {
        this.name = name;
    }
}
```

Auf einem `GameObject`-Objekt liefert `toString()` eine etwas kryptische Kennung:

```
GameObject go = new GameObject();
out.println( go.toString() ); // com.tutego.insel.game.ve.GameObject@e48e1b
```

Es ist also eine gute Idee, `toString()` in den Unterklassen zu überschreiben. Eine Stringkennung sollte den Namen der Klasse und die Zustände eines Objekts beinhalten. Für einen Raum, der einen (geerbten) Namen und eine eigene Größe hat, kann dies wie folgt aussehen:

Listing 5.71: com/tutego/insel/game/ve/Room.java, `Room`

```
public class Room extends GameObject
{
    private int size;
```

```

public void setSize( int size )
{
    if ( size > 0 )
        this.size = size;
}

public int getSize()
{
    return size;
}

@Override public String toString()
{
    return String.format( "Room[name=%s, size=%d]", getName(), getSize() );
}
}

```

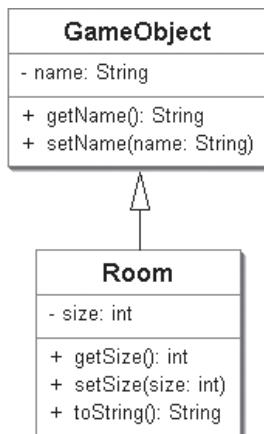


Abbildung 5.25: Room ist eine Unterklasse von GameObject und hat ein eigenes `toString()`

Und der Test sieht so aus:

Listing 5.72: com/tutego/insel/game/ve/Playground.java, main()

```

Room winterfield = new Room();
winterfield.setName( "Winterfield" );

```

```
winterfield.setSize( 2040000 );
System.out.println( winterfield ); // Room[name=Winterfield, size=2040000]
```

Zur Erinnerung: Ein `println()` auf einem beliebigen Objekt ruft die `toString()`-Methode von diesem Objekt auf.

Die Annotation `@Override`

Unsere Beispielklasse `Room` nutzt die Annotation `@Override` an der Methode `toString()` und macht auf diese Weise deutlich, dass die Unterklasse eine Methode der Oberklasse überschreibt. Die Annotation `@Override` bedeutet nicht, dass diese Methode in Unterklassen überschrieben werden muss, sondern nur, dass sie selbst eine Methode überschreibt. Annotationen sind zusätzliche Modifizierer, die entweder vom Compiler überprüft werden oder von uns nachträglich abgefragt werden können. Obwohl wir die Annotation `@Override` nicht nutzen müssen, hat sie den Vorteil, dass der Compiler überprüft, ob wir tatsächlich eine Methode aus der Oberklasse überschreiben – haben wir uns im Methodennamen verschrieben und würde die Unterklasse auf diese Weise eine neue Methode hinzufügen, so würde der Compiler das als Fehler melden. Fehler wie `toString()` fallen schnell auf. Überladene Methoden und überschriebene Methoden sind etwas anderes, da eine überladene Methode mit der Ursprungsmethode nur »zufällig« den Namen teilt, aber sonst keinen Bezug zur Logik hat. Und so hilft `@Override`, dass Entwickler wirklich Methoden überschreiben und nicht aus Versehen Methoden mit falschen Parametern überladen.

Hinweis

Seit Java 6 ist die Annotation `@Override` auch an Methoden gültig, die Operationen aus Schnittstellen implementieren; unter Java 5 war das noch ein Fehler. Die Beispiele der Insel sind der Syntax von Java 6 geschrieben. Zu Schnittstellen kommen wir im Abschnitt 5.13.



Garantiert überschrieben? *

Überschrieben werden nur Methoden, die exakt mit der Signatur einer Methode aus der Oberklasse übereinstimmen. Sind Parametertypen gleich, so müssen sie auch aus dem gleichen Paket stammen. So kann es passieren, dass eine Unterklasse `Sub` doch nicht die Methode `printDate()` aus `Super` überschreibt, obwohl es auf den ersten Blick so aussieht:

Deklaration der Basisklasse	Überladene, keine überschreibende Methode
<pre>import java.util.Date; public class Super { void printDate(Date date) {} }</pre>	<pre>import java.sql.Date; public class Sub extends Super { void printDate(Date date) {} }</pre>

Zwar sehen die Signaturen optisch gleich aus, da aber `Date` aus verschiedenen Paketen stammt, ist die Signatur nicht wirklich gleich. Die Methode aus `printDate()` aus `Sup` überlädt `printDate()` aus `Super`, aber überschreibt sie nicht. Letztendlich bietet `Sub` zwei Methoden:

- `void printDate(java.util.Date date) {}`
- `void printDate(java.sql.Date date) {}`

Es ist gut, wenn eine überschreibende Methode explizit kenntlich gemacht wird. Dazu gibt es die Annotation `@Override`, die an die Methode der Unterklasse gesetzt werden sollte. Denn verspricht eine Methode das Überschreiben, doch macht sie das, wie in unserem Beispiel, nicht, ergibt das einen Compilerfehler, und dem Entwickler wird der Fehler vor Augen geführt. Mit `@Override` wäre dieser Fehler aufgefallen.

5.10.2 Mit super an die Eltern

Wenn wir eine Methode überschreiben, dann entscheiden wir uns für eine gänzlich neue Implementierung. Was ist aber, wenn die Funktionalität im Großen und Ganzen gut war und nur eine Kleinigkeit fehlte? Im Fall der überschriebenen `toString()`-Methode realisiert die Unterklasse eine völlig neue Implementierung und bezieht sich dabei nicht auf die Logik der Oberklasse.

Möchte eine Unterklasse sagen: »Was meine Eltern können, ist doch gar nicht so schlecht«, kann mit der speziellen Referenz `super` auf die Eigenschaften im Namensraum der Oberklasse zugegriffen werden (natürlich ist das Objekt hinter `super` und `this` das gleiche, nur der Namensraum ist ein anderer). Auf diese Weise können Unterklassen immer noch etwas Eigenes machen, aber die Realisierung aus der Elternklasse ist weiterhin verfügbar.

In unserem Spiel gibt es Räume mit einer Größe. Die Größe lässt sich mit `setSize()` setzen und mit `getSize()` erfragen. Eine Konsistenzprüfung in `setSize()` erlaubt nur Grö-

ßen echt größer null. Wenn nun eine Garage als besonderer Raum eine gewisse Größe nicht überschreiten darf – sonst wäre er keine Garage –, lässt sich `setSize()` überschreiben und immer dann das `setSize()` der Oberklasse zum tatsächlichen Setzen des Attributs aufrufen, wenn die Größe im richtigen Bereich lag:

Listing 5.73: com/tutego/insel/game/ve/Garage.java, Garage

```
public class Garage extends Room
{
    private static final int MAX_GARAGE_SIZE = 40;

    @Override public void setSize( int size )
    {
        if ( size <= MAX_GARAGE_SIZE )
            super.setSize( size );
    }
}
```

Stünde statt `super.setSize(size)` nur `setSize(size)`, würde ein Methodenaufruf in die Endlosrekursion führen.

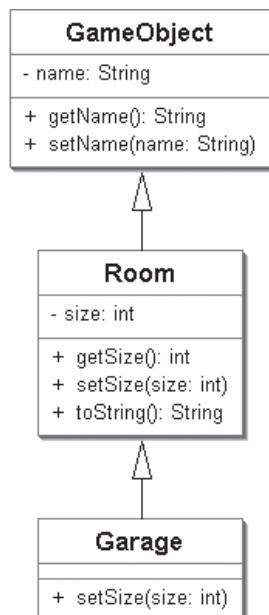


Abbildung 5.26: Garage bietet ein eigenes `setSize()` wie die Oberklasse Room

Eigenschaften der super-Referenz *

Nicht nur in überschriebenen Methoden kann die super-Referenz sinnvoll eingesetzt werden: Sie ist auch interessant, wenn Methoden der Oberklasse aufgerufen werden sollen und nicht eigene überschriebene. So macht das folgende Beispiel klar, dass auf jeden Fall `toString()` der Oberklasse `Object` aufgerufen werden soll und nicht die eigene überschriebene Variante:

Listing 5.74: `ToStringFromSuper.java`

```
public class ToStringFromSuper
{
    public ToStringFromSuper()
    {
        System.out.println( super.toString() ); // Aufruf von Object toString()
    }

    @Override public String toString()
    {
        return "Nein";
    }

    public static void main( String[] args )
    {
        new ToStringFromSuper();           // ToStringFromSuper@3e25a5
    }
}
```

Natürlich kann `super` nur dann eingesetzt werden, wenn in der Oberklasse die Methode eine gültige Sichtbarkeit hat. Es ist also nicht möglich, mit diesem Konstrukt das Geheimnisprinzip zu durchbrechen.

Eine Aneinanderreihung von `super`-Schlüsselwörtern bei einer tieferen Vererbungshierarchie ist nicht möglich. Hinter einem `super` muss eine Objekteigenschaft stehen; sie gilt also für eine überschriebene Methode oder ein überlagertes Attribut. Anweisungen wie `super.super.lol()` sind somit immer ungültig. Eine UnterkLASSE empfängt alle Eigenschaften ihrer Oberklassen als Einheit und unterscheidet nicht, aus welcher Hierarchie etwas kommt.

5.10.3 Finale Klassen und finale Methoden

Soll eine Klasse keine Unterklassen bilden, werden Klassen mit dem Modifizierer `final` versehen. Dadurch lässt sich vermeiden, dass Unterklassen Eigenschaften nachträglich verändern können. Ein Versuch, von einer finalen Klasse zu erben, führt zu einem Compilerfehler. Dies schränkt zwar die objektorientierte Wiederverwendung ein, wird aber aufgrund von Sicherheitsaspekten in Kauf genommen. Eine Passwortüberprüfung soll zum Beispiel nicht einfach überschrieben werden können.

In der Java-Bibliothek gibt es eine Reihe finaler Klassen, von denen wir einige bereits kennen:

- `String, StringBuffer, StringBuilder`
- `Integer, Double ... (Wrapper-Klassen)`
- `Math`
- `System`
- `Font, Color`

Tipp

Eine `protected`-Eigenschaft in einer als `final` deklarierten Klasse ergibt wenig Sinn, da ja keine Unterklasse möglich ist, die diese Methode oder Variable nutzen kann. Daher sollte die Eigenschaft dann paketsichtbar sein (`protected` enthält ja `paketsichtbar`) oder gleich `private` oder `public`.



Nicht überschreibbare (finale) Methoden

In der Vererbungshierarchie möchte ein Designer in manchen Fällen verhindern, dass Unterklassen eine Methode überschreiben und mit neuer Logik implementieren. Das verhindert der zusätzliche Modifizierer `final` an der Methodendeklaration. Da Methodenaufrufe immer dynamisch gebunden werden, könnte ein Aufrufer unbeabsichtigt in der Unterklasse landen, was finale Methoden vermeiden.

Dazu ein Beispiel: Das `GameObject` speichert einen Namen intern im `protected`-Attribut `name` und erlaubt Zugriff nur über einen Setter/Getter. Die Methode `setName()` testet, ob der Name ungleich `null` ist und mindestens ein Zeichen enthält. Diese Methode soll `final` sein, denn eine Unterklasse könnte diese Zugriffsbeschränkungen leicht aushebeln und selbst die `protected`-Variable `name` beschreiben, auf die die Unterklasse Zugriff hat:

Listing 5.75: com/tutego/insel/game/vg/GameObject.java, GameObject

```
public class GameObject
{
    protected String name;

    public String getName()
    {
        return name;
    }

    public final void setName( String name )
    {
        if ( name != null && !name.isEmpty() )
            this.name = name;
    }
}
```

Bei dem Versuch, in einer Unterklasse die Methode zu überschreiben, meldet der Compiler einen Fehler:

Listing 5.76: com/tutego/insel/game/vg/Player.java, Player

```
public class Player extends GameObject
{
    @Override
    public void setName( String name )      // ☹
        ^ Cannot override the final method from GameObject
    {
        this.name = name;
    }
}
```

→ **Hinweis**

Auch private Methoden können final sein, aber private Methoden lassen sich ohnehin nicht überschreiben (sie werden überlagert), sodass final überflüssig ist.

5.10.4 Kovariante Rückgabetypen

Überschreibt eine Methode mit einem Referenztyp als Rückgabe eine andere, so kann die überschreibende Methode einen Untertyp des Rückgabetyyps der überschriebenen Methode als Rückgabotyp besitzen. Das nennt sich *kovarianter Rückgabotyp* und ist sehr praktisch, da sich auf diese Weise Entwickler oft explizite Typanpassungen sparen können.

Ein Beispiel soll dies verdeutlichen: Die Klasse `Loudspeaker` deklariert eine Methode `getThis()`, die lediglich die `this`-Referenz zurückgibt. Eine Unterklasse überschreibt die Methode und liefert den spezielleren Untertyp:

Listing 5.77: BigBassLoudspeaker.java

```
class Loudspeaker
{
    Loudspeaker getThis()
    {
        return this;
    }
}

class BigBassLoudspeaker extends Loudspeaker
{
    @Override
    BigBassLoudspeaker getThis()          // statt »Loudspeaker getThis()«
    {
        return this;
    }
}
```

Die Unterklasse `BigBassLoudspeaker` überschreibt die Methode `getThis()`, auch wenn der Rückgabotyp nicht `Loudspeaker`, sondern `BigBassLoudspeaker` heißt.

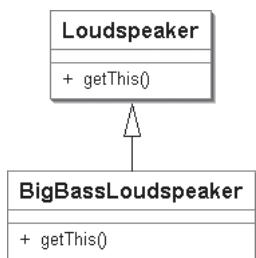


Abbildung 5.27: BigBassLoudspeaker ist ein spezieller LoudSpeaker

Der Rückgabetyp muss auch nicht zwingend der Typ der eigenen Klasse sein. Gäbe es zum Beispiel mit Plasmatweeter eine zweite Unterklasse von Loudspeaker, so könnte `getThis()` von BigBassLoudspeaker auch den Rückgabetyp Plasmatweeter deklarieren. Hauptsache, der Rückgabetyp der überschreibenden Methode ist eine Unterklasse des Rückgabetyps der überschriebenen Methode der Basisklasse.



Hinweis

Merkwürdig in diesem Zusammenhang ist, dass es in Java schon immer veränderte Zugriffsrechte gegeben hat. Eine Unterklasse kann die Sichtbarkeit erweitern. Auch bei Ausnahmen kann eine Unterklasse speziellere Ausnahmen beziehungsweise ganz andere Ausnahmen als die Methode der Oberklasse erzeugen.

5.10.5 Array-Typen und Kovarianz *

Die Aussage »Wer wenig will, kann viel bekommen« gilt auch für Arrays, denn wenn eine Klasse U eine Unterklasse einer Klasse O ist, ist auch $U[]$ ein Untertyp von $O[]$. Diese Eigenschaft nennt sich *Kovarianz*. Da `Object` die Basisklasse aller Objekte ist, kann ein `Object`-Array auch alle anderen Objekte aufnehmen.

Bauen wir uns eine statische Methode `set()`, die einfach ein Element an die erste Stelle ins Feld setzt:

Listing 5.78: ArrayCovariance.java, `set()`

```

public static void set( Object[] array, Object element )
{
    array[ 0 ] = element;
}
  
```

Die Kovarianz ist beim Lesen von Eigenschaften nicht problematisch, beim Schreiben jedoch potenziell gefährlich. Schauen wir, was mit unterschiedlichen Array- und Elementtypen passiert:

Listing 5.79: ArrayCovariance.java, main()

```
Object[] objectArray = new Object[ 1 ];
String[] stringArray = new String[ 1 ];
System.out.println( "It's time for change" instanceof Object ); // true
set( stringArray, "It's time for change" );
set( objectArray, "It's time for change" );
set( stringArray, new StringBuilder("It's time for change") ); // ☹
```

Der String lässt sich in einem String-Array abspeichern. Der zweite Aufruf funktioniert ebenfalls, denn ein String lässt sich auch in einem Object-Feld speichern, da ein Object ja ein Basistyp ist. Vor einem Dilemma stehen wir dann, wenn das Feld eine Referenz speichern soll, die nicht typkompatibel ist. Das zeigt der dritte set()-Aufruf: Zur Compilezeit ist alles noch in Ordnung, aber zur Laufzeit kommt es zu einer `ArrayStoreException`:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.StringBuilder
at ArrayCovariance.set(ArrayCovariance.java:5)
at ArrayCovariance.main(ArrayCovariance.java:19)
```

Das haben wir aber auch verdient, denn ein `StringBuilder`-Objekt lässt sich nicht in einem String-Feld speichern. Selbst ein `new Object()` hätte zu einem Problem geführt.

Das Typsystem von Java kann diese Spitzfindigkeit nicht zur Übersetzungszeit prüfen. Erst zur Laufzeit ist ein Test mit dem bitteren Ergebnis einer `ArrayStoreException` möglich. Bei Generics ist dies etwas anders, denn hier sind vergleichbare Konstrukte bei Vererbungsbeziehungen verboten.

5.11 Drum prüfe, wer sich ewig dynamisch bindet

Bei der Vererbung haben wir eine Form der Ist-eine-Art-von-Beziehung, sodass die Unterklassen immer auch vom Typ der Oberklassen sind. Die sichtbaren Methoden, die die Oberklassen besitzen, existieren somit auch in den Unterklassen. Der Vorteil bei der Spezialisierung ist, dass die Oberklasse eine einfache Implementierung vorgeben und eine Unterklasse diese überschreiben kann. Wir hatten das bisher bei `toString()` gese-

hen. Doch nicht nur die Spezialisierung ist aus Sicht des Designs interessant, sondern auch die Bedeutung der Vererbung. Bietet eine Oberklasse eine sichtbare Methode an, so wissen wir immer, dass alle Unterklassen diese Methode haben werden, egal, ob sie die Methode überschreiben oder nicht. Wir werden gleich sehen, dass dies zu einem der wichtigsten Konstrukte in objektorientierten Programmiersprachen führt.

5.11.1 Gebunden an `toString()`

Da jede Klasse Eigenschaften von `java.lang.Object` erbt, lässt sich auf jedem Objekt die `toString()`-Methode aufrufen. Sie soll in unseren Klassen `GameObject` und `Room` wie folgt implementiert sein:

Listing 5.80: com/tutego/insel/game/vf/GameObject.java, `GameObject`

```
public class GameObject
{
    public String name;

    @Override public String toString()
    {
        return String.format( "GameObject[name=%s]", name );
    }
}
```

Listing 5.81: com/tutego/insel/game/vf/Room.java, `Room`

```
public class Room extends GameObject
{
    public int size;

    @Override public String toString()
    {
        return String.format( "Room[name=%s, size=%d]", name, size );
    }
}
```

Die Unterklassen `GameObject` und `Room` überschreiben die `toString()`-Methode aus `Object`. Bei einem `toString()` auf einem `GameObject` kommt nur der Name in die `toString()`-Kennung, und bei einem `toString()` auf einem `Room`-Objekt kommen Name und Größe in die String-Repräsentation.

Es fehlen noch einige kleine Testzeilen, die drei Räume aufbauen. Alle rufen die `toString()`-Methoden auf den Räumen auf, wobei der Unterschied darin besteht, dass die verweisende Referenzvariable alle Typen von `Room` durchgeht: Ein `Room` ist ein `Room`, ein `Room` ist ein `GameObject`, und ein `Room` ist ein `Object`:

Listing 5.82: com/tutego/insel/game/vf/Playground.java, main()

```
Room rr = new Room();
rr.name = "Affenhausen";
rr.size = 7349944;
System.out.println( rr.toString() );

GameObject rg = new Room();
rg.name = "Affenhausen";
System.out.println( rg.toString() );

Object ro = new Room();
System.out.println( ro.toString() );
```

Jetzt ist die spannendste Frage in der gesamten Objektorientierung folgende: Was passiert bei dem Methodenaufruf `toString()`?

```
Room[name=Affenhausen, size=7349944]
Room[name=Affenhausen, size=0]
Room[name=null, size=0]
```

Die Ausgabe ist leicht zu verstehen, wenn wir berücksichtigen, dass der Compiler nicht die gleiche Weisheit besitzt wie die Laufzeitumgebung. Vom Compiler würden wir erwarten, dass er jeweils das `toString()` in `Room`, aber auch in `GameObject` (die Ausgabe wäre nur der Name) und `toString()` aus `Object` aufruft – dann wäre die Kennung die kryptische.

Doch führt die Laufzeitumgebung die Anweisungen aus, und nicht der Compiler. Da dem im Programmtext vereinbarten Variabtentyp nicht zu entnehmen ist, welche Implementierung der Methode `toString()` aufgerufen wird, sprechen wir von *später dynamischer Bindung*, kurz *dynamischer Bindung*. Erst zur Laufzeit (das ist spät, im Gegensatz zur Übersetzungszeit) wählt die Laufzeitumgebung dynamisch die entsprechende Objektmethode aus – passend zum tatsächlichen Typ des aufrufenden Objekts. Die virtuelle Maschine weiß, dass hinter den drei Variablen immer ein Raum-Objekt steht, und ruft daher das `toString()` vom `Room` auf.

Wichtig ist, dass eine Methode überschrieben wird; von einer gleichlautenden Methode in beiden Unterklassen `GameObject` und `Room` hätten wir nichts, da sie nicht in `Object` deklariert ist. Sonst hätten die Klassen nur rein »zufällig« diese Methode, aber die Ober- und Unterklassen verbindet nichts. Wir nutzen daher ausdrücklich die Gemeinsamkeit, dass `GameObject`, `Player` und weitere Unterklassen `toString()` aus `Object` erben. Ohne die Oberklasse gäbe es kein Bindeglied, und folglich bietet die Oberklasse immer eine Methode an, die Unterklassen überschreiben können. Würden wir eine neue Unterklasse von `Object` schaffen und `toString()` nicht überschreiben, so fände die Laufzeitumgebung `toString()` in `Object`, aber die Methode gäbe es auf jeden Fall; entweder die Original-Methode oder die überschriebene Variante.



Begrifflichkeit

Dynamische Bindung wird oft auch *Polymorphie* genannt; ein dynamisch gebundener Aufruf ist dann ein *polymorpher Aufruf*. Das ist im Kontext von Java in Ordnung, allerdings gibt es in der Welt der Programmiersprachen unterschiedliche Dinge, die »*Polymorphie*« genannt werden, etwa parametrische Polymorphie (in Java heißt das dann *Generics*), und die Theoretiker kennen noch viel mehr beängstigende Begriffe.

5.11.2 Implementierung von `System.out.println(Object)`

Werfen wir einen Blick auf ein Programm, das dynamisches Binden noch deutlicher macht. Die `print()`- und `println()`-Methoden sind so überladen, dass sie jedes beliebige Objekt annehmen und dann die String-Repräsentation ausgeben:

Listing 5.83: `java/io/PrintStream.java`, Skizze von `println()`

```
public void println( Object x )
{
    String s = (x == null) ? "null" : x.toString();
    print( s );
    newLine();
}
```

Die `println()`-Methode besteht aus drei Teilen: Als Erstes wird die String-Repräsentation eines Objekts erfragt – hier findet sich der dynamisch gebundene Aufruf –, dann wird dieser String an `print()` weitergegeben, und `newLine()` produziert abschließend den Zeilenumbruch.

Der Compiler hat überhaupt keine Ahnung, was `x` ist; es kann alles sein, denn alles ist ein `java.lang.Object`. Statisch lässt sich aus dem Argument `x` nichts ablesen, und so muss die Laufzeitumgebung entscheiden, an welche Klasse der Methodenaufruf geht. Das ist das Wunder der dynamischen Bindung.

Eclipse zeigt bei der Tastenkombination `Strg + T` eine Typhierarchie an, standardmäßig die Oberklassen und bekannten Unterklassen.



5.11.3 Nicht dynamisch gebunden bei privaten, statischen und finalen Methoden

Obwohl Methodenaufrufe eigentlich dynamisch gebunden sind, gibt es bei privaten, statischen und finalen Methoden eine Ausnahme. Das liegt daran, dass nur überschriebene Methoden an dynamischer Bindung teilnehmen, und wenn es kein Überschreiben gibt, dann gibt es auch keine dynamische Bindung. Und da weder private noch statische oder finale Methoden überschrieben werden können, sind Methodenaufrufe auch nicht dynamisch gebunden. Sehen wir uns das an einer privaten Methode an:

Listing 5.84: NoPolyWithPrivate.java

```
class NoPolyWithPrivate
{
    public static void main( String[] args )
    {
        Banana unsicht = new Banana();
        System.out.println( unsicht.bar() ); // 2
    }
}

class Fruit
{
    private int furcht()
    {
        return 2;
    }

    int bar()
    {
        return furcht();
```

```

        }
    }

class Banana extends Fruit
{
    // Überschreibt nicht, daher kein @Override
    public int furcht()
    {
        return 1;
    }
}

```

Der Compiler meldet bei der Methode `furcht()` in der UnterkLASSE keinen Fehler. Für den Compiler ist es in Ordnung, wenn es eine Methode in der UnterkLASSE gibt, die den gleichen Namen wie eine private Methode in der OberKLasse trägt. Das ist auch gut so, denn private Implementierungen sind ja ohnehin geheim und versteckt. Die UnterkLASSE soll von den privaten Methoden in der OberKLasse gar nichts wissen. Statt von *Überschreiben* sprechen wir hier von *Überdecken*.

Die Laufzeitumgebung macht etwas Erstaunliches für `unsicht.bar()`: Die Methode `bar()` wird aus der OberKLasse geerbt. Wir wissen, dass in `bar()` aufgerufene Methoden normalerweise dynamisch gebunden werden, das heißt, dass wir eigentlich bei `furcht()` in UnterkLASSEN landen müssten, da wir ein Objekt vom Typ `Banana` haben. Bei privaten Methoden ist das aber anders, da sie nicht vererbt werden. Wenn eine aufgerufene Methode den Modifizierer `private` trägt, wird nicht dynamisch gebunden, und `unsicht.bar()` bezieht sich bei `furcht()` auf die Methode aus `Fruit`.

```
System.out.println( unsicht.bar() ); // 2
```

Anders wäre es, wenn bei `furcht()` der Sichtbarkeitsmodifizierer `public` wäre; wir bekämen dann die Ausgabe 1.

Dass private und statische Methoden nicht überschrieben werden, ist ein wichtiger Beitrag zur Sicherheit. Falls nämlich Unterklassen interne private Methoden überschreiben könnten, wäre dies eine Verletzung der inneren Arbeitsweise der OberKLasse. In einem Satz: Private Methoden sind nicht in den Unterklassen sichtbar und werden daher nicht verdeckt oder überschrieben. Andernfalls könnten private Implementierungen im Nachhinein geändert werden, und OberKLassen wären nicht mehr sicher davor, dass tatsächlich ihre eigenen Methoden benutzt werden.

Schauen wir, was passiert, wenn wir in der Methode `bar()` über die `this`-Referenz auf ein Objekt vom Typ `Banana` casten:

```
int bar()
{
    return ((Banana)(this)).furcht();
}
```

Dann wird ausdrücklich diese `furcht()` aus `Banana` aufgerufen, was jedoch kein typisches objektorientiertes Konstrukt darstellt, da Oberklassen ihre Unterklassen im Allgemeinen nicht kennen. `bar()` in der Klasse `Ober` ist somit unnütz.

5.11.4 Dynamisch gebunden auch bei Konstruktoraufrufen *

Dass ein Konstruktor der Unterklasse zuerst den Konstruktor der Oberklasse aufruft, kann die Initialisierung der Variablen in der Unterklasse stören. Schauen wir uns erst Folgendes an:

```
class Bouncer extends Bodybuilder
{
    String who = "Ich bin ein Rausschmeißer";
}
```

Wo wird nun die Variable `who` initialisiert? Wir wissen, dass die Initialisierungen immer im Konstruktor vorgenommen werden, doch gibt es ja noch gleichzeitig ein `super()` im Konstruktor. Da die Spezifikation von Java Anweisungen vor `super()` verbietet, muss die Zuweisung hinter dem Aufruf der Oberklasse folgen. Das Problem ist nun, dass ein Konstruktor der Oberklasse früher aufgerufen wird, als Variablen in der Unterklasse initialisiert wurden. Wenn es die Oberklasse nun schafft, auf die Variablen der Unterklasse zugreifen, wird der erst später gesetzte Wert fehlen. Der Zugriff gelingt tatsächlich, doch nur durch einen Trick, da eine Oberklasse (etwa `Bodybuilder`) nicht auf die Variablen der Unterklasse zugreifen kann. Wir können aber in der Oberklasse genau jene Methode der Unterklasse aufrufen, die die Unterklasse aus der Oberklasse überschreibt. Da Methodenaufrufe dynamisch gebunden werden, kann eine Methode den Wert auslesen:

Listing 5.85: `Bouncer.java`

```
class Bodybuilder
{
    Bodybuilder()
```

```
{  
    whoAmI();  
}  
  
void whoAmI()  
{  
    System.out.println( "Ich weiß es noch nicht :-( );  
}  
}  
  
public class Bouncer extends Bodybuilder  
{  
    String who = "Ich bin ein Rausschmeißer";  
  
    @Override  
    void whoAmI()  
    {  
        System.out.println( who );  
    }  
  
    public static void main( String[] args )  
    {  
        Bodybuilder bb = new Bodybuilder();  
        bb.whoAmI();  
  
        Bouncer bouncer = new Bouncer();  
        bouncer.whoAmI();  
    }  
}
```

Die Ausgabe ist nun folgende:

```
Ich weiß es noch nicht :-(  
Ich weiß es noch nicht :-(  
null  
Ich bin ein Rausschmeißer
```

Das Besondere an diesem Programm ist die Tatsache, dass überschriebene Methoden – hier `whoAmI()` – dynamisch gebunden werden. Diese Bindung gibt es auch dann schon, wenn das Objekt noch nicht vollständig initialisiert wurde. Daher ruft der Konstruktor der Oberklasse `Bodybuilder` nicht `whoAmI()` von `Bodybuilder` auf, sondern `whoAmI()` von `Bouncer`. Wenn in diesem Beispiel ein `Bouncer`-Objekt erzeugt wird, dann ruft `Bouncer` mit `super()` den Konstruktor von `Bodybuilder` auf. Dieser ruft wiederum die Methode `whoAmI()` in `Bouncer` auf, und er findet dort keinen String, da dieser erst nach `super()` gesetzt wird. Schreiben wir den Konstruktor von `Bouncer` einmal ausdrücklich hin:

```
public class Bouncer extends Bodybuilder
{
    String who;

    Bouncer()
    {
        super();
        who = "Ich bin ein Rausschmeißer";
    }
}
```

Die Konsequenz, die sich daraus ergibt, ist folgende: Dynamisch gebundene Methodenaufrufe über die `this`-Referenz sind in Konstruktoren potenziell gefährlich und sollten deshalb vermieden werden. Vermeiden lässt sich das, indem der Konstruktor nur private (oder finale) Methoden aufruft, da diese nicht dynamisch gebunden werden. Wenn der Konstruktor eine private (finale) Methode in seiner Klasse aufruft, dann bleibt es auch dabei.

5.11.5 Eine letzte Spielerei mit Javas dynamischer Bindung und überschatteten Attributen *

Werfen wir einen Blick auf folgendes Java-Programm:

Listing 5.86: SubBoaster.java

```
class SuperBoaster
{
    int no = 1;
```

```

void boast()
{
    System.out.println( "Ich bin die Nummer " + no );
}

public class SubBoaster extends SuperBoaster
{
    int no = 2;

    @Override void boast()
    {
        super.boast();           // Ich bin die Nummer 1
        System.out.println( super.no ); // 1
        System.out.println( no );   // 2
    }

    public static void main( String[] args )
    {
        new SubBoaster().boast();
    }
}

```

Die Methode `boast()` aus `SubBoaster` ruft mit `super.boast()` die Methode der Oberklasse auf. Ein einfacher Aufruf von `boast()` in der Unterklasse würde in eine Rekursion führen. Die Unterklasse hat mit `super.nr` Zugriff auf die überschattete Objektvariable `nr` aus der Oberklasse. `super` ist wie `this` eine spezielle Referenz und kann auch genauso eingesetzt werden, nur dass `super` in den Namensraum der Oberklasse geht. Eine Aneinanderreihung von `super`-Schlüsselwörtern bei einer tieferen Vererbungshierarchie ist nicht möglich. Hinter einem `super` muss eine Objekteigenschaft stehen, und Anweisungen wie `super.super.nr` sind somit immer ungültig. Für Variablen gibt es eine Möglichkeit, die sich durch einen Cast in die Oberklasse ergibt. Setzen wir in `boast()` der Unterklasse folgende Anweisung:

```
System.out.println( ((SuperBoaster) this).nr );      // 1
```

Die Ausgabe 1 ist also identisch mit `System.out.println(super.nr)`. Die `this`-Referenz entspricht einem Objekt vom Typ `SubBoaster`. Wenn wir dies aber in den Typ `SuperBoaster`

konvertieren, bekommen wir genau das `nr` aus der Basisklasse unserer Hierarchie. Wir erkennen hier eine sehr wichtige Eigenschaft von Java, nämlich dass Variablen nicht dynamisch gebunden werden. Anders sieht es aus, wenn wir Folgendes in die Methode `boast()` der Unterkategorie `SubBoaster` setzen:

```
((SuperBoaster)this).boast();
```

Hier ruft die Laufzeitumgebung nicht `boast()` aus `SuperBoaster` auf, sondern die aktuelle Methode `boast()` aus `SubBoaster`, sodass wir in einer Rekursion landen. Der Grund dafür liegt in der dynamischen Bindung zur Laufzeit, die ein Compiler-Typecast nicht ändert.

5.12 Abstrakte Klassen und abstrakte Methoden

Nicht immer soll eine Klasse sofort ausprogrammiert werden, zum Beispiel dann nicht, wenn die Oberklasse lediglich Methoden für die Unterklassen vorgeben möchte, aber nicht weiß, wie sie diese implementieren soll. In Java gibt es dazu zwei Konzepte: *abstrakte Klassen* und *Schnittstellen* (engl. *interfaces*).

5.12.1 Abstrakte Klassen

Bisher haben wir Vererbung eingesetzt, und jede Klasse konnte Objekte bilden. Das Bilden von Exemplaren ist allerdings nicht immer sinnvoll, zum Beispiel soll es untersagt werden, wenn eine Klasse nur als Oberklasse in einer Vererbungshierarchie existieren soll. Sie kann dann als Modellierungsklasse eine Ist-eine-Art-von-Beziehung ausdrücken und Signaturen für die Unterklassen vorgeben. Eine Oberklasse besitzt dabei Vorgaben für die Unterklassen. Das heißt, alle Unterklassen erben die Methoden. Ein Exemplar der Oberklasse selbst muss nicht existieren.

Um dies in Java auszudrücken, setzen wir den Modifizierer `abstract` an die Typdeklaration der Oberklasse. Von dieser Klasse können dann keine Exemplare gebildet werden, und der Versuch einer Objekterzeugung führt zu einem Compilerfehler. Ansonsten verhalten sich die abstrakten Klassen wie normale, enthalten die gleichen Eigenschaften und können auch selbst von anderen Klassen erben. Abstrakte Klassen sind das Gegen teil von *konkreten Klassen*.

Wir wollen die Klasse `GameObject` als Oberklasse für die Spielgegenstände abstrakt machen, da Exemplare davon nicht existieren müssen:

Listing 5.87: com/tutego/insel/game/vh/GameObject.java, GameObject

```
public abstract class GameObject
{
    public String name;
}
```

Mit dieser abstrakten Klasse `GameObject` drücken wir aus, dass es eine allgemeine Klasse ist, zu der keine konkreten Objekte existieren. Es gibt in der realen Welt schließlich keine allgemeinen und unspezifizierten Spielgegenstände, sondern nur spezielle Unterarten, zum Beispiel Spieler, Schlüssel, Räume und so weiter. Es ergibt also keinen Sinn, ein Exemplar der Klasse `GameObject` zu bilden. Die Klasse soll nur in der Hierarchie auftauchen, um alle Spielobjekte zum Typ `GameObject` zu machen und ihnen einige Eigenschaften zu geben. Dies zeigt, dass Oberklassen allgemeiner gehalten sind und Unterklassen weiter spezialisieren.

**Abbildung 5.28:** In der UML werden die Namen abstrakter Klassen kursiv gesetzt.

Tipp

Abstrakte Klassen lassen sich auch nutzen, um zu verhindern, dass ein Exemplar der Klasse gebildet werden kann. Der Modifizierer `abstract` sollte aber dazu nicht eingesetzt werden. Besser ist es, die Sichtbarkeit des Konstruktors auf `private` oder `protected` zu setzen.

Basistyp abstrakte Klasse

Die abstrakten Klassen werden normalerweise in der Vererbung eingesetzt. Eine Klasse kann die abstrakte Klasse erweitern und dabei auch selbst wieder abstrakt sein. Auch gilt die Ist-eine-Art-von-Beziehung weiterhin, sodass sich schließlich Folgendes schreiben lässt:

Listing 5.88: com/tutego/insel/game/vh/Declarations.java, main()

```
GameObject    go1 = new Room();
GameObject    go2 = new Player();
GameObject[] gos = { new Player(), new Room() };
```



Hinweis

Die Deklaration `GameObject[] gos = { new Player(), new Room() }` ist die Kurzform für `GameObject[] gos = new GameObject[]{ new Player(), new Room() }`. Wenn im Programmcode `new GameObject[] {...}` steht, kennzeichnet das nur den Typ des Feldes. Das ist unabhängig davon, ob die Klasse `GameObject` abstrakt ist oder nicht.

5

5.12.2 Abstrakte Methoden

Der Modifizierer `abstract` vor dem Schlüsselwort `class` leitet die Deklaration einer abstrakten Klasse ein. Eine Klasse kann ebenso abstrakt sein wie eine Methode. Eine abstrakte Methode gibt lediglich die Signatur vor, und eine Unterklasse implementiert irgendwann diese Methode. Die Klasse ist somit für den Kopf der Methode zuständig, während die Implementierung an anderer Stelle erfolgt. Abstrakte Methoden drücken aus, dass die Oberklasse keine Ahnung von der Implementierung hat und dass sich die Unterklassen darum kümmern müssen.

Da eine abstrakte Klasse abstrakte Methoden enthalten kann, aber nicht enthalten muss, unterscheiden wir:

- *Reine (pure) abstrakte Klassen:* Die abstrakte Klasse enthält ausschließlich abstrakte Methoden.
- *Partiell abstrakte Klassen:* Die Klasse ist abstrakt, enthält aber auch konkrete Implementierungen, also nicht abstrakte Methoden. Das bietet den Unterklassen ein Gerüst, das sie nutzen können.



Definition

Hat eine pure abstrakte Klasse nur eine Methode, so sprechen wir von einer *SAM* (für *Single Abstract Method*).

Mit Spielobjekten muss sich spielen lassen

Damit wir mit den Spielobjekten wie *Tür*, *Schlüssel*, *Raum* und *Spieler* wirklich spielen können, sollen die Objekte aufeinander angewendet werden können. Das Ziel unseres Programms ist es, Sätze abzubilden, die aus Subjekt, Verb und Objekt bestehen:

- Schlüssel öffnet Tür.
- Spieler nimmt Bier.

- Pinsel kitzelt Spieler.
- Radio spielt Musik.

Die Programmversion soll etwas einfacher sein und statt unterschiedlicher Aktionen (öffnen, nehmen, ...) nur »nutzen« kennen.

Zur Umsetzung dieser Aufgabe bekommen die Spielklassen wie *Schlüssel*, *Spieler*, *Pinsel*, *Radio* eine spezielle Methode, die ein anderes Spielobjekt nimmt und testet, ob sie aufeinander angewendet werden können. Ein Schlüssel öffnet eine Tür, aber »öffnet« keine Musik. Demnach kann ein Musik-Objekt nicht auf einem Tür-Objekt angewendet werden, wohl aber ein Schlüssel-Objekt. Ist die Anwendung möglich, kann die Methode weitere Aktionen ausführen, etwa Zustände setzen. Denn wenn der Schlüssel auf der Tür gültig ist, ist die Tür danach offen. Ob eine Operation möglich war oder nicht, soll eine Rückgabe aussagen.

Eine Methode in `GameObject` könnte somit folgende Signatur besitzen:

```
boolean useOn( GameObject object )
```

Die Operation `useOn()` soll auf dem eigenen Objekt das an die Methode übergebene Objekt nutzen. Implementiert die Schlüssel-Klasse `useOn()`, so kann sie testen, ob das `GameObject` eine Tür ist, und außerdem kann sie prüfen, ob Schlüssel und Tür zusammenpassen. Wenn ja, kann der Schlüssel die Tür öffnen, und die Rückgabe ist `true`, sonst `false`.

Da jede Spielobjekt-Klasse die Operation `useOn()` implementieren muss, soll die Basisklasse sie abstrakt deklarieren, denn eine abstrakte Methode fordert von den Unterklassen eine Implementierung ein, sonst ließen sich keine Exemplare bilden:

Listing 5.89: com/tutego/insel/game/vi/GameObject.java, `GameObject`

```
public abstract class GameObject
{
    public String name;
    public abstract boolean useOn( GameObject object );
}
```

Die Klasse `GameObject` deklariert eine abstrakte Methode, und da sie immer ohne Implementierung ist, steht statt des Methodenrumpfs ein Semikolon. Ist mindestens eine Methode abstrakt, so ist es automatisch die ganze Klasse. Deshalb müssen wir das Schlüsselwort `abstract` ausdrücklich vor den Klassennamen schreiben. Vergessen wir das Schlüsselwort `abstract` bei einer solchen Klasse, erhalten wir einen Compilerfehler.

Eine Klasse mit einer abstrakten Methode muss abstrakt sein, da sonst irgendjemand ein Exemplar konstruieren und genau diese Methode aufrufen könnte. Versuchen wir, ein Exemplar einer abstrakten Klasse zu erzeugen, so bekommen wir ebenfalls einen Compilerfehler. Natürlich kann eine abstrakte Klasse nicht-abstrakte Eigenschaften haben, so wie es `GameObject` mit dem Attribut `name` zeigt. Konkrete Methoden sind auch erlaubt, die brauchen wir jedoch hier nicht. Eine `toString()`-Methode wäre vielleicht noch interessant, sie könnte dann auf `name` zurückgreifen.

Vererben von abstrakten Methoden

Wenn wir von einer Klasse abstrakte Methoden erben, so haben wir zwei Möglichkeiten:

- Wir überschreiben alle abstrakten Methoden und implementieren sie. Dann muss die Unterklasse nicht mehr abstrakt sein (wobei sie es auch weiterhin sein kann). Von der Unterklasse kann es ganz normale Exemplare geben.
- Wir überschreiben die abstrakte Methode nicht, sodass sie normal vererbt wird. Das bedeutet: Eine abstrakte Methode bleibt in unserer Klasse, und die Klasse muss wiederum abstrakt sein.

Die Unterklasse `Door` soll die abstrakte Methode `useOn()` überschreiben, aber immer `false` zurückgeben, da sich eine Tür in unserem Szenario auf nichts anwenden lässt. Nach dem Implementieren der abstrakten Methode sind Exemplare von Türen möglich:

Listing 5.90: com/tutego/insel/game/vi/Door.java, Door

```
public class Door extends GameObject
{
    int      id;
    boolean isOpen;

    @Override public boolean useOn( GameObject object )
    {
        return false;
    }
}
```

Eine Tür hat für den Schlüssel eine ID, denn nicht jeder Schlüssel passt auf jedes Schloss. Weiterhin hat die Tür einen Zustand: Sie kann offen oder geschlossen sein.

Die dritte Klasse Key speichert ebenfalls eine ID und implementiert `useOn()`:

Listing 5.91: com/tutego/insel/game/vi/Key.java, Key

```
public class Key extends GameObject
{
    int id;

    public @Override boolean useOn( GameObject object )
    {
        if ( object instanceof Door )
            if ( id == ((Door) object).id )
                return ((Door) object).isOpen = true;

        return false;
    }
}
```

Die Realisierung ist etwas komplexer. Als Erstes prüft die Methode mit `instanceof`, ob der Schlüssel auf eine Tür angewendet wird. Wenn ja, muss die ID von Schlüssel und Tür stimmen. Ist auch dieser Vergleich wahr, kann `isOpen` wahr werden, und die Methode liefert `true`.

Im Testprogramm wollen wir zwei Schlüssel auf eine Tür anwenden. Nur der Schlüssel mit der passenden ID öffnet die Tür. Eine Tür kann nicht auf einen Schlüssel angewendet werden, denn die Implementierungen sind nicht symmetrisch ausgelegt:

Listing 5.92: com/tutego/insel/game/vi/Playground.java, main()

```
Door door = new Door();
door.id = 12;

Key key1 = new Key();
key1.id = 99;

Key key2 = new Key();
key2.id = 12;

System.out.printf( "erfolgreich=%b%n", key1.useOn(door) );
```

```
System.out.printf( "erfolgreich=%b, isOpen=%b%n", key2.useOn(door), door.isOpen );  
System.out.printf( "erfolgreich=%b%n", door.useOn(key1) );
```

Die Ausgaben sind:

```
erfolgreich=false  
erfolgreich=true, isOpen=true  
erfolgreich=false
```

Implementiert eine Klasse nicht alle geerbten abstrakten Methoden, so muss die Klasse selbst wieder abstrakt sein. Ist unsere Unterkelas einer abstrakten Basisklasse nicht abstrakt, so bietet Eclipse mit **Strg** + **1** an, entweder die eigene Klasse abstrakt zu machen oder alle geerbten abstrakten Methoden mit einem Dummy-Rumpf zu implementieren.



5.13 Schnittstellen

Da Java nur Einfachvererbung kennt, ist es schwierig, Klassen mehrere Typen zu geben. Das kann immer nur in einer Reihe geschehen, also etwa so: `GameObject` erbt von `Object`, `Building` erbt von `GameObject`, `Castle` erbt von `Building` usw. Es wird schwierig, an einer Stelle zu sagen, dass ein `Building` ein `GameObject` ist, aber zum Beispiel noch zusätzlich einen Typ `Preis` haben soll, was nur nicht gleich alle Spielobjekte haben sollen. Denn soll eine Klasse auf einer Ebene von mehreren Typen erben, geht das durch die Einfachvererbung nicht. Da es aber möglich sein soll, dass in der objektorientierten Modellierung eine Klasse mehrere Typen in einem Schritt besitzt, gibt es das Konzept der *Schnittstelle* (engl. *interface*). Eine Klasse kann dann neben der Oberklasse eine beliebige Anzahl Schnittstellen implementieren und auf diese Weise weitere Typen sammeln.

OOP-Design

Schnittstellen sind eine gute Ergänzung zu abstrakten Klassen/Methoden. Denn im objektorientierten Design wollen wir das Was vom Wie trennen. Abstrakte Methoden sagen wie Schnittstellen etwas über das Was aus, aber erst die konkreten Implementierungen realisieren das Wie.



5.13.1 Schnittstellen deklarieren

Eine Schnittstelle enthält keine Implementierungen, sondern deklariert nur den Kopf einer Methode – also Modifizierer, den Rückgabetyp und die Signatur – ohne Rumpf.

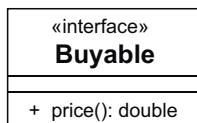


Abbildung 5.29: UML-Diagramm der Schnittstelle Buyable

Sollen in einem Spiel gewisse Dinge käuflich sein, haben sie einen Preis. Eine Schnittstelle `Buyable` soll allen Klassen die Methode `price()` vorschreiben.

Listing 5.93: com/tutego/insel/game/vk/Buyable.java, Buyable

```

interface Buyable
{
    double price();
}
  
```

Die Deklaration einer Schnittstelle erinnert an eine abstrakte Klasse mit abstrakten Methoden, nur steht anstelle von `class` das Schlüsselwort `interface`. Da alle Methoden in Schnittstellen automatisch abstrakt und öffentlich sind, akzeptiert der Compiler das redundante `abstract` und `public`, doch die Modifizierer sollten nicht geschrieben werden. Die von den Schnittstellen deklarierten Operationen sind – wie auch bei abstrakten Methoden – mit einem Semikolon abgeschlossen und haben niemals eine Implementierung.

Eine Schnittstelle darf keinen Konstruktor deklarieren. Das ist auch klar, da Exemplare von Schnittstellen nicht erzeugt werden können, sondern nur von den konkreten implementierenden Klassen.



Hinweis

Der Name einer Schnittstelle endet oft auf `-ble` (`Accessible`, `Adjustable`, `Runnable`). Er beginnt üblicherweise nicht mit einem Präfix wie »`I`«, obwohl die Eclipse-Entwickler diese Namenskonvention nutzen.

Obwohl in einer Schnittstelle keine Methoden ausprogrammiert werden und keine Objektvariablen deklariert werden dürfen, sind `static final`-Variablen (benannte Konstanten) in einer Schnittstelle erlaubt, statische Methoden jedoch nicht.



Existiert eine Klasse, in der Methoden in einer neuen Schnittstelle deklariert werden sollen, lässt sich REFACTOR • EXTRACT INTERFACE... einsetzen. Es folgt ein Dialog, der uns Methoden auswählen lässt, die später in der neuen Schnittstelle deklariert werden. Eclipse legt die Schnittstelle automatisch an und lässt die Klasse die Schnittstelle implementieren. Dort, wo es möglich ist, erlaubt Eclipse, dass die konkrete Klasse durch die Schnittstelle ersetzt wird.

5

5.13.2 Implementieren von Schnittstellen

Möchte eine Klasse eine Schnittstelle verwenden, so folgt hinter dem Klassennamen das Schlüsselwort `implements` und dann der Name der Schnittstelle. Die Ausdrucksweise ist dann: »Klassen werden vererbt und Schnittstellen implementiert.«

Für unsere Spielwelt sollen die Klassen `Chocolate` und `Magazine` die Schnittstelle `Buyable` implementieren. Eine Schokolade soll dabei immer einen sozialistischen Einheitspreis von 0,69 haben.

Listing 5.94: com/tutego/insel/game/vk/Chocolate.java, Chocolate

```
public class Chocolate implements Buyable
{
    @Override public double price()
    {
        return 0.69;
    }
}
```

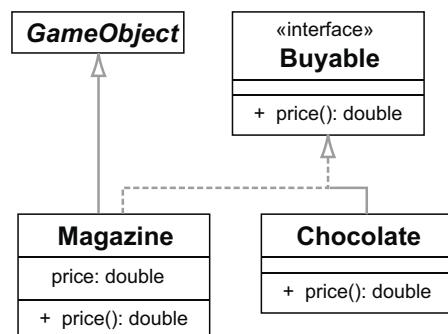


Abbildung 5.30: Die Klassen Magazine und Chocolate implementieren Buyable

Die Annotation `@Override` zeigt wieder eine überschriebene Methode (hier implementierte Methode einer Schnittstelle) an. Unter Java 5 führte `@Override` an implementierten Methoden einer Schnittstelle noch zu einem Compilerfehler.

Während `Chocolate` nur die Schnittstelle `Buyable` implementiert, soll `Magazine` zusätzlich ein `GameObject` sein:

Listing 5.95: com/tutego/insel/game/vk/Magazine.java, Magazine

```
public class Magazine extends GameObject implements Buyable
{
    double price;

    @Override public double price()
    {
        return price;
    }
}
```

Es ist also kein Problem – und bei uns so gewünscht –, wenn eine Klasse eine andere Klasse erweitert und zusätzlich Operationen aus Schnittstellen implementiert.



Hinweis

Da die in Schnittstellen deklarierten Operationen immer `public` sind, müssen auch die implementierten Methoden in den Klassen immer öffentlich sein. Sollte diese Vorgabe wirklich lästig sein, lässt sich immer noch eine abstrakte Klasse mit einer abstrakten Methode eingeschränkter Sichtbarkeit deklarieren. Dann gibt es aber auch nur einmal eine Vererbung.

Implementiert eine Klasse nicht alle Operationen aus den Schnittstellen, so erbt sie damit abstrakte Methoden und muss selbst wieder als abstrakt gekennzeichnet werden.



Tipp

Schnittstellen später zu ändern, wenn schon viele Klassen die Schnittstelle implementieren, ist eine schlechte Idee. Denn erneuert sich die Schnittstelle, etwa wenn nur eine Operation hinzukommt oder sich ein Variabtentyp ändert, dann sind plötzlich alle implementierenden Klassen kaputt. Sun selbst hat diesen Fehler bei der Schnittstelle `java.sql.Connection` gemacht.

Tipp (Forts.)

Beim Übergang von Java 5 auf Java 6 wurde die Schnittstelle erweitert, und keine Treiberimplementierungen konnten mehr kompiliert werden. Die übliche Lösung für das Problem ist, eine neue Schnittstelle mit weiteren Operationen einzuführen, die die alte Schnittstelle erweitert, aber auf 2 endet. `java.awt.LayoutManager2` ist so ein Beispiel aus dem Bereich der grafischen Oberflächen, `Attributes2`, `EntityResolver2`, `Locator2` für XML-Verarbeitung sind weitere.



Eclipse zeigt bei der Tastenkombination `Strg + T` eine Typ hierarchie an, Oberklassen stehen oben und Unterklassen unten. Wird in dieser Ansicht erneut `Strg + T` gedrückt, wird die Ansicht umgedreht, dann stehen die Obertypen unten, was den Vorteil hat, dass auch die implementierte Schnittstelle unter den Obertypen ist.



5.13.3 Markierungsschnittstellen *

Auch Schnittstellen ohne Methoden sind möglich. Diese leeren Schnittstellen werden *Markierungsschnittstellen* (engl. *marker interfaces*) genannt. Sie sind nützlich, da mit `instanceof` leicht überprüft werden kann, ob sie einen gewollten Typ einnehmen.

Die Java-Bibliothek bringt einige Markierungsschnittstellen schon mit, etwa:

- `java.util.RandomAccess`: Eine Datenstruktur bietet schnellen Zugriff über einen Index.
- `java.rmi.Remote`: Identifiziert Schnittstellen, deren Operationen von außen aufgerufen werden können.
- `java.lang.Cloneable`: Sorgt dafür, dass die `clone()`-Methode von `Object` aufgerufen werden kann.
- `java.util.EventListener`: Typ, den jeder Horcher in der Java SE implementiert.
- `java.io.Serializable`: Zustände eines Objekts lassen sich in einen Datenstrom schreiben – mehr dazu folgt in Kapitel 15, »Einführung in Dateien und Datenströme«.

Hinweis

Seit es das Sprachmittel der Annotationen gibt, sind Markierungsschnittstellen bei neuen Bibliotheken nicht mehr anzutreffen.

5.13.4 Ein Polymorphie-Beispiel mit Schnittstellen

Obwohl Schnittstellen auf den ersten Blick nichts »bringen« – Programmierer wollen gerne etwas vererbt bekommen, damit sie Implementierungsarbeit sparen können –, sind sie eine enorm wichtige Erfindung, da sich über Schnittstellen ganz unterschiedliche Sichten auf ein Objekt beschreiben lassen. Jede Schnittstelle ermöglicht eine neue Sicht auf das Objekt, eine Art Rolle. Implementiert eine Klasse diverse Schnittstellen, können ihre Exemplare in verschiedenen Rollen auftreten. Hier wird erneut das Substitutionsprinzip wichtig, bei dem ein mächtigeres Objekt verwendet wird, obwohl je nach Kontext nur die Methode der Schnittstellen erwartet wird.

Mit Magazine und Chocolate haben wir zwei Klassen, die Buyable implementieren. Damit existieren zwei Klassen, die einen gemeinsamen Typ und beide eine gemeinsame Methode price() besitzen.

```
Buyable b1 = new Magazine();
Buyable b2 = new Chocolate();
System.out.println( b1.price() );
System.out.println( b2.price() );
```

Für Buyable wollen wir eine statische Methode calculateSum() schreiben, die den Preis einer Sammlung zum Verkauf stehender Objekte berechnet. Sie soll wie folgt aufgerufen werden:

Listing 5.96: com/tutego/insel/game/vk/Playground.java, main()

```
Magazine madMag = new Magazine();
madMag.price = 2.50;
Buyable schoki = new Chocolate();
Magazine maxim = new Magazine();
maxim.price = 3.00;
System.out.printf( "%.2f", PriceUtils.calculateSum( madMag, maxim, schoki ) ); // 6,19
```

Damit calculateSum() eine beliebige Anzahl Argumente, aber mindestens eins, annehmen kann, realisieren wir die Methode mit einem Vararg:

Listing 5.97: com/tutego/insel/game/vk/PriceUtils.java, calculateSum()

```
static double calculateSum( Buyable price1, Buyable... prices )
{
    double result = price1.price();
```

```
for ( Buyable price : prices )  
    result += price.price();  
  
return result;  
}
```

Die Methode nimmt käufliche Dinge an, wobei es ihr völlig egal ist, um welche es sich dabei handelt. Was zählt, ist die Tatsache, dass die Elemente die Schnittstelle `Buyable` implementieren.

Die dynamische Bindung tritt schon in der ersten Anweisung `price1.price()` auf. Auch später rufen wir auf jedem Objekt, das `Buyable` implementiert, die Methode `price()` auf. Indem wir die unterschiedlichen Werte summieren, bekommen wir den Gesamtpreis der Elemente aus der Parameterliste.

Tipp

Wie schon erwähnt, sollte der Typ einer Variablen immer der kleinste nötige sein. Dabei sind Schnittstellen als Variablentypen nicht ausgenommen. Entwickler, die alle ihre Variablen vom Typ einer Schnittstelle deklarieren, wenden das Konzept *Programmieren gegen Schnittstellen* an. Sie binden sich also nicht an eine spezielle Implementierung, sondern an einen Basistyp.

Im Zusammenhang mit Schnittstellen bleibt zusammenfassend zu sagen, dass hier bei Methodenaufrufen dynamisches Binden pur auftaucht.

5.13.5 Die Mehrfachvererbung bei Schnittstellen *

Bei Klassen gibt es die Einschränkung, dass nur von einer direkten Oberklasse abgeleitet werden darf – egal, ob sie abstrakt ist oder nicht. Der Grund ist, dass Mehrfachvererbung zu dem Problem führen kann, dass eine Klasse von zwei Oberklassen die gleiche Methode erbt und dann nicht weiß, welche sie aufnehmen soll. Ohne Schwierigkeiten kann eine Klasse jedoch mehrere Schnittstellen implementieren. Das liegt daran, dass von einer Schnittstelle kein Code kommt, sondern nur eine Vorschrift zur Implementierung – im schlimmsten Fall gibt es die Vorschrift, eine Operation umzusetzen, mehrfach.

Dass in Java eine Klasse mehrere Schnittstellen implementieren kann, wird gelegentlich als *Mehrfachvererbung in Java* bezeichnet. Auf diese Weise besitzt die Klasse ganz un-

terschiedliche Typen. Ist `U` eine solche Klasse mit der Oberklasse `O` und implementiert sie die Schnittstellen `I1` und `I2`, so liefert für ein Exemplar `o` vom Typ `O` der Test `o instanceof O` ein wahres Ergebnis genauso wie `o instanceof I1` und `o instanceof I2`.



Begrifflichkeit

Wenn es um das Thema Mehrfachvererbung geht, dann müssen wir Folgendes unterscheiden: Geht es um *Klassenvererbung*, sogenannte *Implementierungsvererbung*, ist Mehrfachvererbung nicht erlaubt. Geht es dagegen um *Schnittstellenvererbung*, so ist in dem Sinne Mehrfachvererbung erlaubt, denn eine Klasse kann beliebig viele Schnittstellen implementieren. *Typ-Vererbung* ist hier ein gebräuchliches Wort. Üblicherweise wird der Begriff *Mehrfachvererbung* in Java nicht verwendet, da er sich traditionell auf Klassenvererbung bezieht.

Beginnen wir mit einem Beispiel. `GameObject` soll die Markierungsschnittstelle `Serializable` implementieren, sodass dann alle Unterklassen von `GameObject` ebenfalls vom Typ `Serializable` sind. Die Markierungsschnittstelle schreibt nichts vor, daher gibt es keine spezielle überschriebene Methode:

Listing 5.98: com/tutego/insel/game/v1/GameObject.java, GameObject

```
public abstract class GameObject implements Serializable
{
    protected String name;

    protected GameObject( String name )
    {
        this.name = name;
    }
}
```

Damit gibt es schon verschiedene Ist-eine-Art-von-Beziehungen: `GameObject` ist ein `java.lang.Object`, `GameObject` ist ein `GameObject`, `GameObject` ist `Serializable`.

Ein `Magazine` soll zunächst ein `GameObject` sein. Dann soll es nicht nur die Schnittstelle `Buyable` und damit die Methode `price()` implementieren, sondern sich auch mit anderen Magazinen vergleichen lassen. Dazu gibt es schon eine passende Schnittstelle in der Java-Bibliothek: `java.lang.Comparable`. Die Schnittstelle `Comparable` fordert, dass unser Magazin die Methode `int compareTo(Magazine)` implementiert. Der Rückgabewert der Methode zeigt an, wie das eigene Magazin zum anderen aufgestellt ist. Wir wollen defi-

nieren, dass das günstigere Magazin vor einem teureren steht (eigentlich sollten mit Comparable auch equals() und hashCode() aus Object überschrieben werden, doch das spart das Beispiel aus¹⁸⁾):

Listing 5.99: com/tutego/insel/game/v1/Buyable.java, Buyable

```
interface Buyable
{
    double price();
}
```

Listing 5.100: com/tutego/insel/game/v1/Magazine.java, Magazine

```
public class Magazine extends GameObject implements Buyable, Comparable<Magazine>
{
    private double price;

    public Magazine( String name, double price )
    {
        super( name );
        this.price = price;
    }

    @Override public double price()
    {
        return price;
    }

    @Override public int compareTo( Magazine that )
    {
        return Double.compare( this.price(), that.price() );
    }
}
```

¹⁸ Wenn compareTo() bei zwei gleichen Objekten 0 ergibt, so sollte equals() auch true liefern. Doch wird equals() nicht überschrieben, so führt die in Object implementierte Methode nur einen Referenzvergleich durch. Bei zwei im Prinzip gleichen Objekten würde die equals()-Standardimplementierung also false liefern. Bei hashCode() gilt das Gleiche: Zwei gleiche Objekte müssen auch den gleichen Hashwert haben. Ohne Überschreiben der Methode ist das jedoch nicht gegeben; nur zwei identische Objekte haben den gleichen Hashcode.

```

@Override public String toString()
{
    return name + " " + price;
}
}

```

Die Implementierung nutzt Generics mit Comparable<Magazine>, was wir genauer erst später lernen, aber an der Stelle schon einmal nutzen wollen. Der Hintergrund ist, dass Comparable dann genau weiß, mit welchem anderen Typ der Vergleich stattfinden soll.

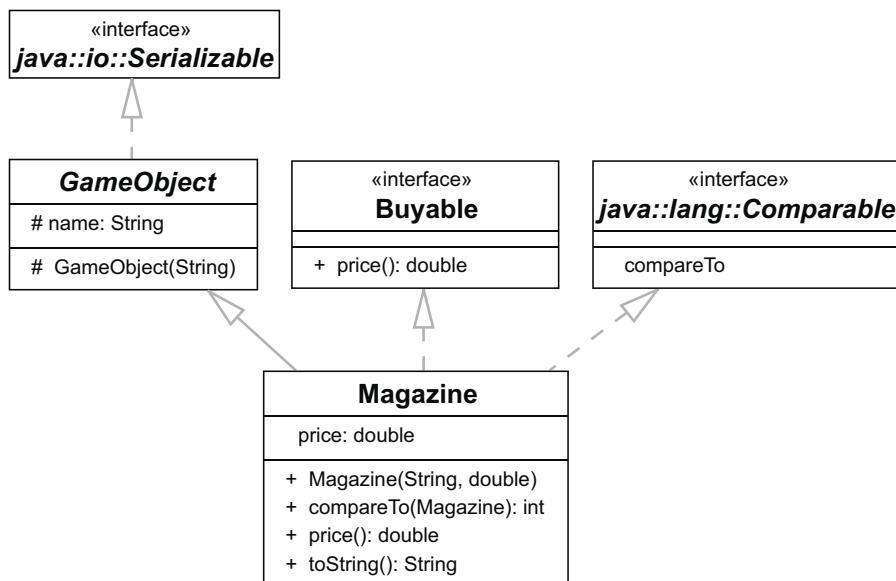


Abbildung 5.31: Die Klasse Magazine mit diversen Obertypen

Durch diese »Mehrfachvererbung« bekommt Magazine mehrere Typen, sodass sich je nach Sichtweise Folgendes schreiben lässt:

```

Magazine          m1 = new Magazine( "Mad Magazine", 2.50 );
GameObject        m2 = new Magazine( "Mad Magazine", 2.50 );
Object            m3 = new Magazine( "Mad Magazine", 2.50 );
Buyable           m4 = new Magazine( "Mad Magazine", 2.50 );
Comparable<Magazine> m5 = new Magazine( "Mad Magazine", 2.50 );
Serializable       m6 = new Magazine( "Mad Magazine", 2.50 );

```

Die Konsequenzen davon sind:

- Im Fall `m1` sind alle Methoden der Schnittstellen verfügbar, also `price()` und `compareTo()` sowie das Attribut `name`.
- Über `m2` ist keine Schnittstellenmethode verfügbar, und nur die geschützte Variable `name` ist vorhanden.
- Mit `m3` sind alle Bezüge zu Spielobjekten verloren. Aber ein `Magazine` als `Object` ist ein gültiger Argumenttyp für `System.out.println(Object)`.
- Die Variable `m4` ist vom Typ `Buyable`, sodass es `price()` gibt, jedoch kein `compareTo()`. Das Objekt könnte daher in `PriceUtils.calculateSum()` eingesetzt werden.
- Mit `m5` gibt es ein `compareTo()`, aber keinen Preis.
- Da `Magazine` die Klasse `GameObject` erweitert und darüber auch vom Typ `Serializable` ist, lässt sich keine besondere Methode aufrufen – `Serializable` ist eine Markierungsschnittstelle ohne Operationen. Damit könnte das Objekt allerdings von speziellen Klassen der Java-Bibliothek serialisiert und so persistent gemacht werden.

Ein kleines Beispiel zeigt abschließend die Anwendung der Methoden `compareTo()` der Schnittstelle `Comparable` und `price()` der Schnittstelle `Buyable`:

Listing 5.101: com/tutego/insel/game/v1/Playground.java, main() – Teil 1

```
Magazine spiegel = new Magazine( "Spiegel", 3.50 );
Magazine madMag = new Magazine( "Mad Magazine", 2.50 );
Magazine maxim = new Magazine( "Maxim", 3.00 );
Magazine neon = new Magazine( "Neon", 3.00 );
Magazine ct = new Magazine( "c't", 3.30 );
```

Da wir einem Magazin so viele Sichten gegeben haben, können wir es natürlich mit unserer früheren Methode `calculateSum()` aufrufen, da jedes `Magazine` ja `Buyable` ist:

Listing 5.102: com/tutego/insel/game/v1/Playground.java, main() – Teil 2

```
System.out.println( PriceUtils.calculateSum( spiegel, madMag, ct ) ); // 9.3
```

Und die Magazine können wir vergleichen:

Listing 5.103: com/tutego/insel/game/v1/Playground.java, main() – Teil 3

```
System.out.println( spiegel.compareTo( ct ) ); // 1
System.out.println( ct.compareTo( spiegel ) ); // -1
System.out.println( maxim.compareTo( neon ) ); // 0
```

So wie es der Methode `calculateSum()` egal ist, was für `Buyable`-Objekte konkret übergeben werden, so gibt es auch für `Comparable` einen sehr nützlichen Anwendungsfall: das Sortieren. Einem Sortierverfahren ist es egal, was für Objekte genau es sortiert, solange die Objekte sagen, ob sie vor oder hinter einem anderen Objekt liegen:

Listing 5.104: com/tutego/insel/game/v1/Playground.java, main() – Teil 4

```
Magazine[] mags = { spiegel, madMag, maxim, neon, ct };
Arrays.sort( mags );
System.out.println( Arrays.toString( mags ) );
// [Mad Magazine 2.5, Maxim 3.0, Neon 3.0, c't 3.3, Spiegel 3.5]
```

Die statische Methode `Arrays.sort()` erwartet ein Feld, dessen Elemente `Comparable` sind. Der Sortieralgorithmus macht Vergleiche über `compareTo()`, muss aber sonst über die Objekte nichts wissen. Unsere Magazine mit den unterschiedlichen Typen können also sehr flexibel in unterschiedlichen Kontexten eingesetzt werden. Es muss somit für das Sortieren keine Spezialsortiermethode geschrieben werden, die nur Magazine sortieren kann, oder eine Methode zur Berechnung einer Summe, die nur auf Magazinen arbeitet. Wir modellieren die unterschiedlichen Anwendungsszenarien mit jeweils unterschiedlichen Schnittstellen, die Unterschiedliches von dem Objekt erwarten.

5.13.6 Keine Kollisionsgefahr bei Mehrfachvererbung *

Bei der Mehrfachvererbung von Klassen besteht die Gefahr, dass zwei Oberklassen die gleiche Methode mit zwei unterschiedlichen Implementierungen den Unterklassen vererben. Die Unterkasse wüsste dann nicht, welche Logik sie erbt. Bei den Schnittstellen gibt es das Problem nicht, denn auch wenn zwei implementierende Schnittstellen die gleiche Methode vorschreiben würden, gäbe es keine zwei verschiedenen Implementierungen von Anwendungslogik. Die implementierende Klasse bekommt sozusagen zweimal die Aufforderung, die Operation zu implementieren. So wie bei folgendem Beispiel: Ein Politiker muss verschiedene Dinge vereinen; er muss sympathisch sein, aber auch durchsetzungsfähig handeln können.

Listing 5.105: Politician.java

```
interface Likeable
{
    void act();
```

```

interface Assertive
{
    void act();
}

public class Politician implements Likeable, Assertive
{
    @Override public void act()
    {
        // Implementation
    }
}

```

Zwei Schnittstellen schreiben die gleiche Operation vor. Eine Klasse implementiert diese beiden Schnittstellen und muss beiden Vorgaben gerecht werden.

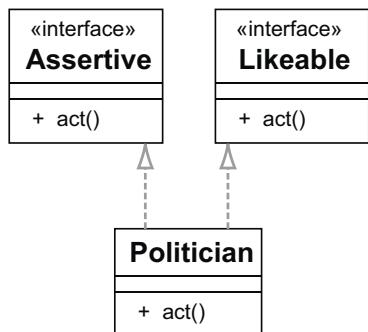


Abbildung 5.32: Eine Klasse erbt von zwei Schnittstellen die gleiche Operation.

5.13.7 Erweitern von Interfaces – Subinterfaces

Ein *Subinterface* ist die Erweiterung eines anderen Interfaces. Diese Erweiterung erfolgt – wie bei der Vererbung – durch das Schlüsselwort `extends`.

```

interface Disgusting
{
    double disgustingValue();
}

```

```
interface Stinky extends Disgusting
{
    double olf();
}
```

Die Schnittstelle modelliert Stinkiges, was besonders abstoßend ist. Zusätzlich soll die Stinkquelle die Stärke der Stinkigkeit in der Einheit Olf angeben. Eine Klasse, die nun Stinky implementiert, muss die Methoden aus beiden Schnittstellen implementieren, demnach die Methode `disgustingValue()` aus `Disgusting` sowie die Operation `olf()`, die in `Stinky` selbst angegeben wurde. Ohne die Implementierung beider Methoden wird eine implementierende Klasse abstrakt sein müssen.



Hinweis

Eine interessante Änderung an der API gab es in Java 5 mit dem Einsatz von `Iterable`. Die Schnittstelle `Collection` erweitert seit Java 5 die Schnittstelle `Iterable`. Nun ist es immer so, dass nachträgliche neue Schnittstellen neue Methoden erzwingen und alle alten Implementierungen ungültig machen können. In diesem Fall war das aber kein Problem, da `Iterable` die Operation `iterator()` vorschreibt, die `Collection` sowieso schon deklarierte. Hätte `Iterable` eine neue Operation eingeführt, hätte das zu einem großen Bruch existierender Programme geführt.

5.13.8 Konstantendeklarationen bei Schnittstellen

Schnittstellen können Attribute besitzen, die jedoch immer automatisch statisch und final, also Konstanten sind.



Beispiele

Die Schnittstelle `Buyable` soll eine Konstante für einen Maximalpreis deklarieren:

```
interface Buyable
{
    int MAX_PRICE = 10000000;
    double price();
}
```

**Tipp**

Da alle Attribute einer Schnittstelle immer implizit public static final sind, ergibt sich ein Problem, wenn das Attribut ein veränderbares Objekt repräsentiert, wie in folgendem Beispiel ein StringBuilder-Objekt:

```
interface Vulcano
{
    StringBuilder EYJAFJALLAJÖKULL = new StringBuilder( "Eyjafjallajökull" );
}
```

Da EYJAFJALLAJÖKULL eine öffentliche StringBuilder-Variable ist, kann sie leicht mit Vulcano.EYJAFJALLAJÖKULL.replace(0, Vulcano.EYJAFJALLAJÖKULL.length(), "Vesuvius"); verändert werden, was der Idee einer Konstante absolut widerspricht. Besser ist es, immer immutable Objekte zu referenzieren, also etwa Strings. Problematisch sind Arrays, in denen Elemente ausgetauscht werden können, veränderbare Objekte wie Date oder StringBuilder sowie mutable Datenstrukturen.

5

Vererbung und Überschattung von statischen Variablen *

Die Konstanten einer Schnittstelle können einer anderen Schnittstelle vererbt werden. Dabei gibt es einige kleine Einschränkungen. Wir wollen an einem Beispiel sehen, wie sich die Vererbung auswirkt, wenn gleiche Bezeichner in den Unterschnittstellen erneut verwendet werden:

Listing 5.106: Colors.java

```
interface BaseColors
{
    int RED      = 1;
    int GREEN    = 2;
    int BLUE     = 3;
}

interface CarColors extends BaseColors
{
    int BLACK   = 10;
    int PURPLE  = 11;
}
```

```

interface CoveringColors extends BaseColors
{
    int PURPLE = 11;
    int BLACK = 20;
    int WHITE = 21;
}

interface AllColors extends CarColors, CoveringColors
{
    int WHITE = 30;
}

public class Colors
{
    @SuppressWarnings("all")
    public static void main( String[] args )
    {
        System.out.println( CarColors.RED );           // 1
        System.out.println( AllColors.RED );           // 1
        System.out.println( CarColors.BLACK );          // 10
        System.out.println( CoveringColors.BLACK );     // 20

        System.out.println( AllColors.BLACK );          // ☹ The field AllColors.BLACK is ambiguous
        System.out.println( AllColors.PURPLE );         // ☹ The field AllColors.PURPLE is ambiguous
    }
}

```

Die erste wichtige Tatsache ist, dass Schnittstellen ohne Fehler übersetzt werden können. Doch das Programm zeigt weitere Eigenschaften:

- Schnittstellen vererben ihre Eigenschaften an die Unterschnittstellen. `CarColors` erbt die Farbe Rot aus `BaseColors`.
- Erbt eine Schnittstelle von mehreren Oberklassen, die jeweils ein bestimmtes Attribut von einer gemeinsamen Oberklasse beziehen, so ist dies kein Fehler. So erbt etwa `AllColors` von `CarColors` und `CoveringColors` die Farbe Rot.

- Konstanten dürfen überschrieben werden. `CoveringColors` überschreibt die Farbe `BLACK` aus `CarColors` mit dem Wert 20. Auch `PURPLE` wird überschrieben, obwohl die Konstante mit dem gleichen Wert belegt ist. Wird jetzt der Wert `CoveringColors.BLACK` verlangt, liefert die Umgebung den Wert 20.
- Unterschnittstellen können aus zwei Oberschnittstellen die Attribute gleichen Namens übernehmen, auch wenn sie einen unterschiedlichen Wert haben. Das zeigt sich an den beiden Beispielen `AllColors.BLACK` und `AllColors.PURPLE`. Bei der Benutzung muss ein qualifizierter Name verwendet werden, der deutlich macht, welches Attribut gemeint ist, also zum Beispiel `CarColors.BLACK`, denn die Farbe ist in den Oberschnittstellen `CarColors` und `CoveringColors` unterschiedlich initialisiert. Ähnliches gilt für die Farbe `PURPLE`. Obwohl `PURPLE` in beiden Fällen den Wert 11 trägt, ist das nicht erlaubt. Das ist ein guter Schutz gegen Fehler, denn wenn der Compiler dies durchließe, könnte sich im Nachhinein die Belegung von `PURPLE` in `CarColors` oder `CoveringColors` ohne Neuübersetzung aller Klassen ändern und zu Schwierigkeiten führen. Diesen Fehler – die Oberschnittstellen haben für eine Konstante unterschiedliche Werte – müsste die Laufzeitumgebung erkennen. Zudem kann und sollte der Compiler für alle Konstanten die Werte direkt einsetzen.

5.13.9 Initialisierung von Schnittstellenkonstanten *

Eine Schnittstelle kann Attribute deklarieren, aber das sind dann immer initialisierte `public static final`-Konstanten. Nehmen wir eine eigene Schnittstelle `PropertyReader` an, die in einer Konstanten ein `Properties`-Objekt für Eigenschaften referenziert und eine Methode `getProperties()` für implementierende Klassen vorschreibt:

```
import java.util.Properties;

public interface PropertyReader
{
    Properties DEFAULT_PROPERTIES = new Properties();

    Properties getProperties();
}
```

Würden wir `DEFAULT_PROPERTIES` nicht mit `new Properties()` initialisieren, gäbe es einen Compilerfehler, da ja jede Konstante `final` ist, also einmal belegt werden muss.



Hinweis

Referenziert eine Schnittstelle eine veränderbare Datenstruktur (wie Properties), dann muss uns die Tatsache bewusst sein, dass sie als statische Variable global ist. Das heißt, alle implementierenden Klassen teilen sich diese Datenstruktur.

Nun stellt sich ein Problem, wenn die statischen Attribute nicht einfach mit einem Standardobjekt initialisiert werden sollen, sondern wenn zusätzlicher Programmcode zur Initialisierung gewünscht ist. Für unser Beispiel soll das Properties-Objekt unter dem Schlüssel date die Zeit speichern, zu der die Klasse initialisiert wurde. Über statische Initialisierer ist dies jedenfalls nicht möglich:

```
import java.util.*;

public interface PropertyReader
{
    Properties DEFAULT_PROPERTIES = new Properties();

    static          // ☹ Compilerfehler: "Interfaces can't have static initializers"
    {
        DEFAULT_PROPERTIES.setProperty( "date", new Date().toString() );
    }

    Properties getProperties();
}
```

Zwar sind statische Initialisierungsblöcke nicht möglich, aber mit drei Tricks kann die Initialisierung erreicht werden. Wir müssen dazu etwas auf innere Klassen vorgreifen, ein Thema, das Kapitel 7 genauer aufgreift.

Konstanteninitialisierung über anonyme innere Klassen, Lösung A

Eine innere anonyme Klasse formt eine Unterklasse, sodass im Exemplarinitialisierer das Objekt (bei uns die Datenstruktur) initialisiert werden kann:

```
import java.util.*;

public interface PropertyReader
{
```

```

Properties DEFAULT_PROPERTIES = new Properties() { {
    setProperty( "date", new Date().toString() );
} };

Properties getProperties();
}

```

Ein Beispielprogramm zeigt die Nutzung:

Listing 5.107: SystemPropertyReaderDemo.java

```

import java.util.Properties;

public class SystemPropertyReaderDemo implements PropertyReader
{
    @Override public Properties getProperties()
    {
        return System.getProperties();
    }

    public static void main( String[] args )
    {
        System.out.println( PropertyReader.DEFAULT_PROPERTIES ); // {date=Thu ...
    }
}

```

Die vorgeschlagene Lösung funktioniert nur, wenn Unterklassen möglich sind; finale Klassen fallen damit raus.

Konstanteninitialisierung über statische innere Klassen, Lösung B

Mit einem anderen Trick lassen sich auch diese Hürden nehmen. Die Idee liegt in der Einführung zweier Hilfskonstrukte:

- einer inneren statischen Klasse, die wir \$\$ nennen wollen. Sie enthält einen statischen Initialisierungsblock, der auf `DEFAULT_PROPERTIES` zugreift und das `Properties`-Objekt initialisiert.
- einer Konstante \$ vom Typ \$. Als `public static final`-Variable initialisieren wir sie mit `new $$()`, was dazu führt, dass die JVM beim Laden der Klasse \$\$ den static-Block abarbeitet und so das `Properties`-Objekt belegt.

Da leider innere Klassen und Konstanten von Schnittstellen nicht privat sein können und so unglücklicherweise von außen zugänglich sind, geben wir ihnen die kryptischen Namen `$` und `$$`, sodass sie nicht so attraktiv erscheinen:

Listing 5.108: PropertyReader.java

```
import java.util.*;

public interface PropertyReader
{
    Properties DEFAULT_PROPERTIES = new Properties();

    $$ $ = new $$();

    static final class $$
    {
        static
        {
            DEFAULT_PROPERTIES.setProperty( "date", new Date().toString() );
        }
    }

    Properties getProperties();
}
```

Innerhalb vom `static`-Block lässt sich auf das `Properties`-Objekt zugreifen, und somit lassen sich auch die Werte eintragen. Ohne die Erzeugung des Objekts `$` geht es nicht, denn andernfalls würde die Klasse `$$` nicht initialisiert werden. Doch es gibt eine weitere Variante, die sogar ohne die Zwischenvariable `$` auskommt.

Konstanteninitialisierung über statische innere Klassen, Lösung C

Bei der dritten Lösung gehen wir etwas anders vor. Wir bauen kein Exemplar mit `DEFAULT_PROPERTIES = new Properties()` auf, sondern initialisieren `DEFAULT_PROPERTIES` mit einer Erzeugermethode einer eigenen internen Klasse, sodass die Initialisierung zu `DEFAULT_PROPERTIES = $$.$();` wird:

Listing 5.109: PropertyReader2.java

```
import java.util.*;  
  
public interface PropertyReader2  
{  
    Properties DEFAULT_PROPERTIES = $$.$(());  
  
    static class $$  
    {  
        static Properties $()  
        {  
            Properties p = new Properties();  
            p.setProperty( "date", new Date().toString() );  
            return p;  
        }  
    }  
  
    Properties getProperties();  
}
```

Mit dieser Lösung kann prinzipiell auch das Aufbauen eines neuen Properties-Exemplars in `$()` entfallen und können etwa schon vorher aufgebaute Objekte zurückgegeben werden.

Hinweis

Aufzählungen über enum können einfacher initialisiert werden.

5.13.10 Abstrakte Klassen und Schnittstellen im Vergleich

Eine abstrakte Klasse und eine Schnittstelle sind sich sehr ähnlich: Beide schreiben den Unterklassen beziehungsweise den implementierten Klassen Operationen vor, die sie implementieren müssen. Ein wichtiger Unterschied ist jedoch, dass beliebig viele Schnittstellen implementiert werden können, doch nur eine Klasse – sei sie abstrakt oder nicht – erweitert werden kann. Des Weiteren bieten sich abstrakte Klassen meist im Refactoring oder in der Design-Phase an, wenn Gemeinsamkeiten in eine Oberklasse ausgelagert werden sollen. Abstrakte Klassen können zusätzlichen Programmcode ent-

halten, was Schnittstellen nicht können. Auch nachträgliche Änderungen an Schnittstellen sind nicht einfach: Einer abstrakten Klasse kann eine konkrete Methode mitgegeben werden, was zu keiner Quellcodeanpassung für Unterklassen führt.

Ein Beispiel: Ist eine Schnittstelle oder eine abstrakte Klasse besser, um folgende Operation zu deklarieren?

```
abstract class Timer
{
    abstract long getTimeInMillis();
}

interface Timer
{
    long getTimeInMillis();
}
```

Eine abstrakte Klasse hätte den Vorteil, dass später einfacher eine Methode wie `getTimeInSeconds()` eingeführt werden kann, die konkret sein darf. Würde diese angenehme Hilfsoperation in einer Schnittstelle vorgeschrieben, so müssten alle Unterklassen diese Implementierung immer neu einführen, wobei sie doch schon in der abstrakten Oberklasse einfach programmiert werden könnte:

```
abstract class Timer
{
    abstract long getTimeInMillis();

    long getTimeInSeconds()
    {
        return getTimeInMillis() / 1000;
    }
}
```

5.14 Zum Weiterlesen

Gute objektorientierte Modellierung ist nicht einfach und bedarf viel Übung. Beim »reinkommen« in die Denkweise hilft es, viel Quellcode zu lesen und sich insbesondere UML-Diagramme der Java-Standard-Bibliothek zu machen, damit die Zusammenhänge klarer werden.



Kapitel 6

Exceptions

»*Wir sind in Sicherheit! Er kann uns nicht erreichen!*«
›Sicher?‹
›Ganz sicher! Bären haben Angst vor Treibsand!‹«
– Hägar, Dik Browne (1917–1989)

6

Fehler beim Programmieren sind unvermeidlich. Schwierigkeiten bereiten nur die unkalkulierbaren Situationen – hier ist der Umgang mit Fehlern ganz besonders heikel. Java bietet die elegante Methode der Exceptions, um mit Fehlern flexibel umzugehen.

6.1 Problembereiche einzäunen

Werden in C Routinen aufgerufen, dann haben diese keine andere Möglichkeit, als über den Rückgabewert einen Fehlschlag anzuzeigen. Der Fehlercode ist häufig `-1`, aber auch `NULL` oder `0`. Allerdings kann die Null auch Korrektheit anzeigen. Irgendwie ist das willkürlich. Die Abfrage dieser Werte ist unschön und wird von uns gern unterlassen, zumal wir oft davon ausgehen, dass ein Fehler in dieser Situation gar nicht auftreten kann – diese Annahme kann aber eine Dummheit sein. Zudem wird der Programmfluss durch Abfragen der Rückgabergebnisse unangenehm unterbrochen, zumal der Rückgabewert, wenn er nicht gerade einen Fehler anzeigt, weiterverwendet wird. Der Rückgabewert ist also im weitesten Sinne überladen, da er zwei Zustände anzeigt. Häufig entstehen mit den Fehlerabfragen kaskadierte `if`-Abfragen, die den Quellcode schwer lesbar machen.

Beispiel

zB

Die Java-Bibliothek geht bei den Methoden `delete()`, `mkdir()`, `mkdirs()` und `renameTo()` der Klasse `File` nicht mit gutem Beispiel voran. Anstatt über eine Ausnahme anzusegnen, dass die Operation nicht geglückt ist, liefern die genannten Methoden `false`. Das ist unglücklich, denn viele Entwickler verzichten auf den Test, und so entstehen Fehler, die später schwer zu finden sind.

6.1.1 Exceptions in Java mit try und catch

Bei der Verwendung von Exceptions wird der Programmfluss nicht durch Abfrage des Rückgabestatus unterbrochen. Ein besonders ausgezeichnetes Programmstück überwacht mögliche Fehler und ruft gegebenenfalls speziellen Programmcode zur Behandlung auf.

Den überwachten Programmreich (Block) leitet das Schlüsselwort `try` ein. Dem `try`-Block folgt in der Regel¹ ein `catch`-Block, in dem Programmcode steht, der den Fehler behandelt. Kurz skizziert, sieht das so aus:

```
try
{
    // Programmcode, der eine Ausnahme ausführen kann
}
catch ( ... )
{
    // Programmcode zum Behandeln der Ausnahme
}
// Es geht ganz normal weiter, denn die Ausnahme wurde behandelt
```

Hinter `catch` folgt also der Programmblock, der beim Auftreten eines Fehlers ausgeführt wird, um den Fehler abzufangen (daher der Ausdruck `catch`). Es ist nach der Fehlerbehandlung nicht mehr so einfach möglich, an der Stelle fortzufahren, an der der Fehler auftrat. Andere Programmiersprachen erlauben das durchaus.



¹ In manchen Fällen auch ein `finally`-Block, sodass es dann ein `try-finally` wird.

6.1.2 Eine NumberFormatException auffangen

Über die Methode `Integer.parseInt()` haben wir an verschiedenen Stellen schon gesprochen. Sie konvertiert eine Zahl, die als Zeichenkette gegeben ist, in eine Dezimalzahl:

```
int vatRate = Integer.parseInt( "19" );
```

In dem Beispiel ist eine Konvertierung möglich, und die Methode führt die Umwandlung ohne Fehler aus. Anders sieht das aus, wenn der String keine Zahl repräsentiert:

Listing 6.1: MissNumberFormatException.java

```
/* 01 */public class MissNumberFormatException
/* 02 */{
/* 03 */    public static int getVatRate()
/* 04 */    {
/* 05 */        return Integer.parseInt( "19%" );
/* 06 */    }
/* 07 */    public static void main( String[] args )
/* 08 */    {
/* 09 */        System.out.println( getVatRate() );
/* 10 */    }
/* 11 */}
```

Die Ausführung des Programms bricht mit einem Fehler ab, und die virtuelle Maschine gibt uns automatisch eine Meldung aus:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "19%"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
  at java.lang.Integer.parseInt(Integer.java:456)
  at java.lang.Integer.parseInt(Integer.java:497)
  at MissNumberFormatException.getVatRate(MissNumberFormatException.java:5)
  at MissNumberFormatException.main(MissNumberFormatException.java:9)
```

In der ersten Zeile können wir ablesen, dass eine `java.lang.NumberFormatException` ausgelöst wurde. In der letzten Zeile steht, welche Stelle in unserem Programm zu dem Fehler führte (Fehlerausgaben wie diese haben wir schon in »Auf null geht nix, nur die NullPointerException« in Abschnitt 3.7.1 beobachtet).

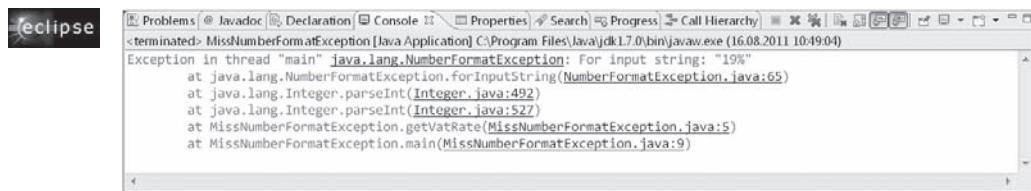


Abbildung 6.1: Tritt eine Exception auf, so erscheint sie im Ausgabefenster rot. Praktischerweise sind die Fehlermeldungen wie Hyperlinks: Ein Klick, und Eclipse zeigt die Zeile, die die Exception auslöst.

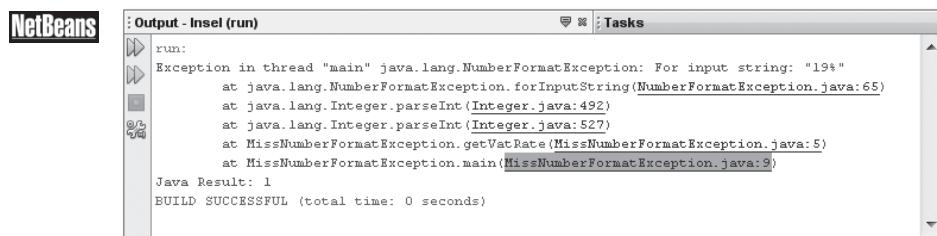


Abbildung 6.2: Auch bei NetBeans führt ein Klick auf die Fehlerstelle und in den Quellcode.

Dokumentierte Fehler

Der Fehler kommt nicht wirklich überraschend, und Entwickler müssen sich darauf vorbereiten, dass, wenn sie etwas Falsches an Methoden übergeben, diese schimpfen. Im besten Fall erklärt die API-Dokumentation, welche Eingaben eine Methode zulässt und welche nicht. Zur »Schnittstelle« einer Methode gehört auch das Verhalten im Fehlerfall. Die API-Dokumentation sollte genau beschreiben, welche Ausnahme – oder Reaktion wie spezielle Rückgabewerte – zu erwarten ist, wenn die Methode ungültige Werte erhält. Die Java-Dokumentation bei `Integer.parseInt()` macht das:

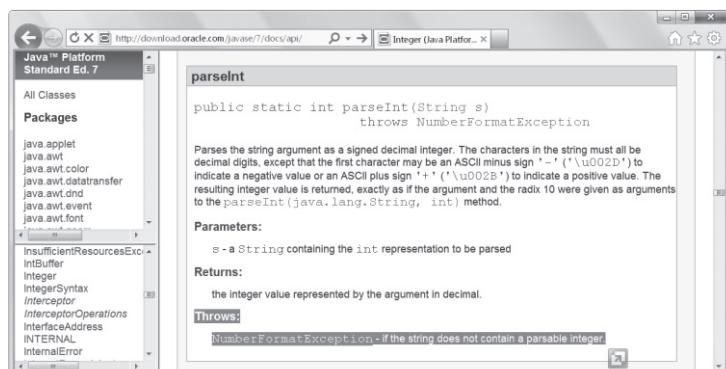


Abbildung 6.3: JavaDoc dokumentiert die Ausnahme.

Stack-Trace

Die virtuelle Maschine merkt sich auf einem Stapel, welche Methode welche andere Methode aufgerufen hat. Dies nennt sich *Stack-Trace*. Wenn also die statische `main()`-Methode die Methode `getVatRate()` aufruft und diese wiederum `parseInt()`, so sieht der Stapel zum Zeitpunkt von `parseInt()` so aus:

```
parseInt  
getVatRate  
main
```

Ein Stack-Trace ist im Fehlerfall nützlich, da wir etwa bei unserem `parseInt("19%")` ablesen können, dass `parseInt()` den Fehler ausgelöst hat und nicht irgendeine andere Methode.

Eine NumberFormatException auffangen

Dass ein Programm einfach so abbricht und die JVM endet, ist üblicherweise keine Lösung. Fehler sollten aufgefangen und gemeldet werden. Um Fehler aufzufangen, ist es erst einmal wichtig zu wissen, was genau für eine Ausnahme ausgelöst wird. In unserem Fall ist das einfach abzulesen, denn die Ausnahme ist ja schon aufgetaucht und klar einem Grund zuzuordnen. Die Java-Dokumentation nennt diesen Fehler auch, und weil ohne den aufgefangenen Fehler das Programm abbricht, soll nun die `NumberFormatException` aufgefangen werden. Dabei kommt die try-catch-Konstruktion zum Einsatz:

Listing 6.2: CatchTheNumberFormatException.java, main()

```
String stringToConvert = "19%";  
  
try  
{  
    Integer.parseInt( stringToConvert );  
}  
catch ( NumberFormatException e )  
{  
    System.err.printf( "'%s' kann man nicht in eine Zahl konvertieren!%n",  
                      stringToConvert );  
}  
System.out.println( "Weiter geht's" );
```

Die gesamte Ausgabe ist:

```
'19%' kann man nicht in eine Zahl konvertieren!
Weiter geht's
```

Die Anweisung `catch(NumberFormatException e)` fängt also alles auf, was vom Ausnahmetyp `NumberFormatException` ist. `Integer.parseInt("19%")` führt, da der String keine Zahl ist, zu einer `NumberFormatException`, die wir behandeln. Danach ist der Fehler wie weggeblasen, und mit der Konsolenausgabe geht es ganz normal weiter.

6.1.3 Ablauf einer Ausnahmesituation

Das Laufzeitsystem erzeugt ein Ausnahme-Objekt, wenn ein Fehler über eine Exception angezeigt werden soll. Dann wird die Abarbeitung der Programmzeilen sofort unterbrochen, und das Laufzeitsystem steuert selbstständig die erste catch-Klausel an (oder springt weiter zum Aufrufer, wie wir später sehen werden). Wenn die erste catch-Anweisung nicht zur Art des aufgetretenen Fehlers passt, werden der Reihe nach alle übrigen catch-Klauseln untersucht, und die erste übereinstimmende Klausel wird angesprungen (oder ausgewählt). Erst wird etwas versucht (daher heißt es im Englischen *try*), und wenn im Fehlerfall ein Exception-Objekt im Programmstück ausgelöst (engl. *throw*) wird, lässt es sich an einer Stelle auffangen (engl. *catch*). Da immer die erste passende catch-Klausel ausgewählt wird, darf im Beispiel die letzte catch-Klausel keinesfalls zuerst stehen, da diese auf jeden Fehler passt. Alle anderen Anweisungen in den catch-Blöcken würden dann nicht ausgeführt; der Compiler erkennt dieses Problem und gibt einen Fehler aus.

6.1.4 Eigenschaften vom Exception-Objekt

Das Exception-Objekt, das uns in der catch-Anweisung übergeben wird, ist reich an Informationen. So lässt sich erfragen, um welche Ausnahme es sich eigentlich handelt und wie die Fehlermeldung heißt. Auch der Stack-Trace lässt sich erfragen und ausgeben:

Listing 6.3: NumberFormatExceptionElements.java, main()

```
try
{
    Integer.parseInt( "19%" );
}
```

```

        catch ( NumberFormatException e )
        {
            String name = e.getClass().getName();
            String msg  = e.getMessage();
            String s    = e.toString();

            System.out.println( name );// java.lang.NumberFormatException
            System.out.println( msg ); // For input string: "19%"
            System.out.println( s ); // java.lang.NumberFormatException: For input string: "19%"

            e.printStackTrace();
        }
    
```

Im letzten Fall, mit `e.printStackTrace()`, bekommen wir das Gleiche auf dem Fehlerkanal `System.err` ausgegeben, was uns die virtuelle Maschine ausgibt, wenn wir die Ausnahme nicht abfangen:

```

java.lang.NumberFormatException: For input string: "19%"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:456)
    at java.lang.Integer.parseInt(Integer.java:497)
    at NumberFormatExceptionElements.main(NumberFormatExceptionElements.java:7)

```

Die Ausgabe besteht aus dem Klassennamen der Exception, der Meldung und dem Stack-Trace. `printStackTrace()` ist parametrisiert und kann auch in einen Ausgabekanal geschickt werden.

Bitte nicht schlucken: leere catch-Blöcke

Java schreibt vor, dass Ausnahmen in einem `catch` behandelt (oder nach oben geleitet) werden, aber nicht, was in `catch`-Blöcken zu geschehen hat. Er kann eine sinnvolle Behandlung beinhalten oder auch einfach leer sein. Ein leerer `catch`-Block ist in der Regel wenig sinnvoll, weil dann die Fehler klammheimlich unterdrückt werden. (Das wäre genauso wie ignorierte Statusrückgabewerte von C-Funktionen.) Das Mindeste ist eine minimale Fehlerausgabe via `System.err.println(e)` oder das informativere `e.printStackTrace()` für eine `Exception e` oder das Loggen dieser Fehler. Noch besser ist das aktive Reagieren, denn die Ausgabe selbst behandelt diesen Fehler nicht! Im `catch`-Block ist es durchaus legitim, wiederum andere Ausnahmen auszulösen und somit den Fehler umzuformen und nach oben weiterzureichen.



Hinweis

Wenn wie bei einem Thread.sleep() die InterruptedException wirklich egal ist, kann natürlich auch der Block leer sein, doch gibt es dafür nicht so viele sinnvolle Beispiele.

6.1.5 Wiederholung abgebrochener Bereiche *

Es gibt in Java bei Ausnahmen bisher keine von der Sprache unterstützte Möglichkeit, an den Punkt zurückzukehren, der den Fehler ausgelöst hat. Das ist aber oft erwünscht, etwa dann, wenn eine fehlerhafte Eingabe zu wiederholen ist.

Wir werden mit JOptionPane.showInputDialog() nach einem String fragen und versuchen, diesen in eine Zahl zu konvertieren. Dabei kann natürlich etwas schiefgehen. Wenn ein Benutzer eine Zeichenkette eingibt, die keine Zahl repräsentiert, löst parseInt() eine NumberFormatException aus. Wir wollen in diesem Fall die Eingabe wiederholen:

Listing 6.4: ContinuelInput.java, main()

```
int number = 0;
while ( true )
{
    try
    {
        String s = javax.swing.JOptionPane.showInputDialog(
            "Bitte Zahl eingeben" );
        number = Integer.parseInt( s );
        break;
    }
    catch ( NumberFormatException ó_ò )
    {
        System.err.println( "Das war keine Zahl!" );
    }
}
System.out.println( "Danke für die Zahl " + number );
System.exit( 0 );                                // Beendet die Anwendung
```

Die gewählte Lösung ist einfach: Wir programmieren den gesamten Teil in einer Endlosschleife. Geht die problematische Stelle ohne Fehler durch, so beenden wir die Schleife

mit `break`. Kommt es zu einer Ausnahme, dann wird `break` nicht ausgeführt, und nach der `Exception` gelangen wir wieder in die Endlosschleife.

6.1.6 Mehrere Ausnahmen auffangen

Wir wollen mithilfe der Klasse `Scanner` eine Webseite zeilenweise auslesen und alle dort enthaltenen E-Mail-Adressen sammeln. Dazu greifen wir zu zwei Klassen, die uns beim Einlesen der Zeilen helfen: `URL` und `Scanner` (siehe dazu Abschnitt 4.9.2, »Die Klasse Scanner«). Zunächst repräsentiert die Klasse `URL` eine URL, also eine Internetadresse. Das `URL`-Objekt fragen wir mit `openStream()` nach einem Datenstrom, und diesen Datenstrom setzen wir in den Konstruktor der `Scanner`-Klasse. Mit dem `Scanner` können wir dann zeilenweise durch die Seite laufen und alles einsammeln, was wie eine E-Mail-Adresse aussieht.

Anders als bei `Integer.parseInt()` kündigt die API-Dokumentation vom Konstruktor der Klasse `URL` an, dass eine Ausnahme ausgelöst wird, und zwar genau dann, wenn die URL falsch formuliert wird (etwa als "telefon://0123-123123"). Vergleichbares gilt bei der `URL`-Methode `openStream()`. Die Methode löst eine `IOException` aus, wenn es keinen Zugriff auf die Webseite gibt.

URL

```
public URL(String spec)
           throws MalformedURLException
```

Creates a `URL` object from the string representation.

This constructor is equivalent to a call to the two-argument constructor with a `null` first argument.

Parameters:
`spec` - the string to parse as a URL.

Throws:
`MalformedURLException` - if no protocol is specified, or an unknown protocol is found, or `spec` is null.

See Also:
`URL(java.net.URL, java.lang.String)`

openStream

```
public final InputStream openStream()
                                    throws IOException
```

Opens a connection to this `URL` and returns an `InputStream` for reading from that connection. This method is a shorthand for:
`openConnection().getInputStream()`

Returns:
an input stream for reading from the `URLConnection`.

Throws:
`IOException` - if an I/O exception occurs.

See Also:
`openConnection(), URLConnection.getInputStream()`

Abbildung 6.4: API-Dokumentation zeigt die Ausnahmen.

Beide Ausnahmen sind sogenannte *geprüfte Ausnahmen*, da sie explizit vom Entwickler behandelt werden müssen. Damit zwingen uns der Konstruktor `new URL()` und die Methode `openStream()` eine Behandlung auf, ohne die wir sie nicht nutzen könnten.



Abbildung 6.5: Eine nicht behandelte Ausnahme wird als Fehler angezeigt.

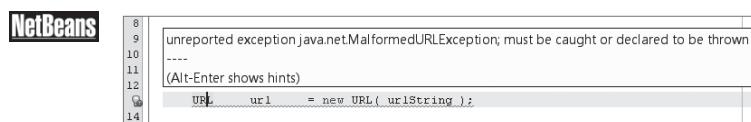


Abbildung 6.6: NetBeans zeigt Fehler an, und mit `Alt` + `←` können sie direkt behoben werden.

Wir müssen uns diesen potenziellen Fehlern also stellen und daher die Problemzonen in einen `try`- und `catch`-Block schreiben:

Listing 6.5: FindAllEmailAddresses.java

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class FindAllEmailAddresses
{
    public static void main( String[] args )
    {
        printAllEMailAddresses( "http://www.galileocomputing.de/hilfe/Impressum" );
    }

    static void printAllEMailAddresses( String urlString )
    {
        try
```

```
{  
    URL url = new URL( urlString );  
    Scanner scanner = new Scanner( url.openStream() );  
    Pattern pattern = Pattern.compile( "[\\w|-]+@[\\w[\\w|-]*\\.\\{2,3}" );  
  
    while ( scanner.hasNextLine() )  
    {  
        String line = scanner.nextLine();  
        for ( Matcher m = pattern.matcher( line ); m.find(); )  
            System.out.println( line.substring( m.start(), m.end() ) );  
    }  
}  
catch ( MalformedURLException e )  
{  
    System.err.println( "URL ist falsch aufgebaut!" );  
}  
catch ( IOException e )  
{  
    System.err.println( "URL konnte nicht geöffnet werden!" );  
}  
}
```

6

Tritt beim Erzeugen des URL-Objekts oder bei der Verbindung ein Fehler auf, fängt der try-Block diesen ab, und der catch-Teil bearbeitet ihn. Einem try-Block können mehrere catch-Klauseln zugeordnet sein, um verschiedene Fehlertypen aufzufangen.

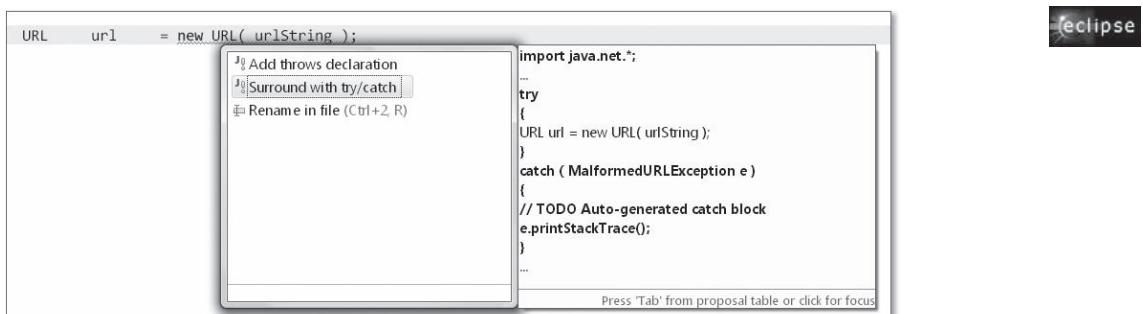


Abbildung 6.7: Einen try-catch-Block kann Eclipse mit **[Strg] + 1** selbst anlegen. Auch bietet Eclipse an, den Fehler an den Aufrufer weiterzuleiten (siehe weiter unten).

6.1.7 throws im Methodenkopf angeben

Neben der rahmenbasierten Ausnahmebehandlung – dem »Einzäunen« von problematischen Blöcken durch einen `try`- und `catch`-Block – gibt es eine weitere Möglichkeit, auf Exceptions zu reagieren: das Weiterleiten an den Aufrufer. Im Kopf der betreffenden Methode wird dazu eine `throws`-Klausel eingeführt. Dadurch zeigt die Methode an, dass sie eine bestimmte Exception nicht selbst behandelt, sondern diese an die aufrufende Methode weitergibt. Wird nun von der aufgerufenen Methode eine Exception ausgelöst, so wird diese Methode abgebrochen, und der Aufrufer muss sich um den Fehler kümmern.

Wir können unsere Methode `printAllEmailAddresses()` so umschreiben, dass sie die Ausnahmen nicht mehr selbst abfängt, sondern nach oben weiterleitet:

Listing 6.6: FindAllEmailAddresses2.java, `printAllEmailAddresses()`

```
static void printAllEmailAddresses( String urlString )
    throws MalformedURLException, IOException
{
    Scanner scanner = new Scanner( new URL( urlString ).openStream() );
    Pattern pattern = Pattern.compile( "[\\w\\-]+@[\\w\\-]*\\.\\w{2,3}" );

    while ( scanner.hasNextLine() )
    {
        String line = scanner.nextLine();
        for ( Matcher m = pattern.matcher( line ); m.find(); )
            System.out.println( line.substring( m.start(), m.end() ) );
    }
    scanner.close();
}
```

Nun ist `main()` am Zug und muss sich mit `MalformedURLException` und `IOException` herumärgern:

Listing 6.7: FindAllEmailAddresses2.java, `main()`

```
public static void main( String[] args )
{
    try
    {
```

```
    printAllEMailAddresses( "http://www.galileocomputing.de/hilfe/Impressum" );
}
catch ( MalformedURLException e )
{
    System.err.println( "URL ist falsch aufgebaut!" );
}
catch ( IOException e )
{
    System.err.println( "URL konnte nicht geöffnet werden!" );
}
}
```

Dadurch steigt der Fehler entlang der Kette von Methodenaufrufen wie eine Blase (engl. *bubble*) nach oben und kann irgendwann von einem Block abgefangen werden, der sich darum kümmert.

Hinweis

Zwar ist die MalformedURLException eine IOException, sodass wir hier nur IOException hätten angeben müssen, doch grundsätzlich lassen sich beliebig viele Ausnahmen, getrennt durch Kommata, aufzählen. Zu den Vererbungsbeziehungen und den Konsequenzen folgt später mehr.

6.1.8 Abschlussbehandlung mit finally

Im Folgenden wollen wir eine optimale Exception-Behandlung programmieren. Es geht im Beispiel darum, die Ausmaße eines GIF-Bildes auszulesen. Das Grafikformat GIF ist sehr einfach und gut dokumentiert, etwa unter <http://www.fileformat.info/format/gif/egff.htm>. Dort lässt sich erfahren, wie sich die Ausmaße ganz einfach im Kopf einer GIF-Datei ablesen lassen, denn nach den ersten Bytes 'G', 'T', 'F', '8', '7' (oder '9'), 'a' folgen in 2 Bytes an Position 6 und 7 die Breite und an Position 8 und 9 die Höhe des Bildes.

Die ignorante Version

In der ersten Variante schreiben wir den Algorithmus einfach herunter und kümmern uns nicht um die Fehlerbehandlung; mögliche Ausnahmen leitet die statische main()-Methode an die JVM weiter:

Listing 6.8: ReadGifSizeIgnoringExceptions.java

```
import java.io.*;

public class ReadGifSizeIgnoringExceptions
{
    public static void main( String[] args )
        throws FileNotFoundException, IOException
    {
        RandomAccessFile f = new RandomAccessFile( "duke.gif", "r" );
        f.seek( 6 );

        System.out.printf( "%s x %s Pixel%n", f.read() + f.read() * 256,
                           f.read() + f.read() * 256 );
    }
}
```

In der Klasse haben wir eine Kleinigkeit noch nicht beachtet: das Schließen des Datenstroms. Das Programm endet mit dem Auslesen der Bytes, aber das Schließen mit `close()` fehlt. Nehmen wir eine Zeile nach der Konsolenausgabe hinzu:

```
...
System.out.printf( "%s x %s Pixel%n", f.read() + f.read() * 256,
                   f.read() + f.read() * 256 );
f.close();
```

Das `close()` wiederum kann auch eine `IOException` auslösen, die jedoch schon über `throws` in der `main`-Signatur angekündigt wurde.

Der gut gemeinte Versuch

Dass ein Programm die JVM beendet, sobald eine Datei nicht da ist, ist ein bisschen hart. Daher wollen wir ein try-catch formulieren und den Fehler ordentlich abfangen und dokumentieren:

Listing 6.9: ReadGifSizeCatchingExceptions.java

```
import java.io.*;

public class ReadGifSizeCatchingExceptions
```

```
{
    public static void main( String[] args )
    {
        try
        {
            RandomAccessFile f = new RandomAccessFile( "duke.gif", "r" );
            f.seek( 6 );

            System.out.printf( "%s x %s Pixel%n", f.read() + f.read() * 256,
                               f.read() + f.read() * 256 );
            f.close();
        }
        catch ( FileNotFoundException e )
        {
            System.err.println( "Datei ist nicht vorhanden!" );
        }
        catch ( IOException e )
        {
            System.err.println( "Allgemeiner Ein-/Ausgabefehler!" );
        }
    }
}
```

Ist damit alles in Ordnung?

Ab jetzt wird scharf geschlossen

Nehmen wir an, das Öffnen führt zu keiner Ausnahme, doch beim Zugriff auf ein Byte kommt es unerwartet zu einem Fehler. Das `read()` wird abgebrochen, und die JVM leitet uns in den Exception-Block, der eine Meldung ausgibt. Das Problem: Dann schließt das Programm den Datenstrom nicht. Wir könnten verleitet werden, in den `catch`-Zweig auch ein `close()` zu schreiben, doch ist das eine Quellcodeduplizierung, die wir vermeiden müssen. Hier kommt ein `finally`-Block zum Zuge. `finally`-Blöcke stehen immer hinter `catch`-Blöcken, und ihre wichtigste Eigenschaft ist die, dass der Programmcode im `finally`-Block immer ausgeführt wird, egal, ob es einen Fehler gab oder ob es keinen Fehler gab und die Routine glatt durchlief. Das ist genau, was wir hier bei der Ressour-

cenfreigabe brauchen. Da `finally` immer ausgeführt wird, wird die Datei geschlossen (und der interne File-Handle freigegeben), wenn alles gut ging – und ebenso im Fehlerfall:

Listing 6.10: ReadGifSize.java, main()

```
RandomAccessFile f = null;

try
{
    f = new RandomAccessFile( "duke.gif", "r" );
    f.seek( 6 );

    System.out.printf( "%s x %s Pixel%n", f.read() + f.read() * 256,
                      f.read() + f.read() * 256 );
}

catch ( FileNotFoundException e )
{
    System.err.println( "Datei ist nicht vorhanden!" );
}

catch ( IOException e )
{
    System.err.println( "Allgemeiner Ein-/Ausgabefehler!" );
}

finally
{
    if ( f != null )
        try { f.close(); } catch ( IOException e ) { }
}
```

Da `close()` eine `IOException` auslösen kann, muss der Aufruf selbst mit einem `try-catch` ummantelt werden. Das führt zu etwas abschreckenden Konstruktionen, die *TCFTC* (`try-catch-finally-try-catch`) genannt werden. Ein zweiter Schönheitsfehler ist der, dass die Variable `f` nun außerhalb des `try`-Blocks deklariert werden muss. Das gibt ihr als lokale Variable einen größeren Radius – größer, als er eigentlich sein sollte. Mit einem extra Block lässt sich das lösen, sieht aber nicht so hübsch aus. Das spezielle Sprachkonstrukt *try-mit-Ressourcen* löst das elegant; Informationen dazu folgen in Abschnitt 6.6.1.

Zusammenfassung

Nach einem `catch` (oder mehreren) kann optional ein `finally`-Block folgen. Die Laufzeitumgebung führt die Anweisungen im `finally`-Block immer aus, egal, ob ein Fehler auftrat oder die Anweisungen im `try`-Block optimal durchliefen. Das heißt, der Block wird auf jeden Fall ausgeführt – lassen wir `System.exit()` oder Systemfehler einmal außen vor –, auch wenn im `try`-Block ein `return`, `break` oder `continue` steht oder eine Anweisung eine neue Ausnahme auslöst. Der Programmcode im `finally`-Block bekommt auch gar nicht mit, ob vorher eine Ausnahme auftrat oder alles glattlief. Wenn das von Interesse ist, müsste eine Anweisung am Ende des `try`-Blocks ein Flag belegen, was ein Ausdruck im `finally`-Block dann testen kann.

Sinnvoll sind Anweisungen im `finally`-Block immer dann, wenn Operationen stets ausgeführt werden sollen. Eine typische Anwendung ist die Freigabe von Ressourcen wie das Schließen von Dateien.

Hinweis

Es gibt bei Objekten einen Finalizer, doch der hat mit `finally` nichts zu tun. Der Finalizer ist eine besondere Methode, die immer dann aufgerufen wird, wenn der Garbage-Collector ein Objekt wegräumt.



Ein `try` ohne `catch`, aber ein `try-finally`

Ein `try`-Block fängt immer Ausnahmen ab, doch nicht zwingend muss ein angehängerter `catch`-Block diese behandeln; `throws` kann die Ausnahmen einfach nach oben weiterleiten. Nur eine Konstruktion der Art `try {}` ohne `catch` ist ungültig, jedoch ist ein `try`-Block ohne `catch` aber mit `finally` absolut legitim. Diese Konstruktion ist in Java gar nicht so selten, denn sie ist wichtig, wenn eben kein Fehler behandelt werden soll, aber unabhängig von möglichen Ausnahmen immer Programmcode abgearbeitet werden soll – ein typisches Beispiel ist die Ressourcen-Freigabe.

Kommen wir zu unserem Programm zurück, das die Größe eines GIF-Bildes ermittelt. Wenn beim IO-Fehler eben nichts zu retten ist, geben wir den Fehler an den Aufrufer weiter, ohne es jedoch zu versäumen, die in der Methode angeforderten Ressourcen wieder freizugeben.

Listing 6.11: ReadGifSizeWithTryFinally.java

```
import java.io.*;
```

```

public class ReadGifSizeWithTryFinally
{
    public static void printGifSize( String filename )
        throws FileNotFoundException, IOException
    {
        RandomAccessFile f = new RandomAccessFile( filename, "r" );

        try
        {
            f.seek( 6 );

            System.out.printf( "%s x %s Pixel%n", f.read() + f.read() * 256,
                f.read() + f.read() * 256 );
        }
        finally
        {
            f.close();
        }
    }

    public static void main( String[] args )
        throws FileNotFoundException, IOException
    {
        printGifSize( "duke.gif" );
    }
}

```

Anstatt im finally-Block die IOException vom close() selbst zu fangen, leiten wir sie in dieser Implementierung auch mit nach oben, wenn es zu einem Fehler beim Schließen kommt. Im vorangehenden Beispiel *ReadGifSize.java* hatten wir geschrieben:

```

if ( f != null )
    try { f.close(); } catch ( IOException e ) { }

```

Eine IOException bei close() würde leise versacken, denn der Behandler ist leer. Bei *ReadGifSizeWithTryFinally.java* wird ein möglicher Schließfehler nach oben geleitet, bei *ReadGifSize.java* jedoch nicht, denn dort ist der der Programmfluss ganz anders.

Aus noch einem Grund ist die Semantik anders, und daher ist von diesem Stil abzusehen, wenn im finally-Block wie bei *ReadGifSizeWithTryFinally.java* Ausnahmen ausgelöst werden können.

Wichtig: Java-Verhalten bei multiplen Ausnahmen

Kommt es im try-Block zur Ausnahme und löst auch gleichzeitig der finally-Block eine Ausnahme aus, so wird die Ausnahme im try-Block ignoriert – wir sprechen von einer *unterdrückten Ausnahme* (engl. *suppressed exception*). In den Zeilen

```
try
{
    throw new Error();
}
finally
{
    System.out.println( "Geht das?" + 1/0 );
}
```

kommt die im Kontext uninteressante `ArithmaticException` durch die Division durch null zum Aufrufer, aber sie unterdrückt den viel wichtigeren harten `Error`.

!

6

Gibt es in unserem Beispiel im try-Block eine Ausnahme und ebenso im finally-Block beim Schließen, dann überdeckt die Schließ-Ausnahme jede andere Ausnahme. Nun ist die Ausnahme im try-Block aber in der Regel wichtiger und sollte nicht verschwinden. Um das Problem zu lösen, gibt es ein anderes Sprachmittel, das Abschnitt 6.6 vorstellt.

6.2 RuntimeException muss nicht aufgefangen werden

Einige Fehlerarten können potenziell an vielen Programmstellen auftreten, etwa eine ganzzahlige Division durch null² oder ungültige Indexwerte beim Zugriff auf Array-Elemente. Treten solche Fehler beim Programmlauf auf, liegt dem in der Regel ein Denkfehler des Programmierers zugrunde, und das Programm sollte normalerweise nicht versuchen, die ausgelöste Ausnahme aufzufangen und zu behandeln. Daher gibt es in der Java-API mit der Klasse `RuntimeException` eine Unterklasse von `Exception`, die Pro-

² Fließkommadivisionen durch 0.0 ergeben entweder \pm unendlich oder NaN.

grammierfehler aufzeigt, die behoben werden müssen. (Der Name »`RuntimeException`« ist jedoch seltsam gewählt, da alle Ausnahmen immer zur Runtime, also zur Laufzeit, erzeugt, ausgelöst und behandelt werden.)

6.2.1 Beispiele für `RuntimeException`-Klassen

Die Java-API bietet insgesamt eine große Anzahl von `RuntimeException`-Klassen, und es werden immer mehr. Die Tabelle listet einige bekannte Fehlertypen auf und zeigt, welche Operationen die Fehler auslösen. Wir greifen hier schon auf spezielle APIs zurück, die erst später im Buch vorgestellt werden.

Unterkasse von <code>RuntimeException</code>	Was den Fehler auslöst
<code>ArithmaticException</code>	Ganzzahlige Division durch 0
<code>ArrayIndexOutOfBoundsException</code>	Indexgrenzen wurden missachtet, etwa durch <code>(new int[0])[1]</code> . Eine <code>ArrayIndexOutOfBoundsException</code> ist neben <code>StringIndexOutOfBoundsException</code> eine Unterklasse von <code>IndexOutOfBoundsException</code> .
<code>ClassCastException</code>	Typanpassung ist zur Laufzeit nicht möglich. So löst <code>(java.util.Stack) new java.util.Vector()</code> eine <code>ClassCastException</code> mit der Meldung » <code>java.util.Vector</code> cannot be cast to <code>java.util.Stack</code> « aus.
<code>EmptyStackException</code>	Der Stapspeicher ist leer. <code>new java.util.Stack().pop()</code> provoziert den Fehler.
<code>IllegalArgumentException</code>	Eine häufig verwendete Ausnahme, mit der Methoden falsche Argumente melden. <code>Integer.parseInt("tutego")</code> löst eine <code>NumberFormatException</code> , eine Unterklasse von <code>IllegalArgumentException</code> , aus.
<code>IllegalMonitorStateException</code>	Ein Thread möchte warten, hat aber den Monitor nicht. Ein Beispiel: <code>new String().wait()</code> ;
<code>NullPointerException</code>	Meldet einen der häufigsten Programmierfehler, beispielsweise durch <code>((String) null).length()</code> .
<code>UnsupportedOperationException</code>	Operationen sind nicht gestattet, etwa durch <code>java.util.Arrays.asList(args).add("chris")</code> .

Tabelle 6.1: `RuntimeException`-Klassen

6.2.2 Kann man abfangen, muss man aber nicht

Eine `RuntimeException` muss der Entwickler nicht abfangen, er kann es aber tun. Da der Compiler nicht auf einem Auffangen besteht, heißen die aus `RuntimeException` hervorgegangenen Ausnahmen auch *nicht geprüfte Ausnahmen* (engl. *unchecked exceptions*), und alle übrigen heißen *geprüfte Ausnahmen* (engl. *checked exceptions*). Auch muss eine `RuntimeException` nicht unbedingt bei `throws` in der Methodensignatur angegeben werden, wobei einige Autoren das zur Dokumentation machen. Tritt eine `RuntimeException` zur Laufzeit auf und kommt nicht irgendwann in der Aufrufhierarchie ein try-catch, beendet die JVM den ausführenden Thread. Löst also eine in `main()` aufgerufene Aktion eine `RuntimeException` aus, ist das das Ende für dieses Hauptprogramm.

6.3 Die Klassenhierarchie der Fehler

Eine `Exception` ist ein Objekt, dessen Typ direkt oder indirekt von `java.lang.Throwable` abgeleitet ist (die Namensgebung mit `-able` legt eine Schnittstelle nahe, aber `Throwable` ist eine nicht-abstrakte Klasse). Von dort aus verzweigt sich die Hierarchie der Fehlerarten nach `java.lang.Exception` und `java.lang.Error`. Die Klassen, die aus `Error` hervorgehen, sollen nicht weiterverfolgt werden. Es handelt sich hierbei um so schwerwiegende Fehler, dass sie zur Beendigung des Programms führen und vom Programmierer nicht weiter beachtet werden müssen und sollen. `Throwable` vererbt eine Reihe von nützlichen Methoden, die in der folgenden Grafik sichtbar sind. Sie fasst gleichzeitig die Vererbungsbeziehungen noch einmal zusammen.

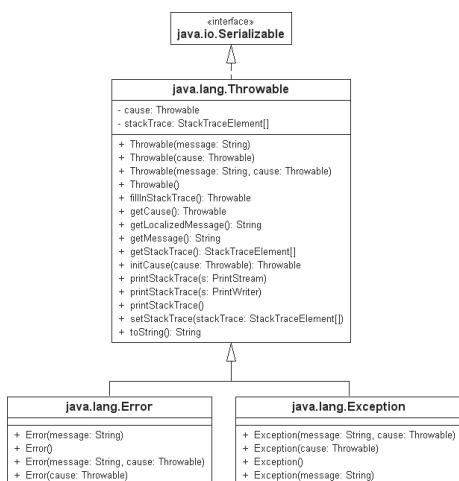


Abbildung 6.8: UML-Diagramm der wichtigen Oberklasse `Throwable`

6.3.1 Die Exception-Hierarchie

Jede Benutzausnahme wird von `java.lang.Exception` abgeleitet. Die Exceptions sind Fehler oder Ausnahmesituationen, die vom Programmierer behandelt werden sollen. Die Klasse `Exception` teilt sich dann nochmals in weitere Unterklassen beziehungsweise Unterhierarchien auf. Die folgende Grafik zeigt einige Unterklassen der Klasse `Exception`:

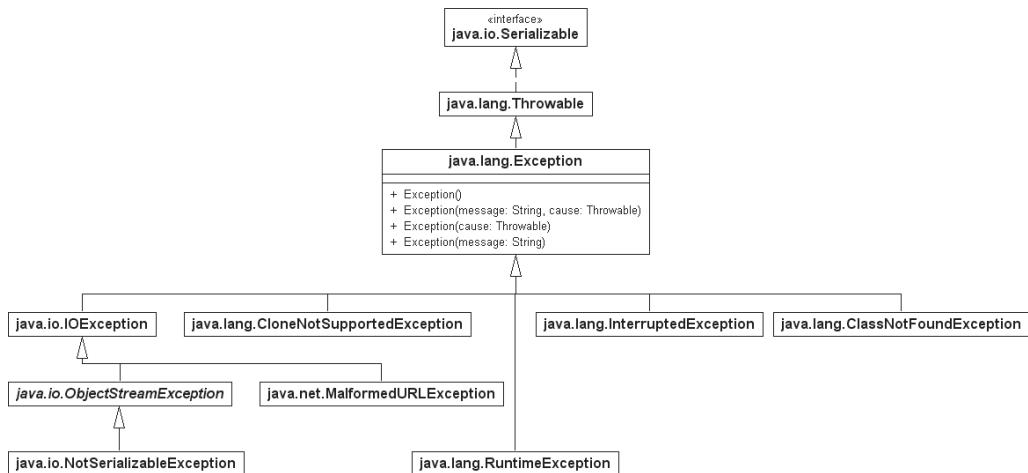


Abbildung 6.9: Ausgewählte Unterklassen von `Exception`

6.3.2 Oberausnahmen auffangen

Eine Konsequenz der Hierarchien besteht darin, dass es ausreicht, einen Fehler der Oberklasse aufzufangen. Wenn zum Beispiel eine `FileNotFoundException` auftritt, ist diese Klasse von `IOException` abgeleitet, was bedeutet, dass `FileNotFoundException` eine Spezialisierung darstellt. Wenn wir jede `IOException` auffangen, behandeln wir damit auch gleichzeitig die `FileNotFoundException` mit.

Erinnern wir uns noch einmal an das Dateibeispiel. Dort haben wir eine `FileNotFoundException` und eine `IOException` einzeln behandelt. Ist die Behandlung aber die gleiche, lässt sie sich wie folgt zusammenfassen:

Listing 6.12: ReadGifSizeShort.java, main()

```
RandomAccessFile f = null;
```

```
try
```

```

{
    f = new RandomAccessFile( "duke.gif", "r" );
    f.seek( 6 );

    System.out.printf( "%s x %s Pixel%n", f.read() + f.read() * 256,
                       f.read() + f.read() * 256 );
}
catch ( IOException e )
{
    System.err.println( "Allgemeiner Ein-/Ausgabefehler!" );
}
finally
{
    if ( f != null ) try { f.close(); } catch ( IOException e ) { }
}

```

Angst davor, dass wir den Fehlertyp später nicht mehr unterscheiden können, brauchen wir nicht zu haben, denn die an die `catch`-Anweisung gebundenen Variablen können wir mit `instanceof` weiter verfeinern. Aus Gründen der Übersichtlichkeit sollte diese Technik jedoch sparsam angewendet werden. Fehlerarten, die unterschiedlich behandelt werden müssen, verdienen immer getrennte `catch`-Klauseln. Das trifft zum Beispiel auf `FileNotFoundException` und `IOException` zu.

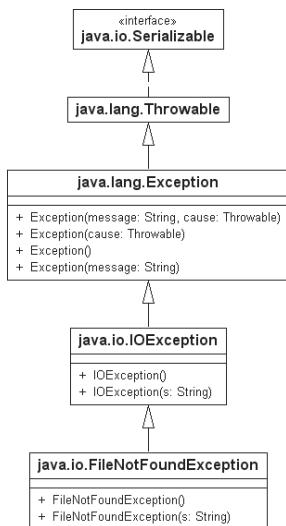


Abbildung 6.10: `IOException` im Klassendiagramm

6.3.3 Schon gefangen?

Der Java-Compiler prüft, ob Ausnahmen vielleicht schon in der Kette aufgefangen wurden, und meldet einen Fehler, wenn catch-Blöcke nicht erreichbar sind. Wir haben gesehen, dass `FileNotFoundException` eine spezielle `IOException` ist, und ein `catch(IOException e)` Fehler vom `FileNotFoundException` gleich mit fängt.

```
try
{
    ...
}
catch ( IOException e ) // fange IOException und alle Unterklassen auf
{
    ...
}
```

Natürlich kann eine `FileNotFoundException` weiterhin als eigener Typ aufgefangen werden, allerdings ist es wichtig, die Reihenfolge der catch-Blöcke zu beachten. Denn die Reihenfolge ist absolut relevant; die Typtests beginnen oben und laufen dann weiter nach unten durch. Wenn ein früher `catch` schon Ausnahmen eines gewissen Typs abfängt, also etwa ein `catch` auf `IOException` alle Ein-/Ausgabefehler, so ist ein nachfolgender `catch` auf die `FileNotFoundException` falsch.

Nehmen wir an, ein try-Block kann eine `FileNotFoundException` und eine `IOException` auslösen. Dann ist die linke Behandlung korrekt, aber die rechte falsch:

Richtig	Mit Compilerfehler
<pre>try { ... } catch (FileNotFoundException e) { } catch (IOException e) {}</pre>	<pre>try { ... } catch (IOException e) { } catch (FileNotFoundException e) // ☹</pre>

Tabelle 6.2: Die Reihenfolge der catch-Blöcke spielt eine Rolle.

6.3.4 Alles geht als Exception durch

Löst ein Programmblöck etwa eine `IOException`, `MalformedURLException` und eine `FileNotFoundException` aus, soll der Fehler aber gleich behandelt werden, so fängt ein `catch(IOException e)` die beiden Fehler `FileNotFoundException` und `MalformedURLException` gleich mit ab, da beide Unterklassen von `IOException` sind. So behandelt ein Block alle drei Fehlertypen. Das ist praktisch.

Nun gibt es jedoch auch Ausnahmen, die in der Vererbungsbeziehung nebeneinander liegen, etwa `SQLException` und `IOException`. Was ist, wenn die Ausnahmebehandlung gleich sein soll? Die naheliegende Idee ist, die Ausnahmehierarchie so weit nach oben zu laufen, bis eine gemeinsame Oberklasse gefunden wurde. Bei `SQLException` und `IOException` ist das `Exception` – sozusagen der kleinste gemeinsame Nenner. Also könnten Entwickler auf die Idee kommen, `Exception` aufzufangen und dort einmal den Fehler zu behandeln. Anstatt also einen Behandler zweimal zu schreiben und eine Codeduplizierung zu verursachen wie in

```
try
{
    irgendwas kann SQLException auslösen ...
    irgendwas kann IOException auslösen ...
}
catch ( SQLException e ) { Behandlung }
catch ( IOException e ) { Behandlung }
```

lässt sich aufgrund der identischen Fehlerbehandlungen eine Optimierung versuchen, die etwa so aussieht:

```
try
{
    irgendwas kann SQLException auslösen ...
    irgendwas kann IOException auslösen ...
}
catch ( Exception e ) { Behandlung }
```

Von dieser Lösung ist dringend abzuraten! Denn was für andere Fehlertypen gut funktionieren mag, ist für `catch(Exception e)` gefährlich, weil wirklich jede Ausnahme aufgefangen und in der Ausnahmebehandlung bearbeitet wird. Taucht beispielsweise eine

null-Referenz durch eine nicht initialisierte Variable mit Referenztyp auf, so würde dies fälschlicherweise ebenso behandelt; der Programmfehler hat aber nichts mit der SQLException oder IOException zu tun:

```
try
{
    Point p = null;
    p.x = 2;           // ☣ NullPointerException
    int i = 0;
    int x = 12 / i;   // ☣ Ganzzahlige Division durch 0

    irgendwas kann SQLException auslösen ...
    irgendwas kann IOException auslösen ...
}
catch ( Exception e ) { Behandlung }
```

Eine NullPointerException und die ArithmeticException sollen nicht mitbehandelt werden. Das zentrale Problem ist hier, dass diese Fehler ungeprüfte Ausnahmen vom Typ RuntimeException sind. RuntimeException ist eine Unterklasse von Exception. Fangen wir alle Exception-Typen, so wird alles mitgefangen – und RuntimeException eben auch. Es ist nicht möglich, alle Nicht-Laufzeitfehler abzufangen, was etwa funktionieren würde, wenn RuntimeException keine Unterklasse von Exception wäre, etwa ein Throwable – aber das haben die Sprachdesigner nicht so modelliert.

Wir werden gleich sehen, wie sich das Problem elegant lösen lässt.

Wenn main() alles weiterleitet

Ist die Fehlerbehandlung in einem Hauptprogramm ganz egal, so können wir alle Fehler auch an die Laufzeitumgebung weiterleiten, die dann das Programm – genau genommen den Thread – im Fehlerfall abbricht:

Listing 6.13: IDontCare.java, main()

```
public static void main( String[] args ) throws Exception
{
    RandomAccessFile f = new RandomAccessFile( "Datei.txt", "r" );
    System.out.println( f.readLine() );
}
```

Das funktioniert, da alle Fehler von der Klasse `Exception`³ abgeleitet sind. Wird der Fehler nirgendwo sonst aufgefangen, erfolgt die Ausgabe einer Laufzeitfehlermeldung, denn das `Exception`-Objekt ist beim Interpreter, also bei der virtuellen Maschine, auf der äußersten Aufrufebene gelandet. Natürlich ist das kein guter Stil – obwohl es aus Gründen kürzerer Programme auch in diesem Buch so gemacht wird. Denn Fehler sollten in jedem Fall behandelt werden.

6.3.5 Zusammenfassen gleicher catch-Blöcke mit dem multi-catch

Greift ein Programm auf Teile zurück, die scheitern können, so ergeben sich in komplexeren Abläufen schnell Situationen, in denen unterschiedliche Ausnahmen auftreten können. Entwickler sollten versuchen, den Programmcode in einem `try`-Block durchzuschreiben, und dann in `catch`-Blöcken auf alle möglichen Fehler zu reagieren, die den Block vom korrekten Durchlaufen abgehalten haben.

Oftmals kommt es zu dem Phänomen, dass die aufgerufenen Programmteile unterschiedliche Ausnahmetypen auslösen, aber die Behandlung der Fehler gleich aussieht. Um Quellcodeduplizierung zu vermeiden, sollte der Programmcode zusammengefasst werden. Nehmen wir an, die Behandlung der Ausnahmen `SQLException` und `IOException` soll gleich sein. Wir haben schon gesehen, dass ein `catch(Exception e)` keine gute Lösung ist und nie im Programmcode vorkommen sollte, denn dann würden auch andere Ausnahmen mitgefangen. Zum Glück gibt es in Java 7 eine elegante Lösung.

Multi-catch

Java 7 führt eine neue Schreibweise für `catch`-Anweisungen ein, um mehrere Ausnahmen auf einmal aufzufangen; sie heißt *multi-catch*. In der abgewandelten Variante von `catch` steht dann nicht mehr nur eine Ausnahme, sondern eine Sammlung von Ausnahmen, die ein »|« trennt. Der Schrägstrich ist schon als Oder-Operator bekannt und wurde daher auch hier eingesetzt, denn die Ausnahmen sind ja auch als eine Oder-Verknüpfung zu verstehen. Die allgemeine Syntax ist:

```
try
{
    ...
}
catch ( E1 | E2 | ... | En exception )
```

³ Genauer gesagt, sind alle Ausnahmen in Java von der `Exception`-Oberklasse `Throwable` abgeleitet.

Die Variable `exception` ist implizit final.

Um das *multi-catch* zu demonstrieren, nehmen wir ein Programm an, das eine Farbtabelle einliest. Die Datei besteht aus mehreren Zeilen, wobei in jeder Zeile die erste Zahl einen Index repräsentiert und die zweite Zahl den hexadezimalen RGB-Farbwert.

Listing 6.14: basiscolors.txt

```
0 000000
1 ff0000
8 00ff00
9 fffff0
```

Eine eigene Methode `readColorTable()` soll die Datei einlesen und ein `int`-Feld der Größe 256 als Rückgabe liefern, wobei an den in der Datei angegebenen Positionen jeweils die Farbwerte eingetragen sind. Nicht belegte Positionen bleiben 0. Gibt es einen Lauf Fehler, soll die Rückgabe `null` sein und die Methode eine Meldung auf dem Fehlerausgabekanal ausgeben.

Das Einlesen soll die `Scanner`-Klasse übernehmen. Bei der Verarbeitung der Daten und der Füllung des Feldes sind diverse Fehler denkbar:

- `IOException`: Die Datei ist nicht vorhanden, oder während des Einlesens kommt es zu Problemen.
- `InputMismatchException`: Der Index oder die Hexadezimalzahl sind keine Zahlen (einmal zur Basis 10, und dann zur Basis 16). Den Fehlertyp löst der `Scanner` aus.
- `ArrayIndexOutOfBoundsException`: Der Index liegt nicht im Bereich von 0 bis 255.

Während der ersten Fehler beim Dateisystem zu suchen ist, sind die zwei unteren Fehler – unabhängig davon, dass sie ungeprüfte Ausnahmen sind – auf ein fehlerhaftes Format zurückzuführen. Die Behandlung soll immer gleich aussehen und kann daher gut in einem *multi-catch* zusammengefasst werden. Daraus folgt:

Listing 6.15: ReadColorTable.java

```
import java.io.*;
import java.util.Scanner;

public class ReadColorTable
{
    private static int[] readColorTable( String filename )
    {
```

```
Scanner input;
int[] colors = new int[ 256 ];
try
{
    input = new Scanner( new File(filename) );
    while ( input.hasNextLine() )
    {
        int index = input.nextInt();
        int rgb   = input.nextInt( 16 );
        colors[ index ] = rgb;
    }
    return colors;
}
catch ( IOException e )
{
    System.err.printf( "Dateioperationen fehlgeschlagen%n%s%n", e );
}
catch ( InputMismatchException | ArrayIndexOutOfBoundsException e )
{
    System.err.printf( "Datenformat falsch%n%s%n", e );
}
finally
{
    input.close();
}
return null;
}

public static void main( String[] args )
{
    readColorTable( "basiscolors.txt" );
}
```

Multi-catch-Blöcke sind also eine Abkürzung, und der Bytecode sieht genauso aus wie mehrere gesetzte catch-Blöcke, also wie:

```

catch ( InputMismatchException e )
{
    System.err.printf( "Datenformat falsch%n%s%n", e );
}
catch ( ArrayIndexOutOfBoundsException e )
{
    System.err.printf( "Datenformat falsch%n%s%n", e );
}

```

Multi-catch-Blöcke sind nur eine Abkürzung, daher teilen sie auch die Eigenschaften der normalen catch-Blöcke. Der Compiler führt die gleichen Prüfungen wie bisher durch, also ob etwa die genannten Ausnahmen im try-Block überhaupt ausgelöst werden können. Nur das, was in der durch »|« getrennten Liste aufgezählt ist, wird behandelt; unser Programm fängt zum Beispiel nicht generisch alle RuntimeExceptions ab. Und genauso dürfen die in catch oder *multi-catch* genannten Ausnahmen nicht in einem anderen (multi)-catch auftauchen.

Neben den Standard-Tests kommen neue Überprüfungen hinzu, ob etwa die exakt gleiche Exception zweimal in der Liste ist oder ob es Widersprüche durch Mengenbeziehungen gibt.



Hinweis

Der folgende *multi-catch* ist falsch:

```

try
{
    new RandomAccessFile("", "");
}
catch ( FileNotFoundException | IOException | Exception e ) { }

```

Der *javac*-Compiler meldet einen Fehler der Art »Alternatives in a multi-catch statement cannot be related by subclassing« und bricht ab.

Mengenprüfungen führt der Compiler auch ohne *multi-catch* durch, und Folgendes ist ebenfalls falsch:

```

try { new RandomAccessFile("", ""); }
catch ( Exception e ) { }
catch ( IOException e ) { }
catch ( FileNotFoundException e ) { }

```

Hinweis (Forts.)

Während allerdings eine Umsortierung der Zeilen die Fehler korrigiert – wie in Abschnitt 6.3.3 erwähnt –, spielt die Reihenfolge bei multi-catch keine Rolle.

6

6.4 Harte Fehler: Error *

Fehler, die von der Klasse `java.lang.Error` abgeleitet sind, stellen Fehler dar, die mit der JVM in Verbindung stehen. Anders reagieren dagegen die von `Exception` abgeleiteten Klassen – sie stehen für eigene Programmfehler. Beispiele für konkrete `Error`-Klassen sind:

- `AnnotationFormatError`
- `AssertionError`
- `AWTError`
- `CoderMalfunctionError`⁴
- `FactoryConfigurationError` (XML-Fehler)
- `IOWorker`
- `LinkageError` (mit vielen Unterklassen)
- `ThreadDeath, TransformerFactoryConfigurationError` (XML-Fehler)
- `VirtualMachineError` (mit den Unterklassen `InternalError, OutOfMemoryError, StackOverflowError, UnknownError`)

Im Fall von `ThreadDeath` lässt sich ableiten, dass nicht alle `Error`-Klassen auf »Error« enden. Das liegt sicherlich auch daran, dass das nicht ein Fehler im eigentlichen Sinne ist, denn die JVM löst `ThreadDeath` aus, wenn das Programm einen Thread mit `stop()` beenden will.

Da ein `Error` »abnormales« Verhalten anzeigt, müssen Operationen, die einen solchen Fehler auslösen können, auch nicht in einem `try`-Block sitzen oder mit `throws` nach oben weitergegeben werden (`Error`-Fehler zählen zu den nicht geprüften Ausnahmen, obwohl `Error` keine Unterklasse von `RuntimeException` ist!). Allerdings ist es möglich, die Fehler aufzufangen, da `Error`-Klassen Unterklassen von `Throwable` sind und sich daher genauso behandeln lassen. Insofern ist ein Auffangen legitim, und auch ein `finally` ist korrekt.

⁴ Die lustigste Fehlerklasse, wie ich finde. Sie könnte bei einigen Entwicklern bei jeder Methode ausgelöst werden.

Ob das Auffangen sinnvoll ist, ist eine andere Frage, denn wenn die JVM einen Fehler anzeigt, bleibt offen, wie darauf sinnvoll zu reagieren ist. Was sollten wir bei einem `LinkageError` tun? Einen `OutOfMemoryError` in bestimmten Programmteilen aufzufangen, kann jedoch von Vorteil sein. Eigene Unterklassen von `Error` sollten keine Anwendung finden. Glücklicherweise sind die Klassen aber nur Unterklassen von `Throwable` und nicht von `Exception`, sodass ein `catch(Exception e)` nicht aus versehen Dinge wie `ThreadDeath` abfängt, die eigentlich nicht behandelt gehören.

6.5 Auslösen eigener Exceptions

Bisher wurden Exceptions lediglich aufgefangen, aber noch nicht selbst erzeugt. In diesem Abschnitt wollen wir sehen, wie eigene Ausnahmen ausgelöst werden. Das kann zum einen erfolgen, wenn die JVM provoziert wird, etwa bei einer ganzzahligen Division durch 0 oder explizit durch `throw`.

6.5.1 Mit `throw` Ausnahmen auslösen

Soll eine Methode oder ein Konstruktor selbst eine Exception auslösen, muss zunächst ein Exception-Objekt erzeugt und dann die Ausnahmebehandlung angestoßen werden. Im Sprachschatz dient das Schlüsselwort `throw` dazu, eine Ausnahme zu signalisieren und die Abarbeitung an der Stelle zu beenden.

Als Exception-Typ soll im folgenden Beispiel `IllegalArgumentException` dienen, das ein fehlerhaftes Argument anzeigt:

Listing 6.16: com/tutego/insel/exceptions/v1/Player.java, Konstruktor

```
Player( int age )
{
    if ( age <= 0 )
        throw new IllegalArgumentException( "Kein Alter <= 0 erlaubt!" );

    this.age = age;
}
```

Wir sehen im Beispiel, dass negative oder null Alter-Übergaben nicht gestattet sind und zu einem Fehler führen. Im ersten Schritt baut dazu der new-Operator das Exception-Objekt über einen parametrisierten Konstruktor auf. Die Klasse `IllegalArgumentException` bietet einen solchen Konstruktor, der eine Zeichenkette annimmt, die den näheren Grund der Ausnahme übermittelt. Welche Parameter die einzelnen Exception-Klassen deklarieren, ist der API zu entnehmen. Nach dem Aufbau des Exception-Objekts beendet `throw` die lokale Abarbeitung, und die JVM sucht ein `catch`, das die Ausnahme behandelt.

Hinweis

Ein `throws` `IllegalArgumentException` am Konstruktor ist in diesem Beispiel überflüssig, da `IllegalArgumentException` eine `RuntimeException` ist, die nicht über ein `throws` in der Methoden-Signatur angegeben werden muss.

Lassen wir ein Beispiel folgen, in dem Spieler mit einem negativen Alter initialisiert werden sollen:

Listing 6.17: com/tutego/insel/exceptions/v1/Player.java, main()

```
try
{
    Player d = new Player( -100 );
    System.out.println( d );
}
catch ( IllegalArgumentException e )
{
    e.printStackTrace();
}
```

Das führt zu einer Exception, und der Stack-Trace, den `printStackTrace()` ausgibt, ist:

```
java.lang.IllegalArgumentException: Kein Alter <= 0 erlaubt!
    at com.tutego.insel.exceptions.v1.Player.<init>(Player.java:10)
    at com.tutego.insel.exceptions.v1.Player.main(Player.java:19)
```

Hinweis

Löst ein Konstruktor eine Ausnahme aus, ist eine Nutzung wie die folgende problematisch:

**Hinweis (Forts.)**

```
Player p = null;
try
{
    p = new Player( v );
}
catch ( IllegalArgumentException e ) { }
p.getAge(); // BUMM: ☣ NullPointerException
```

Die Exception führt zu keinem Player-Objekt, wenn v negativ ist. So bleibt p mit null vorbelegt. Es folgt in der BUMM-Zeile eine NullPointerException. Der Programmcode, der das Objekt erwartet, aber vielleicht mit einer null rechnet, sollte mit in den try-Block. Doch üblicherweise stehen solche Fehler für Programmierfehler und werden nicht aufgefangen.

Da die `IllegalArgumentException` eine `RuntimeException` ist, hätte es in `main()` auch ohne try-catch so heißen können:

```
public static void main( String[] args )
{
    Player d = new Player( -100 );
}
```

Die `Runtime-Exception` müsste nicht zwingend aufgefangen werden, aber der Effekt wäre, dass die Ausnahme nicht behandelt würde und das Programm abbräche.

```
class java.lang.IllegalArgumentException
extends RuntimeException
```

- `IllegalArgumentException()`
Erzeugt eine neue Ausnahme ohne genauere Fehlerangabe.
- `IllegalArgumentException(String s)`
Erzeugt ein neues Fehler-Objekt mit einer detaillierteren Fehlerangabe.

6.5.2 Vorhandene Runtime-Fehlertypen kennen und nutzen

Die Java-API bietet eine große Anzahl von Exception-Klassen, und so muss nicht für jeden Fall eine eigene Exception-Klasse deklariert werden. Viele Standard-Fälle, wie falsche Argumente, können durch Standard-Exception-Klassen abgedeckt werden.

Hinweis

Entwickler sollten nie `throw new Exception()` oder sogar `throw new Throwable()` schreiben, sondern sich immer konkreter Unterklassen bedienen.

6

Einige Standard-Runtime-Exception-Unterklassen des `java.lang`-Pakets in der Übersicht:

IllegalArgumentException

Die `IllegalArgumentException` zeigt an, dass ein Parameter nicht korrekt angegeben ist. Dieser Fehlertyp lässt sich somit nur bei Konstruktoren oder Methoden ausmachen, denen fehlerhafte Argumente übergeben wurden. Oft ist der Grund die Missachtung des Wertebereiches. Wenn die Werte grundsätzlich korrekt sind, darf dieser Fehlertyp nicht ausgelöst werden. Dazu folgen gleich noch ein paar mehr Details.

IllegalStateException

Objekte haben in der Regel Zustände. Gilt es, Operationen auszuführen, aber die Zustände sind nicht korrekt, so kann die Methode eine `IllegalStateException` auslösen und so anzeigen, dass in dem aktuellen Zustand die Operation nicht möglich ist. Wäre der Zustand korrekt, käme es nicht zu der Ausnahme. Bei statischen Methoden sollte es eine `IllegalStateException` nicht geben.⁵

UnsupportedOperationException

Implementieren Klassen Schnittstellen oder realisieren Klassen abstrakte Methoden von Oberklassen, so muss es immer eine Implementierung geben, auch wenn die Unterklasse die Operation eigentlich gar nicht umsetzen kann oder will. Anstatt den Rumpf der Methode nur leer zu lassen und einen potenziellen Aufrufer glauben zu lassen, die Methode führe etwas aus, sollten diese Methoden eine `UnsupportedOperationException`

⁵ Im .NET-Framework gibt es eine vergleichbare Ausnahme, die `System.InvalidOperationException`. In Java trifft der Name allerdings das Problem etwas besser.

auslösen. In den API-Dokumentationen werden Methoden, die Unterklassen vielleicht nicht realisieren wollen, als *optionale Operationen* gekennzeichnet.

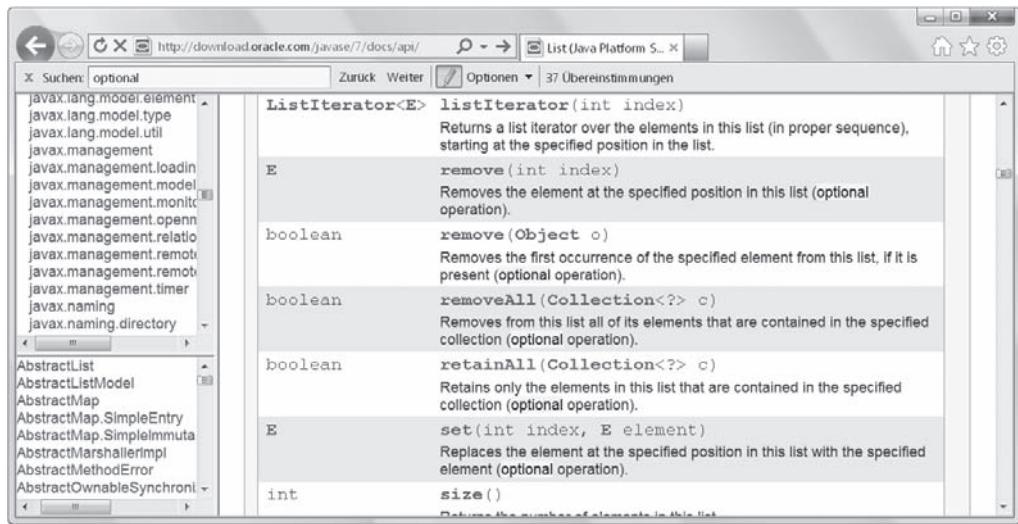


Abbildung 6.11: Optionale Operationen in der Schnittstelle `java.util.List`

Unglücklicherweise gibt es auch eine `javax.naming.OperationNotSupportedException`. Doch diese sollte nicht verwendet werden. Sie ist speziell für Namensdienste vorgesehen und auch keine `RuntimeException`.

IndexOutOfBoundsException

Eine `IndexOutOfBoundsException` löst die JVM automatisch aus, wenn zum Beispiel die Grenzen eines Arrays missachtet werden. Wir können diesen Ausnahmetyp selbst immer dann nutzen, wenn wir Index-Zugriffe haben, etwa auf eine Zeile in einer Datei, und wenn der Index im falschen Bereich liegt. Von `IndexOutOfBoundsException` gibt es die Unterklassen `ArrayIndexOutOfBoundsException` und `StringIndexOutOfBoundsException`. Programmierer werden diese Typen aber in der Regel nicht nutzen. Inkonsistenzen gibt es beim Einsatz von `IllegalArgumentException` und `IndexOutOfBoundsException`. Ist etwa der Index falsch, so entscheiden sich einige Autoren für den ersten Fehlertyp, andere für den zweiten. Beides ist prinzipiell gültig. Die `IndexOutOfBoundsException` ist aber konkreter und zeigt eher ein Implementierungsdetail an.

Keine eigene NullPointerException auslösen

Eine `NullPointerException` gehört mit zu den häufigsten Ausnahmen. Die JVM löst diesen Fehler etwa bei folgendem Programmstück aus:

```
String s = null;  
s.length();           // ☣ NullPointerException
```

Eine `NullPointerException` zeigt immer einen Programmierfehler in einem Stück Code an, und so hat es in der Regel keinen Sinn, diesen Fehler abzufragen – der Programmierfehler muss behoben werden. Aus diesem Grund wird eine `NullPointerException` in der Regel nie explizit vom Programmierer ausgelöst, sondern von der JVM. Sie kann jedoch vom Entwickler bewusst ausgelöst werden, wenn eine zusätzliche Nachricht Klarheit verschaffen soll.

Oft gibt es diese `NullPointerException`, wenn an Methoden `null`-Werte übergeben wurden. Hier muss aus der API-Dokumentation klar hervorgehen, ob `null` als Argument erlaubt ist oder nicht. Wenn nicht, ist es völlig in Ordnung, wenn die Methode eine `NullPointerException` auslöst, wenn fälschlicherweise doch `null` übergeben wurde. Auf `null` zu prüfen, um dann zum Beispiel eine `IllegalArgumentException` auszulösen, ist eigentlich nicht nötig. Allerdings gilt auch hier, dass `IllegalArgumentException` allgemeiner und weniger implementierungsspezifisch als eine `NullPointerException` ist.

Hinweis

Um eine `NullPointerException` auszulösen, ist statt `throw new NullPointerException();` auch einfach ein `throw null;` möglich. Doch da eine selbst aufgebaute `NullPointerException` vermieden werden sollte, ist dieses Idiom nicht wirklich nützlich.

6.5.3 Parameter testen und gute Fehlermeldungen

Eine `IllegalArgumentException` ist eine wertvolle Ausnahme, die einen internen Fehler anzeigen: dass nämlich eine Methode mit falschen Argumenten aufgerufen wurde. Eine Methode sollte im Idealfall alle Parameter auf ihren korrekten Wertebereich prüfen und nach dem *Fail-fast-Verfahren* arbeiten, also so schnell wie möglich einen Fehler melden, anstatt Fehler zu ignorieren oder zu verschleppen. Wenn etwa das Alter einer Person bei `setAge()` nicht negativ sein kann, ist eine `IllegalArgumentException` eine gute Wahl. Wenn der Exception-String dann noch aussagekräftig ist, hilft das bei der Behebung des Fehlers ungemein: Der Tipp ist hier, eine aussagekräftige Meldung anzugeben.



Negativbeispiel

Ist der Wertebereich beim Bilden eines Teilstrings falsch oder ist der Index für einen Feldzugriff zu groß, hagelt es eine `StringIndexOutOfBoundsException` bzw. `ArrayIndexOutOfBoundsException`:

```
System.out.println( "Orakel-Paul".substring( 0, 20 ) ); // ☠
```

liefert:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:  
String index out of range: 20
```

Und

```
System.out.println( "Orakel-Paul".toCharArray()[20] ); // ☠
```

liefert:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 20
```

Eine wichtige Information fehlt allerdings! Wie groß ist denn der String bzw. das Feld? Java-Entwickler warten seit über zehn Jahren auf diese Information.

Da das Testen von Parametern in eine große `if-throws`-Orgie ausarten kann, ist es eine gute Idee, eine Hilfsklasse mit statischen Methoden wie `isNull()`, `isFalse()`, `isInRange()` einzuführen, die dann eine `IllegalArgumentException` auslösen, wenn eben der Parameter nicht korrekt ist.⁶

null-Prüfungen

Für null-Prüfungen führt Java 7 mit `Objects.requireNonNull(reference)` eine Methode ein, die immer dann eine `IllegalArgumentException` auslöst, wenn `reference == null` ist. Optional als zweites Argument lässt sich die Fehlermeldung angeben.

⁶ Die müssen wir nicht selbst schreiben, da die Open-Source-Landschaft bereits mit der Klasse `org.apache.commons.lang.Validate` aus den Apache Commons Lang (<http://commons.apache.org/lang/>) oder mit `com.google.common.base.Preconditions` von Google Guava (<http://code.google.com/p/guava-libraries/>) schon Vergleichbares bietet; in jedem Fall ist eine gute Parameterprüfung bei öffentlichen Methoden von Bibliotheken ein Muss.

Tool-Unterstützung

Eine anderer Ansatz sind Prüfungen durch externe Codeprüfungsprogramme. Google zum Beispiel setzt in seinen vielen Java-Bibliotheken auf Parameter-Annotationen wie @Nonnull oder @Nullable.⁷ Statische Analysetools wie *FindBugs* (<http://findbugs.sourceforge.net/>) testen dann, ob es Fälle geben kann, in denen die Methode mit null aufgerufen wird. Zur Laufzeit findet der Test jedoch nicht statt.



6.5.4 Neue Exception-Klassen deklarieren

Eigene Exceptions sind immer direkte (oder indirekte) Unterklassen von `Exception` (sie können auch Unterklassen von `Throwable` sein, aber das ist unüblich). Eigene Exception-Klassen bieten in der Regel zwei Konstruktoren: einen Standard-Konstruktor und einen mit einem String parametrisierten Konstruktor, um eine Fehlermeldung (die `Exception-Message`) anzunehmen und zu speichern.

Um für die Klasse `Player` im letzten Beispiel einen neuen Fehlertyp zu deklarieren, erweitern wir `RuntimeException` zur `PlayerException`:

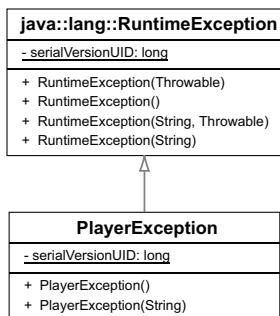


Abbildung 6.12: UML-Diagramm für `PlayerException`

Listing 6.18: com/tutego/insel/exception/v2/PlayerException.java

```

package com.tutego.insel.exception.v2;

public class PlayerException extends RuntimeException
{
  
```

⁷ Sie wurden in JSR-305, »Annotations for Software Defect Detection«, definiert. Java 7 sollte dies ursprünglich unterstützen, doch das wurde gestrichen.

```

public PlayerException()
{
}

public PlayerException( String s )
{
    super( s );
}
}

```

Nehmen wir uns die Initialisierung mit dem Alter noch einmal vor. Statt der `IllegalArgument`Exception löst der Konstruktor im Fehlerfall unsere speziellere `PlayerException` aus:

Listing 6.19: com/tutego/insel/exception/v2/Player.java, Ausschnitt

```

if ( age <= 0 )
    throw new PlayerException( "Kein Alter <= 0 erlaubt!" );

```

Im Hauptprogramm können wir auf die `PlayerException` reagieren, indem wir die Ausnahme explizit mit try-catch auffangen oder an den Aufrufer weitergeben – unsere Exception ist ja eine `RuntimeException` und müsste nicht direkt abgefangen werden:

```

Exception in thread "main" com.tutego.insel.exceptions.v2.PlayerException: Kein Alter
<= 0 erlaubt!
at com.tutego.insel.exceptions.v2.Player.<init>(Player.java:10)
at com.tutego.insel.exceptions.v2.Player.main(Player.java:19)

```



Tipp

Es ist immer eine gute Idee, Unterklassen von `Exception` zu bauen. Würden wir keine Unterklassen anlegen, sondern direkt mit `throw new Exception()` einen Fehler anzeigen, so könnten wir unseren Fehler später nicht mehr von anderen Fehlern unterscheiden. Mit der Hierarchiebildung wird nämlich die Spezialisierung bei mehreren catch-Anweisungen sowie eine Unterscheidung mit `instanceof` unterstützt. Wir müssen immer unseren Fehler mit `catch(Exception e)` auffangen und bekommen so alle anderen Fehler mit aufgefangen, die dann nicht mehr unterschieden werden können. Allerdings sollten Entwickler nicht zu inflationär mit den Ausnahmen-Hierarchien umgehen; in vielen Fällen reicht eine Standard-Ausnahme aus.

6.5.5 Eigene Ausnahmen als Unterklassen von Exception oder RuntimeException?

Java steht mit der Ausnahmebehandlung über Exceptions nicht allein. Alle modernen Programmiersprachen verfügen über diese Sprachmittel. Allerdings gibt es eine Sache, die Java besonders macht: Die Unterscheidung zwischen geprüften und ungeprüften Ausnahmen. Daher stellt sich beim Design von eigenen Ausnahmenklassen die Frage, ob sie eine Unterklasse von `RuntimeException` sein sollen oder nicht. Einige Entscheidungshilfen:

- Betrachten wir, wie die Java-API *geprüfte* und *ungeprüfte Ausnahmen* einsetzt. Die ungeprüften Ausnahmen signalisieren Programmierfehler, die es zu beheben gilt. Ein gutes Beispiel ist eine `NullPointerException`, `ClassCastException` oder `ArrayIndexOutOfBoundsException`. Es steht außer Frage, dass Fehler dieser Art Programmierfehler sind und behoben werden müssen. Ein `catch` wäre unnötig, da die Fehler ja im korrekten Code gar nicht auftreten können. Anders ist es bei geprüften Ausnahmen. Die Ausnahmen zeigen Fehler an, die unter gewissen Umständen einfach auftreten können. Eine `IOException` ist nicht schlimmer, denn die Datei kann nun einmal nicht vorhanden sein. Wir sollten uns bei dieser Unterscheidung aber bewusst sein, dass die JVM die Fehler von sich aus auslöst und nicht eine Methode.
- Soll sich die Anwendung von dem Fehler »erholen« können oder nicht? Kommt es wegen einer `RuntimeException` zu einem Programmfehler, dann sollte die Anwendung zwar nicht »abstürzen«, allerdings ist ein sinnvolles Weiterarbeiten kaum möglich. Bei geprüften Ausnahmen ist das anders. Sie signalisieren, dass der Fehler behoben und das Programm dann normal fortgesetzt werden kann.
- Ein Modul kann intern mit `RuntimeExceptions` arbeiten, und der API-Designer auf der anderen Seite, der Schnittstellen zu Systemen modelliert, kann gut auf geprüfte Ausnahmen zurückgreifen. Das ist einer der Gründe, warum moderne Frameworks wie `EJB 3` oder auch `Spring` fast ausschließlich auf eine `RuntimeException` setzen: Wenn es einen Fehler gibt, dann lässt sich schwer etwas behandeln und einfach korrigieren. Zeigt etwa ein internes Modul beim Datenbankzugriff einen Fehler an, muss die ganze Operation abgebrochen werden, und nichts ist zu retten. Hier gilt im Großen, was auch bei der `NullPointerException` im Kleinen passiert: Der Fehler ist ein echtes Problem, und das Programm kann nicht einfach fortgeführt werden.
- Geprüfte Ausnahmen können melden, wenn sich der Aufrufer nicht an die Vereinbarung der Methode hält. Die `FileNotFoundException` ist so ein Beispiel. Hätte das Programm mit der `File`-Methode `exists()` vorher nach der Existenz der Datei gefragt, wäre uns diese Ausnahme erspart geblieben. Der Aufrufer ist sozusagen selbst schuld, dass er eine geprüfte Ausnahme bekommt, da er die Rahmenbedingungen nicht einhält. Bei einer ungeprüften Ausnahme ist nicht der Aufrufer an dem Problem schuld,

sondern ein Programmierfehler. Da geprüfte Ausnahmen in der Java-Dokumentation auftauchen, ist dem Entwickler klar, was passieren wird, wenn er die Vorbedingungen der Methode nicht einhält. Nach dieser Philosophie müsste eigentlich die `NumberFormatException` eine geprüfte Ausnahme sein, die `Integer.parseInt()` auslöst. Denn der Entwickler hat ja die `parseInt()` mit einem falschen Wert gefüttert, also den Methodenvertrag verletzt. Eine geprüfte Ausnahme wäre nach dieser Philosophie richtig. Auf der anderen Seite lässt sich argumentieren, dass das Missachten von korrekten Parametern ein interner Fehler ist, denn es ist Aufgabe des Aufrufers, das sicherzustellen, und so kann die `parseInt()` mit einer `RuntimeException` aussteigen.

- Die Unterscheidung zwischen *internen Fehlern* und *externen Fehlern* erlaubt eine Einteilung in geprüfte und ungeprüfte Ausnahmen. Die Programmierfehler mit Ausnahmen (wie `NullPointerException` oder `ClassCastException`) lassen sich vermeiden, da wir als Programmierer unseren Quellcode kontrollieren können und die Programmfehler entfernen können. Doch bei externen Fehlern haben wir als Entwickler keine Chance. Das Netzwerk kann plötzlich zusammenbrechen und uns eine `SocketException` und `IOException` bescheren. Alles das liegt nicht in unserer Hand und kann auch durch noch so sorgsame Programmierung nicht verhindert werden. Das schwächt natürlich das Argument aus dem letzten Aufzählungspunkt ab: Es lässt sich zwar abfragen, ob eine Datei vorhanden ist, um eine `FileNotFoundException` abzuwehren, doch wenn die Festplatte plötzlich Feuer fängt, ist uns eine `IOException` gewiss, denn Java-Programme sind nicht wie folgt aufgebaut: »Frage, ob die Festplatte bereit ist, und dann lies.« Wenn der Fehler also nicht innerhalb des Programms liegt, sondern außerhalb, lassen sich geprüfte Ausnahmen verwenden.
- Bei geprüften Ausnahmen in Methodensignaturen muss sich der Nutzer auf eine bestimmte API einstellen. Eine spätere Änderung des Ausnahmetyps ist problematisch, da alle catch-Anweisungen beziehungsweise throws-Klauseln abgeändert werden müssen. `RuntimeExceptions` sind hier flexibler. Werden Programme sehr agil entworfen und ändert sich der Ausnahmetyp im Lebenslauf einer Software öfter, kann das zu vielen Änderungen führen, die natürlich Zeit und somit Geld kosten.

Der erste Punkt führt in der Java-API zu einigen Entscheidungen, die Entwickler quälen, aber nur konsistent sind, etwa die `InterruptedException`. Jedes `Thread.sleep()` zum Schläfenlegen eines Threads muss eine `InterruptedException` auffangen. Sie kann auftreten, wenn ein Thread von außen einen `Interrupt` sendet. Da das auf keinen Fall einen Fehler darstellt, ist `InterruptedException` eine geprüfte Ausnahme, auch wenn wir dies oft als lästig empfinden und selten auf die `InterruptedException` reagieren müssen. Bei einem

Aufbau einer URL ist die `MalformedURLException` ebenfalls lästig, aber stammt die Eingabe aus einer Dialogbox, kann das Protokoll einfach falsch sein.⁸

Geprüfte Ausnahmen sind vielen Entwicklern lästig, was zu einem Problem führt, wenn die Ausnahmen einfach aufgefangen werden, aber nichts passiert – etwa mit einem leeren `catch`-Block. Der Fehler sollte aber vielleicht nach oben laufen. Das Problem besteht bei einer `RuntimeException` seltener, da sie in der Regel an der richtigen zentralen Stelle behandelt wird.

Wenn wir die Punkte genauer betrachten, dann wird schnell eine andere Tatsache klar, sodass heute eine große Unsicherheit über die richtige Exception-Basisklasse besteht. Zwar zwingt uns der Compiler, eine geprüfte Exception zu behandeln, aber nichts spricht dagegen, das bei einer ungeprüften Ausnahme ebenfalls zu tun. `Integer.parseInt()` und `NumberFormatException` sind ein gutes Beispiel: Der Compiler zwingt uns nicht zu einem Test, wir machen ihn aber trotzdem. Sind Entwickler konsequent und prüfen sie Ausnahmen selbstständig, braucht der Compiler den Test prinzipiell nicht zu machen. Daher folgen einige Entwickler einer radikalen Strategie und entwerfen alle Ausnahmen als `RuntimeException`. Die Unterscheidung, ob sich eine Anwendung dann »erholen« soll oder nicht, liegt beim Betrachter und ist nur noch reine Konvention. Mit dieser Alles-ist-ungeprüft-Version würde dann Java gleichauf mit C#, C++, Python, Groovy, ... liegen.⁹

6.5.6 Ausnahmen abfangen und weiterleiten *

Die Ausnahme, die ein `catch`-Block auffängt, kann mit einem `throw` wieder neu ausgelöst werden – das nennt sich *rethrow*. Ein Beispiel soll die Arbeitsweise verdeutlichen. Eine Hilfsmethode `createUriFromHost()` setzt vor einen Hostnamen "http://" und liefert das Ergebnis als URI-Objekt zurück. `createUriFromHost("tutego.de")` liefert somit eine URI mit `http://tutego.de`. Ist der Hostname aber falsch, löst der Konstruktor der `URI`-Klasse eine Ausnahme aus.

Listing 6.20: Rethrow.java

```
import java.net.*;
```

```
public class Rethrow
```

⁸ Luxuriös wäre eine Prüfung zur Compilierzeit, denn wenn etwa `new URL("http://tutego.de")` im Code steht, so kann es die Ausnahme nicht geben. Doch von nötigen `try-catch`-Blöcken, je nachdem, was der Compiler statisch entscheiden kann, sind wir weit entfernt.

⁹ Doch eines ist sicher: Java-Vater James Gosling ist dagegen: <http://www.artima.com/intv/solid.html>.

```

{
    public static URI createUriFromHost( String host ) throws URISyntaxException
    {
        try
        {
            return new URI( "http://" + host );
        }
        catch ( URISyntaxException e )
        {
            System.err.println( "Hilfe! " + e.getMessage() );

            throw e;
        }
    }

    public static void main( String[] args )
    {
        try
        {
            createUriFromHost( "tutego.de" );
            createUriFromHost( "%" );
        }
        catch ( URISyntaxException e )
        {
            e.printStackTrace();
        }
    }
}

```

Die Klasse `URI` testet die Strings genauer als die `URL`-Klasse, sodass wir in diesem Beispiel `URI` nutzen. Die Ausnahmen im Fehlerfall sind auch etwas anders; `URISyntaxException` ist die Ausnahme bei `URI`; `MalformedURLException` ist die Ausnahme bei `URL`. Genau diesen Fehler provozieren wir, indem wir dem Konstruktor ein "http://#" übergeben, was eine offensichtlich falsche `URI` ist. Unsere Methode wird die `URISyntaxException` auffangen, den Fehler auf der Standardfehlerausgabe melden und dann weiterleiten, denn wirklich

behandeln kann unsere Methode das Problem nicht; sie kann nur melden, was ein Vorteil ist, wenn der Aufrufer dies nicht tut.

Die Programmausgabe ist:

```
Hilfe! Malformed escape pair at index 7: http://%
java.net.URISyntaxException: Malformed escape pair at index 7: http://%
    at java.net.URI$Parser.fail(URI.java:2827)
    at java.net.URI$Parser.scanEscape(URI.java:2957)
    at java.net.URI$Parser.scan(URI.java:2980)
    at java.net.URI$Parser.parseAuthority(URI.java:3121)
    at java.net.URI$Parser.parseHierarchical(URI.java:3076)
    at java.net.URI$Parser.parse(URI.java:3032)
    at java.net.URI.<init>(URI.java:595)
    at Rethrow.createUriFromHost(Rethrow.java:9)
    at Rethrow.main(Rethrow.java:24)
```

6.5.7 Aufrufstack von Ausnahmen verändern *

Wenn wir in einer Ausnahmebehandlung eine Exception e auffangen und genau diese dann mit throw e weiterleiten, müssen wir uns bewusst sein, dass die Ausnahme e auch den Aufrufstack weitergibt. Aus dem vorangehenden Beispiel:

```
java.net.URISyntaxException: Malformed escape pair at index 7: http://%
    at java.net.URI$Parser.fail(URI.java:2827)
    at java.net.URI$Parser.scanEscape(URI.java:2957)
...
    at java.net.URI.<init>(URI.java:595)
    at Rethrow.createUriFromHost(Rethrow.java:9)
    at Rethrow.main(Rethrow.java:24)
```

Die main()-Methode fängt den Fehler von createUriFromHost() ab, aber diese Methode steht nicht ganz oben im Aufrufstack. Die Ausnahme stammte ja gar nicht von createUriFromHost() selbst, sondern von fail(), sodass fail() oben steht. Ist das nicht gewünscht, kann es korrigiert werden, denn die Basisklasse für alle Ausnahmen Throwable bietet die Methode fillInStackTrace(), mit der sich der Aufrufstack neu füllen lässt. Unsere bekannte Methode createUriFromHost() soll auf fillInStackTrace() zurückgreifen:

Listing 6.21: RethrowWithFillInStackTrace.java, createUriFromHost()

```

public static URI createUriFromHost( String host ) throws URISyntaxException
{
    try
    {
        return new URI( "http://" + host );
    }
    catch ( URISyntaxException e )
    {
        System.err.println( "Hilfe! " + e.getMessage() );
        e.fillInStackTrace();
        throw e;
    }
}

```

Kommt es in `createUriFromHost()` zur `URISyntaxException`, so fängt unsere Methode diese ab. Ursprünglich ist in `e` der Aufrufstack mit der `fail()`-Methode ganz oben gespeichert, allerdings löscht `fillInStackTrace()` zunächst den ganzen Stacktrace und füllt ihn neu mit dem Pfad, den der aktuelle Thread zu der Methode führt, die `fillInStackTrace()` aufruft – das ist `createUriFromHost()`. Daher beginnt die Konsolenausgabe auch mit unserer Methode:

```

Hilfe! Malformed escape pair at index 7: http://%
java.net.URISyntaxException: Malformed escape pair at index 7: http://%
    at RethrowWithFillInStackTrace.createUriFromHost(RethrowWithFillInStackTrace.java:14)
    at RethrowWithFillInStackTrace.main(RethrowWithFillInStackTrace.java:24)

```

6.5.8 Präzises rethrow *

Die Notwendigkeit, Ausnahmen über einen Basistyp zu fangen, ist mit dem Einzug vom *multi-catch* gesunken. Doch für gewisse Programmteile ist es immer noch praktisch, alle Fehler eines gewissen Typs aufzufangen. Wir können auch so weit in der Ausnahmehierarchie nach oben laufen, um alle Fehler aufzufangen – dann haben wir es mit einem `try { ... } catch(Throwable t){ ... }` zu tun. Ein *multi-catch* ist für geprüfte Ausnahmen besonders gut, aber bei ungeprüften Ausnahmen ist eben nicht immer klar, was als Fehler denn so ausgelöst wird, und ein `catch(Throwable t)` hat den Vorteil, dass es alles wegliest.

Problemstellung

Werden Ausnahmen über einen Basistyp gefangen und wird diese Ausnahme mit `throw` weitergeleitet, dann ist es naheliegend, dass der aufgefangene Typ genau der Typ ist, der auch bei `throws` in der Methodensignatur stehen muss.

Stellen wir uns vor, ein Programmblöck nimmt einen Screenshot und speichert ihn in einer Datei. Kommt es beim Abspeichern zu einem Fehler, soll das, was vielleicht schon in die Datei geschrieben wurde, gelöscht werden; die Regel ist also: Entweder steht der Screenshot komplett in der Datei oder es gibt gar keine Datei. Die Methode kann so aussehen, wobei die Ausnahmen an den Aufrufer weitergegeben werden sollen:

```
public static void saveScreenshot( String filename )
    throws AWTException, IOException
{
    try
    {
        Rectangle r = new Rectangle( Toolkit.getDefaultToolkit().getScreenSize() );
        BufferedImage screenshot = new Robot().createScreenCapture( r );
        ImageIO.write( screenshot, "png", new File( filename ) );
    }
    catch ( AWTException e )
    {
        throw e;
    }
    catch ( IOException e )
    {
        new File( filename ).delete();
        throw e;
    }
}
```

Mit den beiden `catch`-Blöcken sind wir genau auf die Ausnahmen eingegangen, die `createScreenCapture()` und `write()` auslösen. Das ist richtig, aber löschen wir wirklich immer die Dateireste, wenn es Probleme beim Schreiben gibt? Richtig ist, dass wir immer dann die Datei löschen, wenn es zu einer `IOException` kommt. Aber was passiert, wenn die Implementierung eine `RuntimeException` auslöst? Dann wird die Datei nicht gelöscht, aber das ist gefragt! Das scheint einfach gefixt, denn statt

```
catch ( IOException e )
{
    new File( filename ).delete();
    throw e;
}
```

schreiben wir:

```
catch ( Throwable e )
{
    new File( filename ).delete();
    throw e;
}
```

Doch können wir das Problem so lösen? Der Typ `Throwable` passt doch gar nicht mehr mit dem deklarierten Typ `IOException` in der Methodensignatur zusammen:

```
public static void saveScreenshot( String filename )
    throws AWTException /*1*/, IOException /*2*/
{
    ...
    catch ( AWTException /*1*/ e )
    {
        throw e;
    }
    catch ( Throwable /*?*/ e )
    {
        new File( filename ).delete();
        throw e;
    }
}
```

Die erste `catch`-Klausel fängt `AWTException` und leitet es weiter. Damit wird `saveScreenshot()` zum möglichen Auslöser von `AWTException` und die Ausnahme muss mit `throws` an die Signatur. Wenn nun ein `catch`-Block jedes `Throwable` auffängt und diesen `Throwable`-Fehler weiterleitet, ist zu erwarten, dass an der Signatur auch `Throwable` stehen muss und `IOException` nicht reicht. Das war auch bis Java 6 so, aber in Java 7 kam eine Anpassung.

Neu seit Java 7: eine präzisere Typprüfung

In Java 7 hat der Compiler eine kleine Veränderung erfahren, von einer unpräziseren zu einer präziseren Typanalyse: Immer dann, wenn in einem catch-Block ein throw stattfindet, ermittelt der Compiler die im try-Block tatsächlich aufgetretenen geprüften Exception-Typen und schenkt dem im catch genannten Typ für das rethrow im Prinzip keine Beachtung. Statt dem gefangenen Typ wird der Compiler den durch die Codeanalyse gefundenen Typ beim rethrow melden.

Der Compiler erlaubt nur dann das präzise rethrow, wenn die catch-Variable nicht verändert wird. Zwar ist eine Veränderung einer nicht-finalen catch-Variablen wie auch unter Java 1.0 erlaubt, doch wenn die Variable belegt wird, schaltet der Compiler von der präzisen in die unpräzise Erkennung zurück. Führen wir etwa die folgende Zuweisung ein, so funktioniert das Ganze schon nicht mehr:

```
catch ( Throwable e )
{
    new File( filename ).delete();
    e = new IllegalStateException();
    throw e;
}
```

Die Zuweisung führt zu dem Compilerhinweis, dass jetzt auch Throwable mit in die throws-Klausel muss.

Stilfrage

Die catch-Variable kann für die präzisere Typprüfung den Modifizierer final tragen, muss das aber nicht tun. Immer dann, wenn es keine Veränderung an der Variablen gibt, wird der Compiler sie als final betrachten und eine präzisere Typprüfung durchführen – daher nennt sich das auch *effektiv final*. Die Java Language Specification rät vom final-Modifizierer aus Stilgründen ab. Ab Java 7 ist es das Standardverhalten, und daher ist es Quatsch, überall ein final dazuzuschreiben, um die präzisere Typprüfung zu dokumentieren.

Migrationsdetail

Da der Compiler nun mehr Typwissen hat, stellt sich die Frage, ob alter Programmcode mit dem neuen präziseren Verhalten vielleicht ungültig werden könnte. Theoretisch ist das möglich, aber die Sprachdesigner haben in über 9 Millionen Zeilen Code¹⁰ von

¹⁰ Die Zahl stammt aus der FOSDEM 2011-Präsentation »Project Coin: Language Evolution in the Open«.



Migrationsdetail (Forts.)

unterschiedlichen Projekten keine Probleme gefunden. Prinzipiell könnte der Compiler jetzt unerreichbaren Code finden, der vorher versteckt blieb. Ein kleines Beispiel, was vor Java 7 kompiliert, aber ab Java 7 nicht mehr:

```
try
{
    throw new FileNotFoundException();
}
catch ( IOException e )
{
    try
    {
        throw e; // e ist für den Compiler vom Typ FileNotFoundException
    }
    catch ( MalformedURLException f ) { }
}
```

Die Variable `e` in `catch (IOException e)` ist effektiv final, und der Compiler führt die präzisere Typerkennung durch. Er findet heraus, dass der wahre rethrow-Typ nicht `IOException`, sondern `FileNotFoundException` ist. Wenn dieser Typ dann mit `throw e` weitergeleitet wird, kann ihn `catch(MalformedURLException)` nicht auffangen. Unter Java 6 war das etwas anders, denn hier wusste der Compiler nur, dass `e` irgendeine `IOException` ist, und es hätte ja durchaus die `IOException`-Unterklasse `MalformedURLException` sein können. (Warum `MalformedURLException` aber eine Unterklasse von `IOException` ist, steht auf einem ganz anderen Blatt.)

6.5.9 Geschachtelte Ausnahmen *

Der Grund für eine Ausnahme mag der sein, dass ein eingebetteter Teil versagt. Das ist vergleichbar mit einer Transaktion: Ist ein Teil der Kette fehlerhaft, so ist der ganze Teil nicht ausführbar. Bei Ausnahmen ist das nicht anders. Nehmen wir an, wir haben eine Methode `foo()`, die im Falle eines Misslingens eine Ausnahme `HellException` auslöst. Ruft unsere Methode `foo()` nun ein Unterprogramm `bar()` auf, das zum Beispiel eine Ein-/ Ausgabeoperation tätigt, und das geht schief, wird die `IOException` der Anlass für unsere `HellException` sein. Es liegt also nahe, bei der Nennung des Grunds für das eigene

Versagen das Misslingen der Unteraufgabe zu nennen (wieder ein Beweis dafür, wie »menschlich« Programmieren sein kann).

Eine *geschachtelte Ausnahme* (engl. *nested exception*) speichert einen Verweis auf eine weitere Ausnahme. Wenn ein Exception-Objekt aufgebaut wird, lässt sich der Grund (engl. *cause*) als Argument im Konstruktor der Throwable-Klasse übergeben. Die Ausnahme-Basisklasse bietet dafür zwei Konstruktoren:

- `Throwable(Throwable cause)`
- `Throwable(String message, Throwable cause)`

Der Grund kann über die Methode `Throwable getCause()` erfragt werden.

Da Konstruktoren in Java nicht vererbt werden, bieten die Unterklassen oft Konstruktoren an, um den Grund anzunehmen: Zumindest `Exception` macht das und kommt somit auf vier Erzeuger:

- `Exception()`
- `Exception(String message)`
- `Exception(String message, Throwable cause)`
- `Exception(Throwable cause)`

Einige der tiefer liegenden Unterklassen haben dann auch diese Konstruktor-Typen mit `Throwable`-Parameter, wie `IOException`, `SQLException` oder `ClassNotFoundException`, andere wiederum nicht, wie `PrinterException`. Eigene Unterklassen können auch mit `initCause()` genau einmal eine geschachtelte Ausnahme angeben.

Geprüfte Ausnahmen in ungeprüfte Ausnahmen verpacken

In modernen Frameworks ist die Nutzung von Ausnahmen, die nicht geprüft werden müssen, also Exemplare von `RuntimeException` sind, häufiger geworden. Bekannte zu prüfende Ausnahmen werden in `RuntimeException`-Objekte verpackt (eine Art `Exception-Wrapper`), die den Verweis auf die auslösende Nicht-`RuntimeException` speichern.

Dazu ein Beispiel. Die folgenden drei Zeilen ermitteln, ob die Web-Seite zu einer URL verfügbar ist:

```
HttpURLConnection.setFollowRedirects( false );
HttpURLConnection con = (HttpURLConnection)(new URL( url ).openConnection());
boolean available = con.getResponseCode() == HttpURLConnection.HTTP_OK;
```

Da der Konstruktor von `URL` eine `MalformedURLException` auslösen kann und es beim Netzwerkzugriff zu einer `IOException` kommen kann, müssen diese beiden Ausnahmen entweder behandelt oder an den Aufrufer weitergereicht werden. Wir wollen eine Variante wählen, in der wir die geprüften Ausnahmen in eine `RuntimeException` hüllen, sodass es eine Utility-Methode gibt und sich der Aufrufer nicht lange mit irgendwelchen Ausnahmen beschäftigen muss:

Listing 6.22: NestedException.java, NestedException

```
public static boolean isAvailable( String url )
{
    try
    {
        HttpURLConnection.setFollowRedirects( false );
        HttpURLConnection con = (HttpURLConnection)(new URL( url ).openConnection());
        return con.getResponseCode() == HttpURLConnection.HTTP_OK;
    }
    catch ( MalformedURLException e )
    {
        throw new RuntimeException( e );
    }
    catch ( IOException e )
    {
        throw new RuntimeException( e );
    }
}

public static void main( String[] args )
{
    System.out.println( isAvailable( "http://laber.rabar.ber/" ) ); // false
    System.out.println( isAvailable( "http://www.tutego.de/" ) ); // true
    System.out.println( isAvailable( "taube://sonsbeck/schlossstrasse/5/" ) ); // ☺
}
```

In der letzten Zeile kommt es zu einer Ausnahme, da es das Protokoll »taube« nicht gibt. Die Ausgabe ist folgende:

```
false
true
```

```
Exception in thread "main" java.lang.RuntimeException:  
  java.net.MalformedURLException: unknown protocol: taube  
    at NestedException.isAvailable(NestedException.java:25)  
    at NestedException.main(NestedException.java:12)  
Caused by: java.net.MalformedURLException: unknown protocol: taube  
    at java.net.URL.<init>(URL.java:590)  
    at java.net.URL.<init>(URL.java:480)  
    at java.net.URL.<init>(URL.java:429)  
    at NestedException.isAvailable(NestedException.java:20)  
    ... 1 more
```

In der Praxis wird es bei großen Stack-Traces – und einem Szenario, bei dem abgefangen, verpackt wird – fast unmöglich, aus der Ausgabe den Verlauf zu entschlüsseln, da sich diverse Teile wiederholen und dann wieder abgekürzt werden. Die duplizierten Teile sind zur Verdeutlichung fett hervorgehoben.

6.6 Automatisches Ressourcen-Management (try mit Ressourcen)

Java hat eine automatische Garbage-Collection (GC), sodass nicht mehr referenzierte Objekte erkannt und ihr Speicher automatisch freigegeben wird. Nun bezieht sich die GC aber ausschließlich auf Speicher, doch es gibt viele weitere Ressourcen:

- Dateisystem-Ressourcen von Dateien
- Netzwerkressourcen wie Socket-Verbindungen
- Datenbankverbindungen
- nativ gebundene Ressourcen vom Grafiksubsystem
- Synchronisationsobjekte

Auch hier gilt es, nach getaner Arbeit aufzuräumen und Ressourcen freizugeben, etwa Dateien und Datenbankverbindungen zu schließen.

Mit dem try-catch-finally-Konstrukt haben wir gesehen, wie Ressourcen freizugeben sind. Doch es lässt sich auch ablesen, dass relativ viel Quellcode geschrieben werden muss und der try-catch-finally drei Unfeinheiten hat:

1. Sollte eine Variable in finally zugänglich sein, muss sie außerhalb des try-Blocks deklariert werden, was ihr eine höhere Sichtbarkeit als nötig gibt.

2. Das Schließen der Ressourcen bringt oft ein zusätzliches try-catch mit sich.
3. Eine im finally ausgelöste Ausnahme (etwa beim close()) überdeckt die im try-Block ausgelöste Ausnahme.

6.6.1 try mit Ressourcen

Um das Schließen von Ressourcen zu vereinfachen, wurde in Java 7 eine besondere Form der try-Anweisung eingeführt, die *try mit Ressourcen* genannt werden. Mit ihm lassen sich *Ressource-Typen*, die die Schnittstelle `java.lang.AutoCloseable` implementieren, automatisch schließen. Ein-/Ausgabeklassen wie `Scanner`, `InputStream` und `Writer`, implementieren diese Schnittstelle und können direkt verwendet werden. Weil try mit Ressourcen dem *Automatic Resource Management* dient, heißt der spezielle try-Block auch *ARM-Block*.

Gehen wir in die Praxis: Aus einer Datei soll mit einem Scanner die erste Zeile gelesen und ausgegeben werden. Nach dem Lesen soll der Scanner geschlossen werden. Die linke Seite der folgenden Tabelle nutzt die spezielle Syntax, die rechte Seite ist im Prinzip die Übersetzung, allerdings noch etwas vereinfacht, wie wir später genauer sehen werden.

try mit Ressourcen	Vereinfachte ausgeschriebene Implementierung
<pre>InputStream in = ClassLoader.getSystemResourceAsStream("EastOfJava.txt"); try (Scanner res = new Scanner(in)) { System.out.println(res.nextLine()); }</pre>	<pre>InputStream in = ClassLoader.getSystemResourceAsStream("EastOfJava.txt"); { final Scanner res = new Scanner(file); try { System.out.println(res.nextLine()); } finally { res.close(); } }</pre>

Tabelle 6.3: Die main()-Methode von TryWithResources1 und ihre prinzipielle Umsetzung

Üblicherweise folgt nach dem Schlüsselwort `try` ein Block, doch `try-mit-Ressourcen` nutzt eine eigene spezielle Syntax.

1. Nach dem `try` folgt statt dem direkten `{}`-Block eine Anweisung in runden Klammern, und dann erst der `{}`-Block, also `try (...) {...}` statt `try {...}`.
2. In den runden Klammern findet man eine lokale Variablen-deklaration (die Variable ist automatisch `final`) mit einer Zuweisung. Die Ressourcenvariable muss vom Typ `AutoCloseable` sein. Rechts vom Gleichheitszeichen steht ein Ausdruck, etwa ein Konstruktor- oder Methodenaufruf. Der Typ des Ausdrucks muss natürlich auch `instanceof AutoCloseable` sein.

Die in dem `try` deklarierte lokale `AutoCloseable`-Variable ist nur in dem Block gültig und wird automatisch freigegeben, gleichgültig ob der ARM-Block korrekt durchlaufen wurde oder ob es bei der Abarbeitung zu einem Fehler kam. Der Compiler fügt alle nötigen Prüfungen ein.

6.6.2 Ausnahmen vom `close()` bleiben bestehen

Unser Scanner-Beispiel hat eine Besonderheit, denn keine der Methoden löst eine geprüfte Ausnahme aus – weder `getSystemResourceAsStream()`, `new Scanner(InputStream)`, `nextLine()` noch das `close()`, was `try-mit-Ressourcen` automatisch aufruft. Anders ist es, wenn die Ressource ein `InputStream` ist, denn dort deklariert die `close()`-Methode eine `IOException`. Die muss daher auch behandelt werden, wie es das folgende Beispiel zeigt:

Listing 6.23: TryWithResourcesReadsLine, `readFirstLine()`

```
static String readFirstLine( File file )
{
    try ( BufferedReader br = new BufferedReader(new FileReader(file)) )
    {
        return br.readLine();
    }
    catch ( IOException e ) { e.printStackTrace(); return null; }
}
```

Wenn `try-mit-Ressourcen` verwendet wird, bleibt die Ausnahme bestehen; es zaubert die Ausnahmen beim `catch` also nicht weg.



Hinweis

Löst `close()` eine geprüfte Ausnahme aus, und wird diese nicht behandelt, so kommt es zum Compilerfehler. Die `close()`-Methode vom `BufferedReader` löst zum Beispiel eine `IOException` aus, sodass sich die folgende Methode nicht übersetzen lässt:

```
void no()
{
    try ( Reader r = new BufferedReader(null) ) { }      // ☹ Compilerfehler
}
```

Der Ausdruck `new BufferedReader(null)` benötigt keine Behandlung, denn der Konstruktor löst keine Ausnahme aus. Einzig der nicht behandelte Fehler von `close()` führt zu »exception thrown from implicit call to close() on resource variable 'r'«.

6.6.3 Die Schnittstelle AutoCloseable

Die ARM-Anweisung schließt Ressourcen vom Typ `AutoCloseable`. Daher wird es Zeit, sich diese Schnittstelle etwas genauer anzuschauen:

```
package java.lang;
public interface AutoCloseable
{
    void close() throws Exception;
}
```

Anders als das übliche `close()` ist die Ausnahme deutlich allgemeiner mit `Exception` angegeben; die Ein-/Ausgabe-Klassen lösen beim Misslingen immer eine `IOException` aus, aber jede Klasse hat eigene Ausnahmetypen:

Typ	Signatur
<code>java.io.Scanner</code>	<code>close()</code> // ohne Ausnahme
<code>javax.sound.sampled.Line</code>	<code>close()</code> // ohne Ausnahme
<code>java.io.InputStream</code>	<code>close()</code> throws <code>IOException</code>
<code>java.sql.Connection</code>	<code>close()</code> throws <code>SQLException</code>

Tabelle 6.4: Einige Typen, die `AutoCloseable` implementieren

Eine Unterklasse darf die Ausnahme ja auch weglassen, das machen Klassen wie der Scanner, der keine Ausnahme weiterleitet, sondern sie intern schluckt – wenn es Ausnahmen gab, liefert sie die Scanner-Methode `ioException()`.

AutoCloseable und Closeable

Auf den ersten Blick einleuchtend wäre es, die schon existierende Schnittstelle `Closeable` als Typ zu nutzen. Doch das hätte Nachteile: Die `close()`-Methode ist mit einem `throws IOException` deklariert, was bei einer allgemeinen automatischen Ressourcen-Freigabe unpassend ist, wenn etwa ein Grafikobjekt bei der Freigabe eine `IOException` auslöst. Vielmehr ist der Weg anders herum: `Closeable` erweitert `AutoCloseable`, denn das Schließen von Ein-/Ausgabe-Ressourcen ist eine besondere Art, allgemeine Ressourcen zu schließen.

```
package java.io;

import java.io.IOException;

public interface Closeable extends AutoCloseable
{
    void close() throws IOException;
}
```

Wer ist AutoCloseable?

Da alle Klassen, die `Closeable` implementieren, auch automatisch vom Typ `AutoCloseable` sind, kommen schon einige Typen zusammen. Im Wesentlichen sind es aber Klassen aus dem `java.io`-Paket, wie `Channel`-, `Reader`-, `Writer`-Implementierungen, `FileLock`, `XMLDecoder` und noch ein paar Exoten wie `URLClassLoader`, `ImageOutputStream`. Auch Typen aus dem `java.sql`-Paket gehören zu den Nutznießern. Klassen aus dem Bereich `Threading`, wo etwa ein Lock wieder freigeben werden könnte, oder `Grafik`-Anwendungen, bei denen der `Grafik`-Kontext wieder freigegeben werden muss, gehören nicht dazu.

6.6.4 Mehrere Ressourcen nutzen

Unsere beiden Beispiele zeigten die Nutzung eines Ressource-Typs. Es sind aber auch mehrere Typen möglich, die dann mit einem Semikolon getrennt werden:

```
try ( InputStream in = new FileInputStream(src);
      OutputStream out = new FileOutputStream(dest) )
{ ... }
```



Hinweis

Die Trennung erledigt ein Semikolon, und jedes Segment kann einen unterschiedlichen Typ deklarieren, etwa `InputStream/OutputStream`. Die Ressourcen-Typen müssen also nicht gleich sein, und auch wenn sie es sind, muss der Typ immer neu geschrieben werden, also etwa:

```
try ( InputStream in1 = ...; InputStream in2 = ... )
```

Es ist ungültig, Folgendes zu schreiben:

```
try ( InputStream in1 = ..., in2 = ... ) // ☹ Compilerfehler
```

Wenn es beim Anlegen in der Kette zu einem Fehler kommt, wird nur das geschlossen, was auch aufgemacht wurde. Wenn es also bei der ersten Initialisierung von `in1` schon zu einer Ausnahme kommt, wird die Belegung von `in2` erst gar nicht begonnen und daher auch nicht geschlossen. (Intern setzt der Compiler das als geschachtelte `try-catch-finally`-Blöcke um.)



Beispiel

Am Schluss der Ressourcensammlung kann – muss aber nicht – ein Semikolon stehen, so wie auch bei Feldinitialisierungen zum Schluss ein Komma stehen kann:

```
int[] array = { 1, 2, };
// ^ Komma optional
try ( InputStream in = new FileInputStream(src); ) { ... }
// ^ Semikolon optional
try ( InputStream in = new FileInputStream(src);
      OutputStream out = new FileOutputStream(dest); ) { ... }
// ^ Semikolon optional
```

Ob das stilvoll ist, muss jeder selbst entscheiden; in der Insel steht kein unnützes Zeichen.

6.6.5 Unterdrückte Ausnahmen *

Aufmerksame Leser haben bestimmt schon ein Detail wahrgenommen: Im Text steht »vereinfachte ausgeschriebene Implementierung«, was vermuten lässt, dass es ganz so einfach doch nicht ist. Das stimmt, denn es können zwei Ausnahmen auftauchen, die einiges an Sonderbehandlung benötigen:

- Ausnahme im try-Block. An sich unproblematisch.
- Ausnahme beim `close()`. Auch an sich unproblematisch. Aber es gibt mehrere `close()`-Aufrufe, wenn nicht nur eine Ressource verwendet wurde. Ungünstig.
- Die Steigerung: Ausnahme im try-Block und dann auch noch Ausnahme(n) beim `close()`. Das ist ein echtes Problem!

Eine Ausnahme alleine ist kein Problem, aber zwei Ausnahmen auf einmal bilden ein großes Problem, da ein Programmblöck nur genau eine Ausnahme melden kann und nicht eine Sequenz von Ausnahmen. Daher sind verschiedene Fragen zu klären, falls der try-Block und `close()` beide eine Ausnahme auslösen:

- Welche Ausnahme ist wichtiger? Die Ausnahme im try-Block oder die vom `close()`?
- Wenn es zu zwei Ausnahmen kommt: Soll die von `close()` vielleicht immer verdeckt werden und immer nur die vom try-Block zum Anwender kommen?
- Wenn beide Ausnahmen wichtig sind, wie sollen sie gemeldet werden?

Wie haben sich die Java-Ingenieure entschieden? Eine Ausnahme bei `close()` darf bei einem gleichzeitigen Auftreten einer Exception im try-Block auf keinen Fall verschwinden.¹¹ Wie also beide Ausnahmen melden? Hier gibt es einen Trick: Da die Ausnahme im try-Block wichtiger ist, ist sie die »Haupt-Ausnahme«, und die `close()`-Ausnahme kommt Huckepack als Extra-Information mit oben drauf.

Dieses Verhalten soll das nächste Beispiel zeigen. Um die Ausnahmen besser steuern zu können, soll eine eigene AutoCloseable-Implementierung eine Ausnahme in `close()` auslösen.

Listing 6.24: SuppressedClosed.java

```
public class SuppressedClosed
{
    public static void main( String[] args )
    {
```

¹¹ In einem frühen Prototyp war dies tatsächlich der Fall – die Ausnahme wurde komplett geschluckt.

```

class NotCloseable implements AutoCloseable
{
    @Override public void close()
    {
        throw new UnsupportedOperationException( "close() mag ich nicht" );
    }
}

try ( NotCloseable res = new NotCloseable() ) {
    throw new NullPointerException();
}
}
}

```

Das Programm löst also im `close()` und im `try`-Block eine Ausnahme aus. Das Resultat ist:

```

Exception in thread "main" java.lang.NullPointerException
    at SuppressedClosed.main(SuppressedClosed.java:14)
Suppressed: java.lang.UnsupportedOperationException: close() mag ich nicht
    at SuppressedClosed$1NotCloseable.close(SuppressedClosed.java:9)
    at SuppressedClosed.main(SuppressedClosed.java:15)

```

Die interessante Zeile beginnt mit »`Suppressed:`«, denn dort ist die `close()`-Ausnahme referenziert. An den Aufrufer kommt die spannende Ausnahme vom misslungenen `try`-Block aber nicht direkt von `close()`, sondern verpackt in der Hauptausnahme und muss extra erfragt werden.

Zum Vergleich: Kommentieren wir `throw new NullPointerException()` aus, gibt es nur noch die `close()`-Ausnahme und es folgt auf der Konsole:

```

Exception in thread "main" java.lang.UnsupportedOperationException: close() mag ich nicht
    at SuppressedClosed$1NotCloseable.close(SuppressedClosed.java:9)
    at SuppressedClosed.main(SuppressedClosed.java:15)

```

Die Ausnahme ist also nicht irgendwo anders untergebracht, sondern die »Hauptausnahme«.

Eine Steigerung ist, dass es mehr als eine Ausnahme beim Schließen geben kann. Simulieren wir auch dies wieder an einem Beispiel, indem wir unser Beispiel um eine Zeile ergänzen:

```
try ( NotCloseable res1 = new NotCloseable();
      NotCloseable res2 = new NotCloseable() )
{
    throw new NullPointerException();
}
```

Aufgerufen führt dies zu:

```
Exception in thread "main" java.lang.NullPointerException
  at SuppressedClosed.main(SuppressedClosed.java:15)
Suppressed: java.lang.UnsupportedOperationException: close() mag ich nicht
  at SuppressedClosed$1NotCloseable.close(SuppressedClosed.java:9)
  at SuppressedClosed.main(SuppressedClosed.java:16)
Suppressed: java.lang.UnsupportedOperationException: close() mag ich nicht
  at SuppressedClosed$1NotCloseable.close(SuppressedClosed.java:9)
  at SuppressedClosed.main(SuppressedClosed.java:16)
```

Jede unterdrückte `close()`-Ausnahme taucht auf.

Umsetzung



Im Kapitel über `finally` wurde das Verhalten vorgestellt, dass eine Ausnahme im `finally` eine Ausnahme im `try`-Block unterdrückt. Der Compiler setzt bei der Umsetzung vom `try-mit-Ressourcen` das `close()` in einen `finally`-Block. Ausnahmen im `finally`-Block sollen eine mögliche Hauptausnahme aber nicht schlucken. Daher fängt die Umsetzung vom Compiler jede mögliche Ausnahme im `try`-Block ab sowie die `close()`-Ausnahme und hängt diese Schließ-Ausnahme, falls vorhanden, an die Hauptausnahme.

Spezielle Methoden in `Throwable` *

Damit eine normale Exception die unterdrückten `close()`-Ausnahmen Huckepack nehmen kann, sind in der Basisklasse `Throwable` seit Java 7 zwei Methoden hinzugekommen:

```
final class java.lang.Throwable
```

- final Throwable[] getSuppressed()

Liefert alle unterdrückten Ausnahmen. Die `printStackTrace()`-Methode zeigt alle unterdrückten Ausnahmen und greift auf `getSuppressed()` zurück. Für Anwender wird es selten Anwendungsfälle für diese Methode geben.

- final void addSuppressed(Throwable exception)

Fügt eine neue unterdrückte Ausnahme hinzu. In der Regel ruft der `finally`-Block vom `try-mit-Ressourcen` die Methode auf, doch wir können auch selbst die Methode nutzen, wenn wir mehr als eine Ausnahme melden wollen. Die Java-Bibliothek selbst nutzt das bisher nur an sehr wenigen Stellen.

Neben den beiden Methoden gibt es einen `protected`-Konstruktor, der bestimmt, ob es überhaupt unterdrückte Ausnahmen geben soll oder ob sie nicht vielleicht komplett verschluckt werden. Wenn, dann zeigt sie auch `printStackTrace()` nicht mehr an.



Blick über den Tellerrand

In C++ gibt es Dekonstruktoren, die beliebige Anweisungen ausführen, wenn ein Objekt freigegeben wird. Hier lässt sich auch das Schließen von Ressourcen realisieren. C# nutzt statt `try` das spezielle Schlüsselwort `using`, mit Typen, die die Schnittstelle `IDisposable` implementieren, mit einer Methode `Dispose()` statt `close()`. (In Java sollte die Schnittstelle ursprünglich auch `Disposable` statt nun `AutoCloseable` heißen.) In Python 2.5 wurde ein *context management protocol* mit dem Schlüsselwort `with` realisiert, sodass Python automatisch bei Betreten eines Blockes `__enter__()` aufruft und beim Verlassen die Methode `__exit__()`. Das ist insofern interessant, als dass hier zwei Methoden zur Verfügung stehen. Bei Java ist es nur `close()` beim Verlassen des Blockes, aber es gibt keine Methode zum Betreten eines Blockes; so etwas muss beim Anlegen der Ressource erledigt werden.

6.7 Besonderheiten bei der Ausnahmebehandlung *

Bei der Ausnahmebehandlung gibt es ein paar Überraschungen, die vier Abschnitte gesondert vorstellen.

6.7.1 Rückgabewerte bei ausgelösten Ausnahmen

Java versucht, durch den Programmfluss den Ablauf innerhalb einer Methode zu bestimmen und zu melden, ob sie definitiv einen Rückgabewert liefert. Dabei verfolgt der Compiler die Programmpfade und wertet bestimmte Ausdrücke aus. Doch die Aussage »Jede Methode mit einem Ergebnistyp ungleich void muss eine return-Anweisung besitzen« müssen wir etwas relativieren. Nur in einem speziellen Fall müssen wir dies nicht: nämlich genau dann, wenn vor dem Ende der Methode eine throw-Anweisung die Abarbeitung beendet:

Listing 6.25: NoReturn.java

```
class Windows7KeyGenerator
{
    public String generateKey()
    {
        throw new UnsupportedOperationException();
    }
}
```

Ein Blick auf generateKey() verrät, dass trotz eines angekündigten Rückgabewerts keine return-Anweisung im Rumpf steht. Die Abarbeitung wird vor dem Rücksprung durch eine Exception abgebrochen. generateKey() muss diese Exception nicht mit throws ankündigen, da UnsupportedOperationException eine RuntimeException ist.

6.7.2 Ausnahmen und Rückgaben verschwinden: Das Duo »return« und »finally«

Ein Phänomen in der Ausnahmebehandlung von Java ist eine return-Anweisung innerhalb eines finally-Blocks. Zunächst einmal »überschreibt« ein return im finally-Block den Rückgabewert eines return im try-Block:

```
static String getIsbn()
{
    try {
        return "3821829877";
    }
    finally {
        return "";
    }
}
```

Der Aufrufer empfängt immer einen leeren String.

Interessant ist auch folgendes Programm:

```
public static int a()
{
    while ( true )
    {
        try {
            return 0;
        }
        finally {
            break;
        }
    }

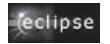
    return 1;
}
```

Die Ausgabe auf der Konsole ist 1. Das `break` im `finally` lässt die Laufzeitumgebung aus der Schleife austreten und den Rückgabewert ignorieren.

Ein weiteres Kuriosum sind Ausnahmen. Die Laufzeitumgebung gibt bei einer `return`-Anweisung im `finally`-Block eine im `try`-Block ausgelöste Ausnahme *nicht* zum Aufrufer weiter, sondern bietet einfach die Rückgabe an.

Die folgende Methode löst zum Beispiel eine `RuntimeException` aus, die aber der Aufrufer der Methode nie sieht:

```
static void obamaVsMcCain()
{
    try {
        throw new RuntimeException();
    }
    finally
    {
        return;
    }
}
```



Entfernen wir die Zeile mit dem `return`, ist das Verhalten der Laufzeitumgebung wie erwartet.

Der Java-Compiler von Eclipse markiert die Diskrepanz und zeigt eine Warnung an (»finally block does not complete normally«). Mit der Annotation `@SuppressWarnings("finally")` schalten wir diesen Hinweis ab.

6

6.7.3 throws bei überschriebenen Methoden

Beim Überschreiben von Methoden gibt es eine wichtige Regel: Überschriebene Methoden in einer UnterkLASSE dürfen *nicht mehr* Ausnahmen auslösen, als schon beim `throws`-Teil der OberKLasse aufgeführt sind. Da das gegen das Substitutionsprinzip verstieße, kann eine Methode der UnterkLASSE nur

- dieselben Ausnahmen wie die OberKLasse auslösen,
- Ausnahmen spezialisieren oder
- weglassen.

Dazu sehen wir hier ein konstruiertes Beispiel für die beiden letzten Fälle:

Listing 6.26: ExceptionlessRandomAccessFile.java

```
import java.io.*;
import java.net.ProtocolException;

public class ExceptionlessRandomAccessFile extends RandomAccessFile
{
    public ExceptionlessRandomAccessFile( File file, String mode )
        throws FileNotFoundException
    {
        super( file, mode );
    }

    @Override
    public long length()
    {
        try
        {
            return super.length();
        }
    }
}
```

```

        }
    catch ( IOException e )
    {
        return 0;
    }
}

@Override
public void write( int b ) throws ProtocolException
{
    try
    {
        super.write( b );
    }
    catch ( IOException e )
    {
        throw new ProtocolException();
    }
}
}

```

Die Methoden `length()` und `write()` lösen in `RandomAccessFile` eine `IOException` aus. Unsere Unterklasse `ExceptionlessRandomAccessFile` überschreibt zunächst `length()` und lässt die Ausnahme in der Signatur weg. Das hat in der Nutzung einige Folgen, denn wenn wir die Klasse als `ExceptionlessRandomAccessFile` der Art

```

ExceptionlessRandomAccessFile raf = ...
raf.length();

```

verwenden, muss die nicht vorhandene zu prüfende Ausnahme von `length()` ebenfalls nicht abgefangen werden – und darf es auch gar nicht, weil ein `try-catch` auf eine `IOException` zu einem Compilerfehler führt.

Umgekehrt: Ist `raf` vom Typ der Basisklasse `RandomAccessFile`, muss die Ausnahme auf jeden Fall abgefangen werden:

```

RandomAccessFile raf = ...;
try
{

```

```

    raf.length();
}
catch ( IOException e ) { }

```

Das zeigt die Schwierigkeit bei überschriebenen Methoden, die Ausnahmen weglassen.

Bei der Methode `write()` führt `throws` den Ausnahmetyp `ProtocolException` als Unterklasse von `IOException` auf. Natürlich reicht es nicht aus, in `write()` einfach `super.write()` stehen zu lassen (was nur eine allgemeinere `IOException` auslösen würde, aber nicht die versprochene speziellere `ProtocolException`). Daher fangen wir im Rumpf der Methode `das super.write() ab und erzeugen die speziellere ProtocolException.`

6

Design



Wenn demnach eine überschriebene Methode der Unterklassie keine geprüften Fehler hinzufügen kann, muss das Design der Basistypen so entworfen sein, dass Unterklassen notwendige Fehler melden können.

6.7.4 Nicht erreichbare catch-Klauseln

Eine `catch`-Klausel heißt *erreichbar*, wenn es in dem `try`- und `catch`-Block eine Anweisung gibt, die die in der `catch`-Klausel abgefangene Fehlerart tatsächlich auslösen kann. Zusätzlich darf vor dieser `catch`-Klausel natürlich kein anderes `catch` stehen, das diesen Fehlerfall mit abfängt. Wenn wir zum Beispiel `catch(Exception e)` als erstes Auffangbecken bereitstellen, werden natürlich alle Ausnahmen dort behandelt. Die Konsequenz daraus: `catch`-Klauseln sollten immer von den speziellen zu den allgemeinen Fehlerarten sortiert werden.

Wenn wir ein Objekt `RandomAccessFile` aufbauen und anschließend `readLine()` verwenden, so muss eine `FileNotFoundException` vom Konstruktor und eine `IOException` von `readLine()` abgefangen werden. Da eine `FileNotFoundException` eine Spezialisierung ist, also eine Unterklasse von `IOException`, würde ein `catch(IOException e)` schon reichen. Steht im Quellcode folglich der `catch` für die `FileNotFoundException` dahinter, wird der Teil nie ausgeführt werden können, und der Compiler merkt das zu Recht an.

Übertriebene throws-Klauseln

Eine Methode compiliert, auch wenn sie zu viele oder zu allgemeine Fehlerarten in ihrer `throws`-Klausel angibt:

Listing 6.27: TooManyExceptions.java, openFile()

```
void openFile() throws FileNotFoundException,
                     IOException,
                     InterruptedException
{
    new RandomAccessFile( "", "" );
}
```

Unsere Methode `openFile()` ruft den Konstruktor von `RandomAccessFile` auf, was bekannterweise zu einer `FileNotFoundException` führen kann. `openFile()` jedoch gibt neben `FileNotFoundException` noch die allgemeinere Oberklasse `IOException` an und meldet mit `InterruptedException` noch eine geprüfte Ausnahme, die der Rumpf überhaupt auslöst. Trotzdem lässt der Compiler das durch.

Beim Aufruf solcher Methoden in `try`-Blöcken müssen in den `catch`-Klauseln die zu viel deklarierten Exceptions aufgefangen werden, auch wenn sie nicht wirklich erreicht werden können:

Listing 6.28: TooManyExceptions.java, useFile()

```
try
{
    openFile();
}
catch ( IOException e ) { }
catch ( InterruptedException e ) { }
```

Der Sinn besteht darin, dass dies später in einer Erweiterung einer Methode, etwa einer `InterruptedException`, durchaus vorkommen kann, und dann sind die Aufrufer darauf schon vorbereitet.

6.8 Den Stack-Trace erfragen *

Bei jedem Methodenaufruf legt die JVM eine Kennung auf einen *Aufrufstapel* (eng. *call stack*). Um den Aufrufstapel zu einem bestimmten Zeitpunkt sichtbar zu machen, lässt sich ein *Stack-Trace* erfragen. Der Stack-Trace ist dabei allerdings nicht der Aufrufstapel selbst – Java setzt einige Abstraktionen über Maschinencode und gibt die internen Vorgänge nicht preis –, sondern eine Liste von `StackTraceElement`-Objekten. Jedes dieser Ob-

ekte bietet Zugriff auf den Namen der Datei, in der die Methode deklariert wurde, auf die Programmzeile, den Methodennamen und die Information darüber, ob die Methode nativ ist oder nicht.

Warum sind Stack-Traces nun nützlich? Einige Antworten:

- Stack-Traces sind unverzichtbare Aussagen im Fehlerfall und beim Debugging, denn Entwickler möchten gerne wissen, welchen Weg Aufrufe genommen haben.
- Sie helfen beim Erkennen von Rekursionen. Bei einer Rekursion ruft eine Methode direkt oder indirekt sich selbst auf. Das allein ist nicht problematisch, aber wenn eine Abbruchbedingung fehlt, so kommt es zu einer Endlosrekursion, was zum Platzen des Stack-Speichers führt und letztendlich zu einem `StackOverflowError`. Eine Untersuchung zur Laufzeit kann rekursive Aufrufe identifizieren und den Thread gleich abbrechen, anstatt dass der wild gewordene Thread etwa in jedem Methodenaufruf noch Speicher alloziert und es dann mit einem `OutOfMemoryError` noch schlimmer kommt.
- Methoden können durch Stack-Inspektionen Sicherheitsprüfungen durchführen, etwa nachschauen, ob ein Aufruf überhaupt autorisiert ist. Die Java Security basiert auf dieser Arbeitsweise, und ein Access Controller arbeitet genau für diesen Test den Stack ab.

6.8.1 StackTraceElement

Java bietet unterschiedliche Möglichkeiten, einen Stack-Trace zu erfragen und somit an eine Sammlung von `StackTraceElement`-Objekten zu kommen, die die Aufrufgeschichte zeigen:

- Bei einer ausgelösten Ausnahme verrät die Methode `getStackTrace()` vom `Throwable`-Objekt den aktuellen Stack-Trace.
- Ein `Thread`-Objekt gibt seine eigene Aufrufhierarchie mit `getStackTrace()` an. Jeder Thread hat einen eigenen Aufrufstapel.
- Die statische Methode `Thread.getAllStackTraces()` liefert für alle Threads die aktuellen Stack-Traces.

Die genannten Methoden liefern Sammlungen von `StackTraceElement`-Objekten mit folgenden Informationen:

```
final class java.lang.StackTraceElement
    implements Serializable
```

- String getClassName()
- String getFileName()
- int getLineNumber()
- String getMethodName()
- boolean isNativeMethod()

Bedauerlicherweise fehlen wichtige Informationen, wie etwa Zeitpunkte.

6.8.2 printStackTrace()

Eine Methode, die in den Buchbeispielen schon oft zum Einsatz kam, war `printStackTrace()`: Sie gibt auf dem Bildschirm den Stack-Trace aus. Die Methode `printStackTrace()` stammt von `Throwable` und ist überladen; die Signaturen sind: `printStackTrace()`, `printStackTrace(PrintStream)` und `printStackTrace(PrintWriter)`. Intern ruft `printStackTrace()` einfach `printStackTrace(System.err)` auf, sodass der Stack-Trace automatisch auf den Fehlerausgabekanal kommt.

```
class java.lang.Throwable
    implements Serializable
```

- void `printStackTrace()`

Schreibt das `Throwable` und anschließend den Stack-Inhalt in den Standardausgabestrom.

- void `printStackTrace(PrintStream s)`

Schreibt das `Throwable` und anschließend den Aufruf-Stack in den angegebenen PrintStream.

- void `printStackTrace(PrintWriter s)`

Schreibt das `Throwable` und anschließend den Aufruf-Stack in den angegebenen PrintWriter.

6.8.3 StackTraceElement vom Thread erfragen

Sehen wir uns ein Beispielprogramm an, das mehrere Methoden aufruft und dann den Stack-Trace ausgibt:

Listing 6.29: GetStackTrace.java

```
public class GetStackTrace
{
    public static void showTrace()
    {
        for ( StackTraceElement trace : Thread.currentThread().getStackTrace() )
            System.out.println( trace );
    }

    public static void m( int n )
    {
        if ( n == 0 )
            showTrace();
        else
            m( n - 1 );
    }

    public static void main( String[] args )
    {
        m( 2 );
    }
}
```

Ein StackTraceElement deklariert eine `toString()`-Methode, und die Ausgabe sieht dann so aus:

```
java.lang.Thread.getStackTrace(Thread.java:1578)
GetStackTrace.showTrace(GetStackTrace.java:5)
GetStackTrace.m(GetStackTrace.java:12)
GetStackTrace.m(GetStackTrace.java:14)
GetStackTrace.m(GetStackTrace.java:14)
GetStackTrace.main(GetStackTrace.java:19)
```

```
class java.lang.Thread
    implements Runnable
```

- static Map<Thread, StackTraceElement[]> getAllStackTraces()

Die statische Methode liefert von allen Threads in einem Assoziativspeicher ein Feld von StackTraceElement-Objekten.

- StackTraceElement[] getStackTrace()

Liefert den Stack-Trace für den Thread.

zB

Beispiel

Finde heraus, in welcher Methode wir gerade stehen:

```
StackTraceElement e = Thread.currentThread().getStackTrace()[2];
System.out.println( e.getMethodName() );
```

Intern könnte sich durchaus die Position (hier im Stack an Stelle zwei) verschieben!

!

Realitätstest

Der Stack-Trace, den Java bietet, ist relativ arm an Informationen, und die Standardausgabe über printStrackTrace() ist unübersichtlich. Zudem fehlen spannende Informationen wie Zeitpunkte, und ein getAllStackTraces() ist bei vielen Threads und tiefen Aufrufhierarchien sehr langsam. Summa summarum: Ein guter Debugger beziehungsweise ein Analysetool mit Zugriff auf die JVM-Eigenschaften ist für die ernsthafte Entwicklung unumgänglich.

6.9 Assertions *

Die Übersetzung des englischen Worts *assertion* lässt vermuten, worum es geht: um *Behauptungen*. Mit diesen werden innerhalb von Methoden *Zusicherungen* (Vor- und Nachbedingungen) aufgestellt, die den korrekten Ablauf der Methode garantieren sollen. Ist eine Bedingung nicht erfüllt, wird ein Fehler ausgelöst, der darauf hinweist, dass im Programm etwas falsch gelaufen sein muss. Die ausgelösten Fehler sind vom Typ »Error« und nicht vom Typ »Exception« und sollten daher auch nicht aufgefangen werden, da eine nicht erfüllte Bedingung ein Programmierfehler ist.

Java-Geschichte



Änderungen in der Sprache Java gab es immer wieder. In Java 1.1 wurden innere Klassen eingeführt, in Java 1.3 das Schlüsselwort `strict`, in Java 1.4 das Schlüsselwort `assert` für die »Behauptungen«, und in Java 5 gab es neben weiteren Änderungen auch Aufzählungen mit dem eingeführten Schlüsselwort `enum`.

6.9.1 Assertions in eigenen Programmen nutzen

Assertions werden im Java-Quellcode mit der `assert`-Anweisung benutzt. Es gibt zwei Varianten, eine mit und eine ohne Meldung:

```
assert AssertConditionExpression;
assert AssertConditionExpression : MessageExpression;
```

`AssertConditionExpression` steht für eine Bedingung, die zur Laufzeit ausgewertet wird. Wertet sich das Ergebnis zu `true` aus, führt die Laufzeitumgebung die Abarbeitung normal weiter; ergibt die Auswertung `false`, wird das Programm mit einem `java.lang.AssertionError` beendet. Der optionale zweite Parameter, `MessageExpression`, ist ein Text, der beim Stack-Trace als Nachricht in der Fehlermeldung erscheint.

Formulieren wir ein Beispiel: Eine eigene statische Methode `subAndSqrt(double, double)` bildet die Differenz zweier Zahlen und zieht aus dem Ergebnis die Wurzel. Natürlich weiß jeder Entwickler, dass die Wurzel aus negativen Zahlen nicht erlaubt ist, aber dennoch ginge so etwas in Java durch, nur das Ergebnis ist eine `NaN`. Sollte irgendein Programmteil nun die Methode `subAndSqrt()` mit einem falschen Paar Zahlen aufrufen und das Ergebnis `NaN` sein, muss ein `Assert-Error` erfolgen, da es einen internen Programmfehler zu korrigieren gilt:

Listing 6.30: com/tutego/insel/assertion/AssertKeyword.java

```
package com.tutego.insel.assertion;

public class AssertKeyword
{
    public static double subAndSqrt( double a, double b )
    {
        double result = Math.sqrt( a - b );
        assert result >= 0 : "Wurzel einer negativen Zahl!";
        return result;
    }
}
```

```
assert ! Double.isNaN( result ) : "Berechnungsergebnis ist NaN!";

    return result;
}

public static void main( String[] args )
{
    System.out.println( "Sqrt(10-2)=" + subAndSqrt(10, 2) );
    System.out.println( "Sqrt(2-10)=" + subAndSqrt(2, 10) );
}
```

6.9.2 Assertions aktivieren

Ein aktueller Java-Compiler übersetzt das Beispiel ohne Fehler, doch zur Laufzeit werden die Assertions standardmäßig nicht beachtet, da sie abgeschaltet sind. Somit entsteht kein Geschwindigkeitsverlust bei der Ausführung der Programme. Um Assertions zu aktivieren, muss die Laufzeitumgebung mit dem Schalter `-ea` (*enable assertions*) gestartet werden.

```
$ java -ea AssertKeyword
```

Die Ausgabe ist dann:

```
Sqrt(10-2)=2.8284271247461903
Exception in thread "main" java.lang.AssertionError: Berechnungsergebnis ist NaN!
at com.tutego.insel.assertion.AssertKeyword.subAndSqrt(AssertKeyword.java:9)
at com.tutego.insel.assertion.AssertKeyword.main(AssertKeyword.java:17)
```

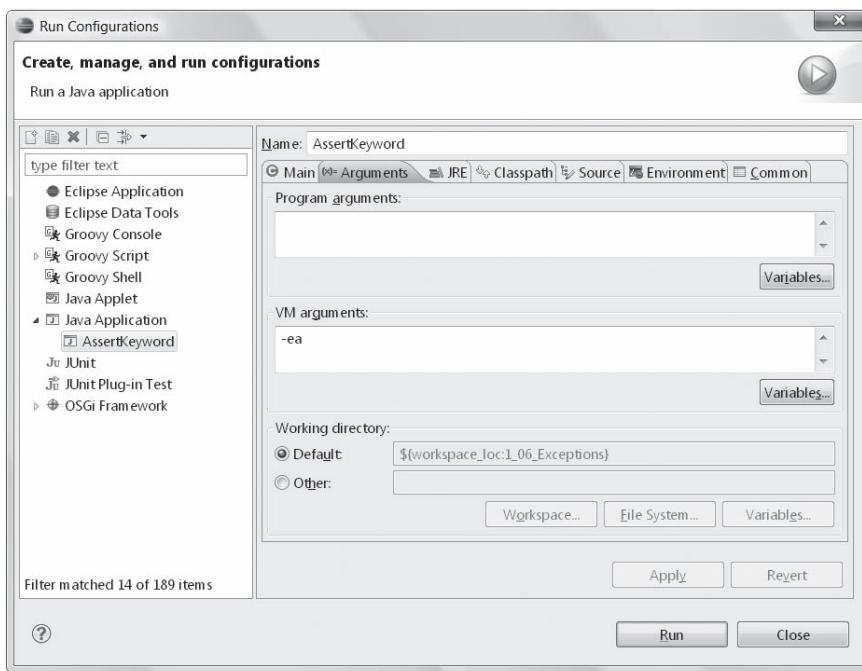


Abbildung 6.13: Wurde das Programm in Eclipse schon gestartet, kann im Menü RUN • RUN CONFIGURATIONS ... auf dem Reiter »Arguments« bei den VM arguments der Schalter »-ea« gesetzt werden.

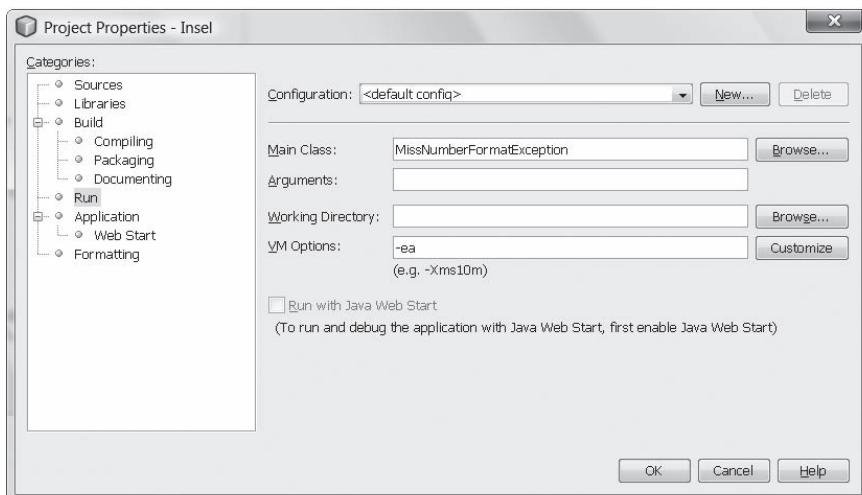


Abbildung 6.14: Unter FILE • PROJECT PROPERTIES kann der VM-Schalter für die Assertions gesetzt werden.

6.10 Zum Weiterlesen

Unterschiedliche Programm-Architekten setzen Ausnahmen unterschiedlich ein, und so sind zwei Lager anzutreffen: diejenigen, die eher mit geprüften Ausnahmen arbeiten, und die, die eher mit ungeprüften Ausnahmen modellieren. Daher muss dieser Aspekt bei jeder neuen Bibliothek und mit jedem neuen Framework mit gelernt werden, so wie die Artikel zu den Substantiven im Deutschen. Eine vernünftige Strategie ist jedoch unabdingbar, zumindest sollten Ausnahmen geloggt werden – das Kapitel 20 aus »Java 7 – Mehr als eine Insel« stellt das vor. Beim Testen ist zudem darauf zu achten, dass der Methode nicht nur nette Eingabewerte gereicht werden, sondern auch falsche Argumente, sodass die bei Falscheingaben zu erwarteten Ausnahmen mit getestet werden müssen.



Kapitel 7

Äußere.innere Klassen

»Der Nutzen ist ein Teil der Schönheit.«

– Albrecht Dürer (1471–1528)

7.1 Geschachtelte (innere) Klassen, Schnittstellen, Aufzählungen

Bisher haben wir Klassen, Schnittstellen und Aufzählungen kennengelernt, die entweder allein in der Datei oder zusammen mit anderen Typen in einer Datei, also einer Compilationseinheit, deklariert wurden. Es gibt darüber hinaus die Möglichkeit, eine Klasse, Aufzählung oder Schnittstelle in andere Typdeklarationen hineinzunehmen. Das ist sinnvoll, denn die Motivation dahinter ist, noch mehr Details zu verstecken, denn es gibt sehr lokale Typdeklarationen, die keine größere Sichtbarkeit brauchen.

Für eine Klasse `In`, die in eine Klasse `Out` gesetzt wird, sieht das im Quellcode so aus:

```
class Out {  
    class In {  
    }  
}
```

Eine geschachtelte Klasse, die so eingebunden wird, heißt *innere Klasse*. Im Folgenden wollen wir nicht mehr ständig betonen, dass auch Schnittstellen als Typen eingebettet werden können, und bleiben bei der einfachen Sprachregelung *innere Klassen*. (Aufzählungen werden vom Compiler in Klassen übersetzt und müssen daher nicht unbedingt gesondert behandelt werden. Natürlich lassen sich auch Aufzählungen innerhalb von Klassen oder Schnittstellen deklarieren.)

Die Java-Spezifikation beschreibt vier Typen von inneren Klassen, die im Folgenden vorgestellt werden. Egal, wie sie deklariert werden, es ist eine enge Kopplung der Typen, und der Name des inneren Typs muss sich vom Namen des äußeren Typs unterscheiden.

Typ	Beispiel
statische innere Klasse	<pre>class Out { static class In {}</pre>
Mitgliedsklasse	<pre>class Out { class In {}</pre>
lokale Klasse	<pre>class Out { Out() { class In {}</pre>
anonyme innere Klasse	<pre>class Out { Out() { new Runnable() { public void run() {}</pre>

Tabelle 7.1: Die vier Typen von inneren Klassen



Hinweis

Das Gegenteil von geschachtelten Klassen, also das, womit wir uns bisher die ganze Zeit beschäftigt haben, heißt *Top-Level-Klasse*. Die Laufzeitumgebung kennt nur Top-Level-Klassen, und geschachtelte innere Klassen werden letztendlich zu ganz »normalen« Klassendeklarationen.

7.2 Statische innere Klassen und Schnittstellen

Die einfachste Variante einer inneren Klasse oder Schnittstelle wird wie eine statische Eigenschaft in die Klasse eingesetzt und heißt *statische innere Klasse*. Wegen der Schachtelung wird dieser Typ im Englischen *nested top-level class* genannt. Die Namensgebung betont mit dem Begriff *top-level*, dass die Klassen das Gleiche können wie

»normale« Klassen oder Schnittstellen, nur bilden sie quasi ein kleines Unterpaket mit eigenem Namensraum. Insbesondere sind zur Erzeugung von Exemplaren von statischen inneren Klassen nach diesem Muster keine Objekte der äußeren Klasse nötig. (Die weiteren inneren Typen, die wir kennenlernen wollen, sind alle nicht-statisch und benötigen einen Verweis auf das äußere Objekt.) Oracle betont in der Spezifikation der Sprache, dass die statischen inneren Klassen keine »echten« inneren Klassen sind, doch um die Sprache einfach zu halten, bleiben wir bei »statischen inneren Typen«.

Deklarieren wir `Lamp` als äußere Klasse und `Bulb` als eine innere statische Klasse:

Listing 7.1: com/tutego/insel/inner/Lamp.java, Lamp

```
public class Lamp
{
    static String s = "Huhu";
    int i = 1;

    static class Bulb
    {
        void output()
        {
            System.out.println( s );
        }
    }

    public static void main( String[] args )
    {
        Bulb bulb = new Lamp.Bulb(); // oder Lamp.Bulb bulb = ...
        bulb.output();
    }
}
```

Die statische innere Klasse `Bulb` besitzt Zugriff auf alle anderen statischen Eigenschaften der äußeren Klasse `Lamp`, in unserem Fall auf die Variable `s`. Ein Zugriff auf Objektvariablen ist aus der statischen inneren Klasse heraus nicht möglich, da sie als gesonderte Klasse gezählt wird, die im gleichen Paket liegt. Der Zugriff von außen auf innere Klassen gelingt mit der Schreibweise `ÄußereKlasse.InnereKlasse`; der Punkt wird also so verwendet, wie wir es vom Zugriff auf statische Eigenschaften her kennen und auch von

den Paketen als Namensraum gewöhnt sind. Die innere Klasse muss einen anderen Namen als die äußere haben.

Modifizierer und Sichtbarkeit

Erlaubt sind die Modifizierer `abstract`, `final` und einige Sichtbarkeitsmodifizierer. Normale Top-Level-Klassen können `paketsichtbar` oder `public` sein; innere Klassen dürfen ebenfalls `public` oder `paketsichtbar`, alternativ aber auch `protected` oder `private` sein. Eine private statische innere Klasse ist dabei wie eine normale private statische Variable zu verstehen: Sie kann nur von der umschließenden äußeren Klasse gesehen werden, aber nicht von anderen Top-Level-Klassen. `protected` an statischen inneren Typen ermöglicht für den Compiler einen etwas effizienteren Bytecode, ist aber ansonsten nicht in Gebrauch.

Umsetzung der inneren Typen *

Die Sun-Entwickler haben es geschafft, die Einführung von inneren Klassen in Java 1.1 ohne Änderung der virtuellen Maschine über die Bühne zu bringen. Der Compiler generiert aus den inneren Typen nämlich einfach normale Klassendateien, die jedoch mit einigen sogenannten *synthetischen Methoden* ausgestattet sind. Für die inneren Typen generiert der Compiler neue Namen nach dem Muster: `ÄußererTyp$InnererTyp`, das heißt, ein Dollar-Zeichen trennt die Namen von äußerem und innerem Typ. Genauso heißt die entsprechende `.class`-Datei auf der Festplatte.

7.3 Mitglieds- oder Elementklassen

Eine *Mitgliedsklasse* (engl. *member class*), auch *Elementklasse* genannt, ist ebenfalls vergleichbar mit einem Attribut, nur ist sie nicht statisch (statische innere Klassen lassen sich aber auch als *statische Mitgliedsklassen* bezeichnen). Deklarieren wir eine innere Mitgliedsklasse `Room` in `House`:

Listing 7.2: com/tutego/insel/inner/House.java, Ausschnitt

```
class House
{
    private String owner = "Ich";
```

```

class Room
{
    void ok()
    {
        System.out.println( owner );
    }
    // static void error() { }
}

```

Ein Exemplar der Klasse Room hat Zugriff auf alle Eigenschaften von House, auch auf die privaten. Eine wichtige Eigenschaft ist, dass innere Mitgliedsklassen selbst keine statischen Eigenschaften deklarieren dürfen. Der Versuch führt in unserem Fall zu einem Compilerfehler:

The method error cannot be declared static; static methods can only be declared in a static or top level type

7.3.1 Exemplare innerer Klassen erzeugen

Um ein Exemplar von Room zu erzeugen, muss ein Exemplar der äußeren Klasse existieren. Das ist eine wichtige Unterscheidung gegenüber den statischen inneren Klassen aus Abschnitt 7.2. Statische innere Klassen existieren auch ohne Objekt der äußeren Klasse.

In einem Konstruktor oder in einer Objektmethode der äußeren Klassen kann einfach mit dem new-Operator ein Exemplar der inneren Klasse erzeugt werden. Kommen wir von außerhalb – oder von einem statischen Block der äußeren Klasse – und wollen wir Exemplare der inneren Klasse erzeugen, so müssen wir bei Elementklassen sicherstellen, dass es ein Exemplar der äußeren Klasse gibt. Java schreibt eine spezielle Form für die Erzeugung mit new vor, die folgendes allgemeine Format besitzt:

```
referenz.new InnereKlasse(...)
```

Dabei ist referenz eine Referenz vom Typ der äußeren Klasse. Um in der statischen main()-Methode vom Haus ein Room-Objekt aufzubauen, schreiben wir:

Listing 7.3: com/tutego/insel/inner/House.java, main()

```

House h = new House();
Room r = h.new Room();

```

Oder auch in einer Zeile:

```
Room r = new House().new Room();
```

7.3.2 Die this-Referenz

Möchte eine innere Klasse In auf die this-Referenz der sie umgebenden Klasse Out zugreifen, schreiben wir Out.this. Wenn Variablen der inneren Klasse die Variablen der äußeren Klasse überdecken, so schreiben wir Out.this.Eigenschaft, um an die Eigenschaften der äußeren Klasse Out zu gelangen:

Listing 7.4: com/tutego/insel/inner/FurnishedHouse.java,FurnishedHouse

```
class FurnishedHouse
{
    String s = "House";

    class Room
    {
        String s = "Room";

        class Chair
        {
            String s = "Chair";

            void output()
            {
                System.out.println( s );                      // Chair
                System.out.println( this.s );                  // Chair
                System.out.println( Chair.this.s );           // Chair
                System.out.println( Room.this.s );            // Room
                System.out.println( FurnishedHouse.this.s ); // House
            }
        }
    }

    public static void main( String[] args )
    {
```

```

    new FurnishedHouse().new Room().new Chair().output();
}
}

```

Hinweis

Elementklassen können beliebig geschachtelt sein, und da der Name eindeutig ist, gelangen wir mit Klassename.this immer an die jeweilige Eigenschaft.

7

Betrachten wir das obige Beispiel, dann lassen sich Objekte für die inneren Klassen Room und Chair wie folgt erstellen:

```

FurnishedHouse h      = new FurnishedHouse(); // Exemplar von FurnishedHouse
FurnishedHouse.Room r = h.new Room();          // Exemplar von Room in h
FurnishedHouse.Room.Cham c = r.new Chair();    // Exemplar von Chair in r
c.out();                // Methode von Chair

```

Die Qualifizierung mit dem Punkt bei FurnishedHouse.Room.Cham bedeutet nicht automatisch, dass FurnishedHouse ein Paket mit dem Unterpaket Room ist, in dem die Klasse Chair existiert. Die Doppelbelegung des Punkts verbessert die Lesbarkeit nicht gerade, und es droht Verwechslungsgefahr zwischen inneren Klassen und Paketen. Deshalb sollte die Namenskonvention beachtet werden: Klassennamen beginnen mit Großbuchstaben, Paketnamen mit Kleinbuchstaben.

7.3.3 Vom Compiler generierte Klassendateien *

Für das Beispiel House und Room erzeugt der Compiler die Dateien *House.class* und *House\$Room.class*. Damit die innere Klasse an die Attribute der äußeren gelangt, generiert der Compiler automatisch in jedem Exemplar der inneren Klasse eine Referenz auf das zugehörige Objekt der äußeren Klasse. Damit kann die innere Klasse auch auf nicht-statische Attribute der äußeren Klasse zugreifen. Für die innere Klasse ergibt sich folgendes Bild in *House\$Room.class*:

```

class HouseBorder$Room
{
    final House this$0;

    House$Room( House house )
    {

```

```

    this$0 = house;
}
// ...
}

```

Die Variable `this$0` referenziert das Exemplar `House.this`, also die zugehörige äußere Klasse. Die Konstruktoren der inneren Klasse erhalten einen zusätzlichen Parameter vom Typ `House`, um die `this$0`-Variable zu initialisieren. Da wir die Konstruktoren sowieso nicht zu Gesicht bekommen, kann uns das egal sein.

7.3.4 Erlaubte Modifizierer bei äußeren und inneren Klassen

Ist in einer Datei nur eine Klasse deklariert, kann diese nicht privat sein. Private innere Klassen sind aber legal. Statische Hauptklassen gibt es zum Beispiel auch nicht, aber innere statische Klassen sind legitim. Die folgende Tabelle fasst die erlaubten Modifizierer noch einmal kompakt zusammen:

Modifizierer erlaubt auf	äußeren Klassen	inneren Klassen	äußeren Schnittstellen	inneren Schnittstellen
public	ja	ja	ja	ja
protected	nein	ja	nein	ja
private	nein	ja	nein	ja
static	nein	ja	nein	ja
final	ja	ja	nein	nein
abstract	ja	ja	ja	ja

Tabelle 7.2: Erlaubte Modifizierer

7.3.5 Innere Klassen greifen auf private Eigenschaften zu

Es ist ein Sonderfall, dass innere Klassen auf private Eigenschaften der äußeren Klasse zugreifen können. Das folgende Beispiel soll das illustrieren:

Listing 7.5: com/tutego/insel/inner/NotSoPrivate.java, NotSoPrivate

```

public class NotSoPrivate
{
    private static class Family { private String dad, mom; }
}

```

```

public static void main( String[] args )
{
    class Node { private Node next; }

    Node n = new Node();
    n.next = new Node();

    Family ullenboom = new Family();
    ullenboom.dad = "Heinz";
    ullenboom.mom = "Eva";
}
}

```

Eine Klasse `Outsider`, die in der gleichen Compilationseinheit (also Datei) definiert wird, kann schon nicht mehr auf `NotSoPrivate.Family` zugreifen und natürlich auch keine Klasse einer anderen Compilationseinheit.

Zugriffsrechte *

Eine innere Klasse kann auf alle Attribute der äußeren Klasse zugreifen. Da eine innere Klasse als ganz normale Klasse übersetzt wird, stellt sich allerdings die Frage, wie sie das genau macht. Auf öffentliche Variablen kann jede andere Klasse ohne Tricks zugreifen, so auch die innere. Und da eine innere Klasse als normale Klassendatei im gleichen Paket sitzt, kann sie ebenfalls ohne Verrenkungen auf paketsichtbare und `protected`-Eigenschaften der äußeren Klasse zugreifen. Eine innere Klasse kann jedoch auch auf `private` Eigenschaften zurückgreifen, eine Designentscheidung, die sehr umstritten ist und lange kontrovers diskutiert wurde. Doch wie ist das zu schaffen, ohne gleich die Zugriffsrechte des Attributs zu ändern? Der Trick ist, dass der Compiler eine synthetische statische Methode in der äußeren Klasse einführt:

```

class House
{
    private String owner;

    static String access$( House house )
    {

```

```

        return house.owner;
    }
}

```

Die statische Methode `access$0()` ist der `HelpersHelper`, der für ein gegebenes `House` das private Attribut nach außen gibt. Da die innere Klasse einen Verweis auf die äußere Klasse pflegt, gibt sie diesen beim gewünschten Zugriff mit, und die `access$0()`-Methode erledigt den Rest.

Für jedes von der inneren Klasse genutzte private Attribut erzeugt der Compiler eine solche Methode. Wenn wir eine weitere private Variable `int size` hinzunehmen, würde der Compiler ein `int access$1(House)` generieren.



Hinweis

Problematisch ist das bei Klassen, die in ein Paket hineingeschmuggelt werden. Nehmen wir an, `House` liegt im Paket `p1.p2`. Dann kann ein Angreifer seine Klassen auch in ein Paket legen, das `p1.p2` heißt. Da die `access$XXX()`-Methoden paketsichtbar sind, können hineingeschmuggelte Klassen die packetsichtbaren `access$XXX()`-Methoden aufrufen. Es reicht ein Exemplar der äußeren Klasse, um über einen `access$XXX()`-Aufruf auf die privaten Variablen zuzugreifen, die eine innere Klasse nutzt. Glücklicherweise lässt sich gegen eingeschleuste Klassen in Java-Archiven leicht etwas unternehmen – sie müssen nur abgeschlossen werden, was bei Java *sealing* heißt.

7.4 Lokale Klassen

Lokale Klassen sind ebenfalls innere Klassen, die jedoch nicht einfach wie eine Eigenschaft im Rumpf einer Klasse, sondern direkt in Anweisungsblöcken von Methoden, Konstruktoren und Initialisierungsblöcken gesetzt werden. Lokale Schnittstellen sind nicht möglich.

Im folgenden Beispiel deklariert die `main()`-Methode eine innere Klasse mit einem Konstruktor, der auf die finale Variable `j` zugreift:

Listing 7.6: com/tutego/insel/inner/FunInside.java, FunInside

```

public class FunInside
{
    public static void main( String[] args )

```

```

{
    int i = 2;
    final int j = 3;

    class In
    {
        In() {
            System.out.println( j );
//            System.out.println( i ); // ☹ Compiler error because i is not final
        }
    }
    new In();
}
}

```

Die Deklaration der inneren Klasse `In` wird hier wie eine Anweisung eingesetzt. Ein Sichtbarkeitsmodifizierer ist bei inneren lokalen Klassen ungültig, und die Klasse darf keine Klassenmethoden und allgemeinen statischen Variablen deklarieren (finale Konstanten schon).

Jede lokale Klasse kann auf Methoden der äußeren Klasse zugreifen und zusätzlich auf die lokalen Variablen und Parameter, die mit dem Modifizierer `final` als unveränderlich ausgezeichnet sind. Liegt die innere Klasse in einer statischen Methode, kann sie keine Objektmethoden der äußeren Klasse aufrufen.

7.5 Anonyme innere Klassen

Anonyme Klassen gehen noch einen Schritt weiter als lokale Klassen. Sie haben keinen Namen und erzeugen immer automatisch ein Objekt; Klassendeklaration und Objekt-erzeugung sind zu einem Sprachkonstrukt verbunden. Die allgemeine Notation ist folgende:

```
new KlasseOderSchnittstelle() { /* Eigenschaften der inneren Klasse */ }
```

In dem Block geschweifter Klammern lassen sich nun Methoden und Attribute deklarieren oder Methoden überschreiben. Hinter `new` steht der Name einer Klasse oder Schnittstelle:

- new Klassename(Optionale Argumente) { ... }. Steht hinter new ein Klassentyp, dann ist die anonyme Klasse eine Unterklasse von Klassename. Es lassen sich mögliche Argumente für den Konstruktor der Basisklasse angeben (das ist zum Beispiel dann nötig, wenn die Oberklasse keinen Standardkonstruktor deklariert).
- new Schnittstellename() { ... }. Steht hinter new der Name einer Schnittstelle, dann erbt die anonyme Klasse von Object und implementiert die Schnittstelle Schnittstellenname. Implementiert sie nicht die Operationen der Schnittstelle, ist das ein Fehler; wir hätten nichts davon, denn dann hätten wir eine abstrakte innere Klasse, von der sich kein Objekt erzeugen lässt.

Für anonyme innere Klassen gilt die Einschränkung, dass keine zusätzlichen extends- oder implements-Angaben möglich sind. Ebenso sind keine eigenen Konstruktoren möglich und nur Objektmethoden und finale statische Variablen erlaubt.

Wir wollen eine innere Klasse schreiben, die Unterklasse von java.awt.Point ist. Sie soll die `toString()`-Methode überschreiben:

Listing 7.7: com/tutego/insel/inner/InnerToStringPoint.java, main()

```
Point p = new Point( 10, 12 ) {
    @Override public String toString() {
        return "(" + x + "," + y + ")";
    }
};

System.out.println( p ); // (10,12)
```

Da sofort eine Unterklasse von Point aufgebaut wird, fehlt der Name der inneren Klasse. Das einzige Exemplar dieser anonymen Klasse lässt sich über die Variable p weiterverwenden.



Hinweis

Eine innere Klasse kann Methoden der Oberklasse überschreiben, Operationen aus Schnittstellen implementieren und sogar neue Eigenschaften anbieten:

```
String s = new Object() {
    String quote( String s ) {
        return String.format( "'%s'", s );
    }
}.quote( "Juvy" );
System.out.println( s ); // 'Juvy'
```

Hinweis (Forts.)

Der neu deklarierte anonyme Typ hat eine Methode `quote()`, die direkt aufgerufen werden kann. Ohne diesen direkten Aufruf ist die `quote()`-Methode aber unsichtbar, denn der Typ ist ja anonym, und so sind nur die Methoden der Oberklasse (bei uns `Object`) beziehungsweise der Schnittstelle bekannt. (Wir lassen die Tatsache außen vor, dass eine Anwendung mit Reflection auf die Methoden zugreifen kann.)

7.5.1 Umsetzung innerer alterer Klassen *

Auch für innere anonyme Klassen erzeugt der Compiler eine normale Klassendatei. Wir haben gesehen, dass der Java-Compiler bei einer »normalen« inneren Klasse die Notation `ÄußereKlasse$InnereKlasse` wählt. Das klappt bei anonymen inneren Klassen natürlich nicht mehr, da uns der Name der inneren Klasse fehlt. Der Compiler wählt daher folgende Notation für Klassennamen: `InnerToStringDate$1`. Falls es mehr als eine innere Klasse gibt, folgen `$2`, `$3` und so weiter.

7.5.2 Nutzung innerer Klassen für Threads *

Sehen wir uns ein weiteres Beispiel für die Implementierung von Schnittstellen an: Um nebenläufige Programme zu implementieren, gibt es die Klasse `Thread` und die Schnittstelle `Runnable` (für das Beispiel greifen wir vor; Threads werden in Kapitel 12, »Einführung in die nebenläufige Programmierung«, genau beschrieben).

Die Schnittstelle `Runnable` schreibt eine Operation `run()` vor, in die der parallel abzuarbeitende Programmcode gesetzt wird. Das geht gut mit einer inneren anonymen Klasse, die `Runnable` implementiert:

```
new Runnable() {      // Anonyme Klasse extends Object implements Runnable
    public void run() {
        ...
    }
}
```

Das so erzeugte Exemplar kommt in den Konstruktor der Klasse `Thread`. Der Thread wird mit `start()` angekurbelt. Damit folgt zusammengesetzt und mit Implementierung von `run()`:

Listing 7.8: com/tutego/insel/inner/FirstThread, main()

```

new Thread( new Runnable() {
    @Override public void run() {
        for ( int i = 0; i < 10; i++ )
            System.out.printf( "%d ", i );
    }
} ).start();

for ( int i = 0; i < 10; i++ )
    System.out.printf( "%d ", i );

```

In der Ausgabe wird zum Beispiel Folgendes erscheinen (hier komprimiert):

```
0 0 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
```

Der neue Thread beginnt mit der 0 und wird dann unterbrochen. Der `main`-Thread kann in einem Zug 0 bis 9 ausgeben. Danach folgt wieder der erste Thread und kann den Rest ausgeben. Ausführliche Informationen zu Threads vermittelt Kapitel 12.

7.5.3 Konstruktoren innerer alterer Klassen *

Der Compiler setzt anonyme Klassen in normale Klassendateien um. Jede Klasse kann einen eigenen Konstruktor deklarieren, und auch für anonyme Klassen sollte das möglich sein, um Initialisierungscode dort hineinzusetzen. Da aber anonyme Klassen keinen Namen haben, muss für Konstruktoren ein anderer Weg gefunden werden. Hier helfen *Exemplarinitialisierungsblöcke*, also Blöcke in geschweiften Klammern direkt innerhalb einer Klasse, die wir schon in Kapitel 5, »Eigene Klassen schreiben«, vorgestellt haben. Exemplarinitialisierer gibt es ja eigentlich gar nicht im Bytecode, sondern der Compiler setzt den Programmcode automatisch in jeden Konstruktor. Obwohl anonyme Klassen keinen direkten Konstruktor haben können, gelangt doch über den Exemplarinitialisierer Programmcode in den Konstruktor der Bytecode-Datei.

Dazu ein Beispiel: Die anonyme Klasse ist eine Unterklasse von `Point` und initialisiert im Konstruktor einen Punkt mit den Koordinaten $-1, -1$. Aus diesem speziellen Punkt-Objekt lesen wir dann die Koordinaten wieder aus:

Listing 7.9: com/tutego/insel/inner/AnonymousAndInside.java, main()

```
java.awt.Point p = new java.awt.Point() { { x = -1; y = -1; } };
```

```
System.out.println( p.getLocation() ); // java.awt.Point[x=-1,y=-1]

System.out.println( new java.awt.Point( -1, 0 )
{
{
    y = -1;
}
}.getLocation() ); // java.awt.Point[x=-1,y=-1]
```

Gar nicht super() *

Innerhalb eines »anonymen Konstruktors« kann kein `super()` verwendet werden, um den Konstruktor der Oberklasse aufzurufen. Dies liegt daran, dass automatisch ein `super()` in den Initialisierungsblock eingesetzt wird. Die Parameter für die gewünschte Variante des (überladenen) Oberklassen-Konstruktors werden am Anfang der Deklaration der anonymen Klasse angegeben. Dies zeigt das zweite Beispiel:

```
System.out.println( new Point(-1, 0) { { y = -1; } }.getLocation() );
```

Beispiel

zB

Wir initialisieren ein Objekt `BigDecimal`, das beliebig große Ganzzahlen aufnehmen kann. Im Konstruktor der anonymen UnterkLASSE geben wir anschließend den Wert mit der geerbten `toString()`-Methode aus:

```
new java.math.BigDecimal( "12345678901234567890" ) {
    { System.out.println( toString() ); }
};
```

7.6 Zugriff auf lokale Variablen aus lokalen inneren und anonymen Klassen *

Lokale und innere Klassen können auf die lokalen Variablen beziehungsweise Parameter der umschließenden Methode lesend zugreifen, jedoch nur dann, wenn die Variable `final` ist. Verändern können lokale und innere Klassen diese Variablen natürlich nicht, denn `final` verbietet einen zweiten Schreibzugriff.

Ist eine Veränderung nötig, ist ein Trick möglich. Zwei Lösungen bieten sich an:

- die Nutzung eines finalen Feldes der Länge 1, das das Ergebnis aufnehmen kann
- die Nutzung von AtomicXXX-Klassen aus dem java.util.concurrent.atomic-Paket, die ein primitives Element oder eine Referenz aufnehmen

Ein Beispiel:

Listing 7.10: com/tutego/insel/inner/ModifyLocalVariable.java, main()

```
public static void main( String[] args )
{
    final int[] result1 = { 0 };
    final String[] result2 = { null };
    final AtomicInteger result3 = new AtomicInteger();
    final AtomicReference<String> result4 = new AtomicReference<String>();

    System.out.println( result1[0] );      // 0
    System.out.println( result2[0] );      // null
    System.out.println( result3.get() );   // 0
    System.out.println( result4.get() );   // null

    new Object(){
        result1[0] = 1;
        result2[0] = "Der Herr der Felder";
        result3.set( 1 );
        result4.set( "Wurstwasser-Wette" );
    };

    System.out.println( result1[0] );      // 1
    System.out.println( result2[0] );      // Der Herr der Felder
    System.out.println( result3.get() );   // 1
    System.out.println( result4.get() );   // Wurstwasser-Wette
}
```

Die AtomicXXX-Klassen haben eigentlich die Aufgabe, Schreib- und Veränderungsoperationen atomar durchzuführen, können jedoch in diesem Szenario hilfreich sein.

7.7 this in Unterklassen *

Wenn wir ein qualifiziertes `this` verwenden, dann bezeichnet `C.this` die äußere Klasse, also das umschließende Exemplar. Das haben wir schon in Abschnitt 7.3.2, »Die `this`-Referenz«, kennengelernt. Gilt jedoch die Beziehung `C1.C2....Ci.....Cn.`, dann haben wir mit `Ci.this` ein Problem, wenn `Ci` eine Oberklasse von `Cn` ist. Es geht also um den Fall, dass eine textuell umgebende Klasse zugleich auch Oberklasse ist. Das eigentliche Problem besteht darin, dass hier zweidimensionale Namensräume hierarchisch kombiniert werden müssen. Die eine Dimension sind die Bezeichner beziehungsweise Methoden aus den lexikalisch umgebenden Klassen, die andere Dimension sind die ererbten Eigenschaften aus der Oberklasse. Hier sind beliebige Überlappungen und Mehrdeutigkeiten denkbar. Durch diese ungenaue Beziehung zwischen inneren Klassen und Vererbung kam es unter JDK 1.1 und 1.2 zu unterschiedlichen Ergebnissen. Aber das ist ja schon Steinzeit ...

Im nächsten Beispiel soll von der Klasse `Shoe` die innere Klasse `LeatherBoot` den `Shoe` erweitern und die Methode `out()` überschreiben:

Listing 7.11: com/tutego/insel/inner/Shoe.java, Shoe

```
public class Shoe
{
    void out()
    {
        System.out.println( "Ich bin der Schuh des Manitu." );
    }

    class LeatherBoot extends Shoe
    {
        void what()
        {
            Shoe.this.out();
        }
    }

    @Override
    void out()
    {
        System.out.println( "Ich bin ein Shoe.LeatherBoot." );
    }
}
```

```
}

public static void main( String[] args )
{
    new Shoe().new LeatherBoot().what();
}
}
```

Legen wir in der statischen `main()`-Methode ein Objekt der Klasse `LeatherBoot` an, dann landen wir bei `what()` in der Klasse `LeatherBoot`, was `Shoe.this.out()` ausführt. Interessant ist aber, dass hier kein dynamisch gebundener Aufruf an `out()` vom `LeatherBoot`-Objekt erfolgt, sondern die Ausgabe von `Shoe` ist:

Ich bin der Schuh des Manitu.

Die überschriebene Ausgabe von `LeatherBoot` liefert die ähnlich aussehende Anweisung `((Shoe)this).out()`. Vor Version 1.2 kam als Ergebnis immer diese Zeichenkette heraus, aber das ist Geschichte und nur eine historische Randnotiz.



Kapitel 8

Besondere Klassen der Java SE

»Einen Rat befolgen heißt, die Verantwortung verschieben.«
– Johannes Urzidil (1896–1970)

8.1 Vergleichen von Objekten

Sollen Objekte verglichen werden, muss es eine Ordnung dieser Typen geben. Wie sollte das System sonst selbstständig entscheiden können, ob eine Person zum Beispiel kleiner als eine andere Person ist? Weil die eine Person 1,50 Meter groß ist, die andere aber 1,80 Meter, oder weil die eine Person eine Million Euro auf dem Konto hat und die andere nur fünf Euro?¹ Diese Fragen sind wichtig, wenn wir zum Beispiel eine Liste sortieren wollen.

8.1.1 Natürlich geordnet oder nicht?

In Java gibt es zwei unterschiedliche Schnittstellen (in zwei unterschiedlichen Paketen) zur Bestimmung der Ordnung:

- Comparable: Implementiert eine Klasse Comparable, so können sich die Objekte selbst mit anderen Objekten vergleichen. Da die Klassen im Allgemeinen nur ein Sortierkriterium implementieren, wird hierüber eine sogenannte *natürliche Ordnung* (engl. *natural ordering*) realisiert.
- Comparator: Eine implementierende Klasse, die sich Comparator nennt, nimmt zwei Objekte an und vergleicht sie. Ein Comparator für Räume könnte zum Beispiel nach der Anzahl der Personen oder auch nach der Größe in Quadratmetern vergleichen; die Implementierung von Comparable wäre nicht sinnvoll, weil hier nur ein Kriterium natürlich umgesetzt werden kann, ein Raum aber nicht *die* Ordnung hat.

¹ Im 10. Jahrhundert lebte der Großwesir Abdul Kassem Ismael, der immer seine gesamte Bibliothek mit 117.000 Bänden mitführte. Die trainierten 400 Kamele transportierten die Werke in alphabetischer Reihenfolge.

Zusammenfassend lässt sich sagen: Während Comparable üblicherweise nur ein Sortierkriterium umsetzt, kann es viele Extraklassen vom Typ Comparator geben, die jeweils unterschiedliche Ordnungen definieren.

Comparable und Comparator in der Java-API

Eine Implementierung von Comparable findet sich genau dort, wo eine natürliche Ordnung naheliegt, etwa bei:

- String
- BigDecimal, BigInteger, Byte, Character, Double, Float, Integer, Long, Short
- Date
- File, URI
- Enum
- TimeUnit

Von Comparator finden wir in der API-Dokumentation nur `java.text.Collator` vermerkt.

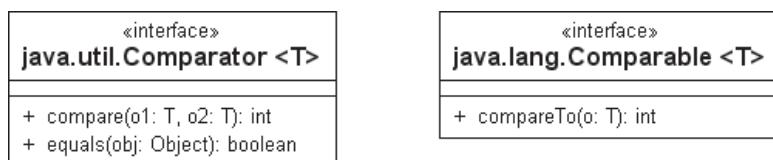


Abbildung 8.1: UML-Diagramm von Comparator und Comparable

8.1.2 Die Schnittstelle Comparable

Die Schnittstelle Comparable kommt aus dem `java.lang`-Paket und deklariert eine Methode:

```

interface java.lang.Comparable<T>
    ■ int compareTo(T o)
        Vergleicht sich mit einem anderen Objekt.

```

Hinweis

Wichtig ist neben einer Implementierung von `compareTo()` auch die passende Realisierung in `equals()`. Sie ist erst dann konsistent, wenn `e1.compareTo(e2) == 0` das gleiche Ergebnis wie `e1.equals(e2)` liefert, wobei `e1` und `e2` den gleichen Typ besitzen. Ein Verstoß

Hinweis (Forts.)

gegen diese Regel kann bei sortierten Mengen schnell Probleme bereiten; ein Beispiel nennt die API-Dokumentation. Auch sollte die hashCode()-Methode korrekt realisiert sein, denn sind Objekte gleich, müssen auch die Hashcodes gleich sein. Und die Gleichheit bestimmen eben equals()/compareTo().

`e.compareTo(null)` sollte eine NullPointerException auslösen, auch wenn `e.equals(null)` die Rückgabe false liefert.

8.1.3 Die Schnittstelle Comparator

Die Schnittstelle Comparator kommt aus dem Paket `java.util` und deklariert:

```
interface java.util.Comparator<T>
```

- `int compare(T o1, T o2)`
Vergleicht zwei Argumente auf ihre Ordnung.
- `boolean equals(Object obj)`
Testet, ob Comparator-Objekte gleich sind. Das testet keine Gleichheit von Objekten! Die Methode muss nicht zwingend implementiert werden, da ja schon `Object` eine Implementierung bereitstellt. Sie steht hier nur, damit eine API-Dokumentation dieses Missverständnis erklärt.

8.1.4 Rückgabewerte kodieren die Ordnung

Der Rückgabewert von `compare()` beim Comparator beziehungsweise `compareTo()` bei Comparable ist `<0`, `=0` oder `>0` und bestimmt so die Ordnung der Objekte. Nehmen wir zwei Objekte `o1` und `o2` an, deren Klassen Comparable implementieren. Dann gilt folgende Übereinkunft:

- `o1.compareTo(o2) < 0` \leftrightarrow `o1` ist »kleiner als« `o2`.
- `o1.compareTo(o2) == 0` \leftrightarrow `o1` ist »gleich« `o2`.
- `o1.compareTo(o2) > 0` \leftrightarrow `o1` ist »größer als« `o2`.

Ein externer Comparator (symbolisch `comp` genannt) verhält sich ähnlich:

- `comp.compare(o1, o2) < 0` \leftrightarrow o1 ist »kleiner als« o2.
- `comp.compare(o1, o2) == 0` \leftrightarrow o1 ist »gleich« o2.
- `comp.compare(o1, o2) > 0` \leftrightarrow o1 ist »größer als« o2.



Tipp

Sollen Objekte mit einem Comparator verglichen werden, aber null-Werte vorher aussortiert werden, so wird seit Java 7 die statische Methode `int compare(T a, T b, Comparator<? super T> c)` aus der Klasse `Objects` nützlich. Die Methode liefert 0, wenn a und b beide entweder null sind oder der Comparator die Objekte a und b für gleich erklärt. Sind a und b beide ungleich null, so ist die Rückgabe `c.compare(a, b)`. Ist nur a oder b gleich null, so hängt es vom Comparator und der Reihenfolge der Parameter ab.

Den größten Raum einer Sammlung finden

Wir wollen Räume ihrer Größe nach sortieren und müssen dafür einen Comparator schreiben (dass Räume Comparable sind, ist nicht angebracht, da es keine natürliche Ordnung für Räume gibt). Daher soll ein externes Comparator-Objekt entscheiden, welches Raum-Objekt nach der Anzahl seiner Quadratmeter größer ist.

Der Raum enthält für das kleine Demoprogramm nur einen parametrisierten Konstruktor und merkt sich dort seine Quadratmeter:

Listing 8.1: com/tutego/insel/util/RoomComparatorDemo.java, Room

```
class Room
{
    int sm;

    Room( int sm )
    {
        this.sm = sm;
    }
}
```

Der spezielle Raum-Comparator ist das eigentlich Interessante:

Listing 8.2: com/tutego/insel/util/RoomComparatorDemo.java, RoomComparator

```
class RoomComparator implements Comparator<Room>
{
    @Override public int compare( Room room1, Room room2 )
    {
        return room1.sm - room2.sm;
    }
}
```

Er bildet die Differenz der Raumgrößen, was eine einfache Möglichkeit darstellt, eine Rückgabe <0, =0 oder >0 zu bekommen. Bei Fließkommazahlen funktioniert das nicht, denn `(int)(0.1 - 0.0)`, `(int)(0.0 - 0.1)` und `(int)(0.0 - 0.0)` ergeben alle 0, wären also gleich – bei Ganzzahlen ist der Vergleich aber in Ordnung. Ab Java 7 lässt sich `Integer.compareTo(room1.sm, room2.sm)` einsetzen.

Mit dem Comparator-Objekt lässt sich eine Raumliste sortieren:

Listing 8.3: com/tutego/insel/util/RoomComparatorDemo.java, RoomComparatorDemo main()

```
List<Room> list = Arrays.asList(new Room(100), new Room(1123), new Room(123));
Collections.sort( list, new RoomComparator() );
System.out.println( list.get(0).sm );      // 100
```

Hinweis

Ist ein Comparator mit einer Datenstruktur – wie dem TreeSet oder der TreeMap – verbunden, muss die Comparator-Klasse Serializable (siehe Kapitel 15, »Einführung in Dateien und Datenströme«) implementieren, wenn auch die Datenstruktur serialisiert werden soll.

8.1.5 Aneinanderreihung von Comparatoren *

Oftmals ist das Ordnungskriterium aus mehreren Bedingungen zusammengesetzt, wie die Sortierung in einem Telefonbuch zeigt. Erst gibt es eine Sortierung nach dem Nachnamen, dann folgt der Vorname. Um diese mit einem Comparator-Objekt zu lösen, müssen entweder alle Einzelvergleiche in ein neues Comparator-Objekt verpackt werden oder einzelne Comparatoren zu einem »Super«-Comparator zusammengebunden werden. Die

zweite Lösung ist natürlich schöner, weil sie die Wiederverwendbarkeit erhöht, denn einzelne Comparatoren können dann leicht für andere Zusammenhänge genutzt werden.

Comparatoren in eine Vergleichskette setzen

Am Anfang steht ein besonderer Comparator, der sich aus mehreren Comparatoren zusammensetzt. Immer dann, wenn ein Teil-Comparator bei zwei Objekten aussagt, dass sie gleich sind (der Vergleich liefert 0 ist), soll der nächste Comparator die Entscheidung fällen – kann er das auch nicht, weil das Ergebnis wieder 0 ist, geht es zum nächsten Vergleicher.

Den Programmcode wollen wir in eine neue Hilfsklasse ComparatorChain setzen:

Listing 8.4: com/tutego/insel/util/ComparatorChain.java

```
package com.tutego.insel.util;

import java.util.*;

/**
 * A {@link Comparator} that puts one or more {@code Comparator}s in a sequence.
 * If a {@code Comparator} returns zero the next {@code Comparator} is taken.
 */
public class ComparatorChain<E> implements Comparator<E>
{
    private List<Comparator<E>> comparatorChain = new ArrayList<Comparator<E>>();

    /**
     * Construct a new comparator chain from the given {@code Comparator}s.
     * The argument is not allowed to be {@code null}.
     * @param comparators Sequence of {@code Comparator}s
     */
    @SafeVarargs // ab Java 7
    public ComparatorChain( Comparator<E>... comparators )
    {
        if ( comparators == null )
            throw new IllegalArgumentException( "Argument is not allowed to be null" );
    }
}
```

```
Collections.addAll( comparatorChain, comparators );  
}  
  
/**  
 * Adds a {@link Comparator} to the end of the chain.  
 * The argument is not allowed to be {@code null}.  
 * @param comparator {@code Comparator} to add  
 */  
public void addComparator( Comparator<E> comparator )  
{  
    if ( comparator == null )  
        throw new IllegalArgumentException( "Argument is not allowed to be null" );  
  
    comparatorChain.add( comparator );  
}  
  
/**  
 * {@inheritDoc}  
 */  
@Override  
public int compare( E o1, E o2 )  
{  
    if ( comparatorChain.isEmpty() )  
        throw new UnsupportedOperationException(  
            "Unable to compare without a Comparator in the chain" );  
  
    for ( Comparator<E> comparator : comparatorChain )  
    {  
        int order = comparator.compare( o1, o2 );  
        if ( order != 0 )  
            return order;  
    }  
  
    return 0;  
}  
}
```

Die ComparatorChain können wir auf zwei Weisen mit den Comparator-Gliedern füttern: einmal zur Initialisierungszeit im Konstruktor und dann später noch über die addComparator()-Methode. Beim Weg über den Konstruktor ist ab Java 7 die Annotation @SafeVarargs zu nutzen, da sonst die Kombination eines Varargs und Generics auf der Nutzerseite zu einer Warnung führt.

Ist kein Comparator intern in der Liste, wird das compare() eine Ausnahme auslösen. Der erste Comparator in der Liste ist auch das Vergleichsobjekt, das zuerst gefragt wird. Liefert er ein Ergebnis ungleich 0, liefert das die Rückgabe der compare()-Methode. Ein Ergebnis gleich 0 führt zur Anfrage des nächsten Comparators in der Liste.

Wir wollen diese ComparatorChain für ein Beispiel nutzen, das eine Liste nach Nach- und Vornamen sortiert.

Listing 8.5: com/tutego/insel/util/ComparatorChainDemo.java

```
package com.tutego.insel.util;

import java.util.*;

public class ComparatorChainDemo
{
    public static class Person
    {
        public String firstname, lastname;

        public Person( String firstname, String lastname )
        {
            this.firstname = firstname;
            this.lastname = lastname;
        }

        @Override public String toString()
        {
            return firstname + " " + lastname;
        }
    }
}
```

```
public final static Comparator<Person>
PERSON_FIRSTNAME_COMPARATOR = new Comparator<Person>() {
    @Override public int compare( Person p1, Person p2 ) {
        return p1.firstname.compareTo( p2.firstname );
    }
};

public final static Comparator<Person>
PERSON_LASTNAME_COMPARATOR = new Comparator<Person>() {
    @Override public int compare( Person p1, Person p2 ) {
        return p1.lastname.compareTo( p2.lastname );
    }
};

public static void main( String[] args )
{
    List<Person> persons = Arrays.asList(
        new Person( "Onkel", "Ogar" ), new Person( "Olga", "Ogar" ),
        new Person( "Peter", "Lustig" ), new Person( "Lara", "Lustig" ) );

    Collections.sort( persons, PERSON_LASTNAME_COMPARATOR );
    System.out.println( persons );

    Collections.sort( persons, PERSON_FIRSTNAME_COMPARATOR );
    System.out.println( persons );

    Collections.sort( persons, new ComparatorChain<Person>(
        PERSON_LASTNAME_COMPARATOR, PERSON_FIRSTNAME_COMPARATOR ) );
    System.out.println( persons );
}
```

Die Ausgabe ist:

```
[Peter Lustig, Lara Lustig, Onkel Ogar, Olga Ogar]
[Lara Lustig, Olga Ogar, Onkel Ogar, Peter Lustig]
[Lara Lustig, Peter Lustig, Olga Ogar, Onkel Ogar]
```

8.2 Wrapper-Klassen und Autoboxing

Die Klassenbibliothek bietet für jeden primitiven Datentyp wie int, double, char spezielle Klassen an. Diese sogenannten *Wrapper-Klassen* (auch *Ummantelungsklassen*, *Mantelklassen* oder *Envelope Classes* genannt) erfüllen drei wichtige Aufgaben:

- Wrapper-Klassen bieten statische Hilfsmethoden zur Konvertierung eines primitiven Datentyps in einen String (Formatierung) und vom String zurück in einen primitiven Datentyp (Parse).
- Die Datenstrukturen wie Listen und Mengen, die in Java Verwendung finden, können nur Referenzen aufnehmen. So stellt sich das Problem, wie primitive Datentypen diesen Containern hinzugefügt werden können. Wrapper-Objekte kapseln einen einfachen primitiven Wert in einem Objekt, sodass eine Referenz existiert, die etwa in einer vorgefertigten Datenstruktur gespeichert werden kann.
- Der Wrapper-Typ ist wichtig bei Generics. Wenn etwa ein spezieller Comparator zwei Fließkommazahlen vergleichen soll, ist eine Comparator-Implementierung mit Comparator<double> nicht korrekt, es muss Comparator<Double> heißen. Primitive Datentypen gibt es auch bei Generics nicht, es kommen immer die Wrapper-Typen zum Einsatz.

Es existieren Wrapper-Klassen zu allen primitiven Datentypen.

Wrapper-Klasse	Primitiver Typ
Byte	byte
Short	short
Integer	int
Long	long
Double	double
Float	float
Boolean	boolean
Character	char

Tabelle 8.1: Wrapper-Klassen und primitive Datentypen



Hinweis

Für void, das kein Datentyp ist, existiert die Klasse Void. Sie deklariert nur die Konstante TYPE vom Typ Class<Void> und ist für Reflection (das Auslesen von Eigenschaften einer Klasse) interessanter.

In diesem Abschnitt wollen wir uns zunächst um das Erzeugen von Wrapper-Objekten kümmern, dann um Methoden, die in allen Wrapper-Klassen vorkommen, und schließlich die individuellen Methoden der einzelnen Wrapper-Klassen vorstellen. Der Klasse Character haben wir uns schon zu Beginn von Kapitel 4, »Der Umgang mit Zeichenketten«, gewidmet, als es um Zeichen und Zeichenketten ging.

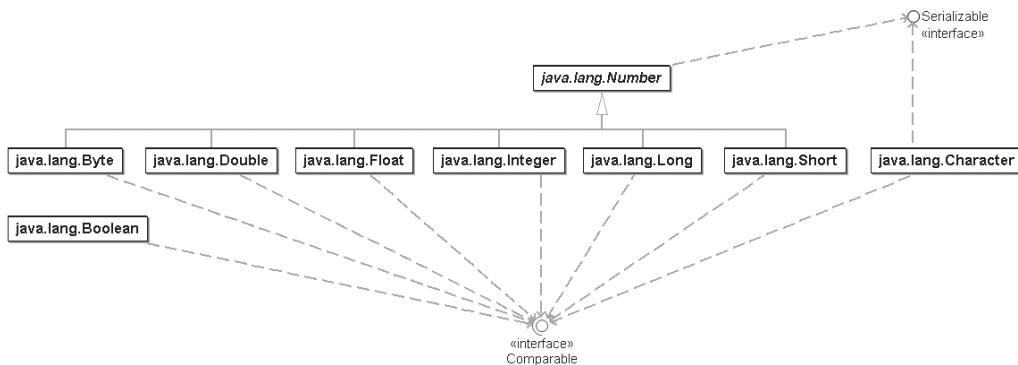


Abbildung 8.2: Vererbungsbeziehung der Wrapper-Klassen (Boolean ist ebenfalls Serializable)

8.2.1 Wrapper-Objekte erzeugen

Wrapper-Objekte lassen sich auf drei Arten aufbauen:

- über statische `valueOf()`-Methoden, denen ein primitiver Ausdruck oder ein String übergeben wird
- über Boxing: Aus einem primitiven Wert erzeugt der Compiler automatisch `valueOf()`-Methodenaufrufe, die das Wrapper-Objekt liefern.
- über Konstruktoren der Wrapper-Klassen

Beispiel

zB

Erzeuge einige Wrapper-Objekte:

```

Integer int1 = Integer.valueOf( "30" ); // valueOf()
Long lng1 = Long.valueOf( 0xCOBOL ); // valueOf()
Integer int2 = new Integer( 29 ); // Konstruktor
Long lng2 = new Long( 0xCOBOL ); // Konstruktor
Double dobl = new Double( 12.3 ); // Konstruktor
Boolean bool = true; // Boxing
Integer int3 = 42; // Boxing
  
```

Nun gibt es also drei Möglichkeiten, an Wrapper-Objekte zu kommen. Ganz selbstverständlich stellt sich die Frage, ob es eine bevorzugte Variante gibt. Boxing ist vom Schreibaufwand her gesehen die kürzeste und im Allgemeinen die beste, weil kompakteste Variante. (Boxing ist allerdings nicht ganz unproblematisch, wie der Abschnitt »Mehr Probleme als Lösungen durch Autoboxing? « zeigt.) Da Boxing auf die `valueOf()`-Methoden zugreift, sind die beiden Varianten semantisch identisch und unterscheiden sich nur im Programmcode, aber nicht im Bytecode. Uns bleibt daher die Lösung »Konstruktor versus `valueOf()`«. Eine statische Methode zum Erzeugen von Objekten einzusetzen ist clever, da anders als ein Konstruktor eine statische Methode Objekte nicht immer neu erzeugen muss, sondern auch auf vorkonstruierte Objekte zurückgreifen kann. Und das ist genau das, was `valueOf()` bei den drei Klassen `Byte`, `Short`, `Integer` und `Long` macht: Stammen die Ganzzahlen aus dem Wertebereich `-128` bis `+127`, so greift `valueOf()` auf vorbereitete Objekte aus einem Cache zurück. Das Ganze klappt natürlich nur, weil Aufrüfer von `valueOf()` ein unveränderliches (engl. *immutable*) Objekt bekommen – ein Wrapper-Objekt kann nach dem Aufbau nicht verändert werden.



Hinweis

In der Wrapper-Klasse `Integer` gibt es drei statische überladene Methoden `getInteger(String)`, `getInteger(String, int)`, `getInteger(String, Integer)`, die von Spracheinsteigern wegen der gleichen Rückgabe und Parameter schnell mit der `valueOf(String)`-Methode verwechselt werden können. Allerdings lesen die `getInteger(String)`-Methoden eine Umgebungsvariable aus und haben somit eine völlig andere Aufgabe als `valueOf(String)`. In der Wrapper-Klasse `Boolean` gibt es mit `getBoolean(String)` Vergleichbares. Die anderen Wrapper-Klassen haben keine Methoden zum Auslesen einer Umgebungsvariable.

Wrapper-Objekte sind *immutable*

Ist ein Wrapper-Objekt erst einmal erzeugt, lässt sich der im Wrapper-Objekt gespeicherte Wert nachträglich nicht mehr verändern. Um dies auch wirklich sicherzustellen, sind die konkreten Wrapper-Klassen allesamt `final`. Die Wrapper-Klassen sind nur als Ummantelung und nicht als vollständiger Datentyp gedacht. Da sich der Wert nicht mehr ändern lässt (er ist ja *immutable*), heißen Objekte mit dieser Eigenschaft auch *Werte-Objekte*. Wollen wir den Inhalt eines `Integer`-Objekts io zum Beispiel um eins erhöhen, so müssen wir ein neues Objekt aufbauen:

```

int i = 12;
Integer io = Integer.valueOf( i );
io = Integer.valueOf( io.intValue() + 1 );
i = io.intValue();

```

Die Variable `io` referenziert nun ein zweites `Integer`-Objekt, und der Wert vom ersten `io`-Objekt mit 12 bleibt unangetastet.

8.2.2 Konvertierungen in eine String-Repräsentation

Alle Wrapper-Klassen bieten statische `toString(value)`-Methoden zur Konvertierung des primitiven Elements in einen String an:

Listing 8.6: com/tutego/insel/wrapper/WrapperToString.java, main()

```

String s1 = Integer.toString( 1234567891 ),
       s2 = Long.toString( 123456789123L ),
       s3 = Float.toString( 12.345678912f ),
       s4 = Double.toString( 12.345678912 ),
       s5 = Boolean.toString( true );

System.out.println( s1 ); // 1234567891
System.out.println( s2 ); // 123456789123
System.out.println( s3 ); // 12.345679
System.out.println( s4 ); // 12.345678912
System.out.println( s5 ); // true

```

Tipp



Ein Java-Idiom² zur Konvertierung ist auch folgende Anweisung:

```
String s = "" + number;
```

Der String erscheint immer in der englisch geschriebenen Variante. So steht bei den Dezimalzahlen ein Punkt statt des uns vertrauten Kommas.

² Es ist wiederum ein JavaScript-Idiom, mit dem Ausdruck `s - 0` aus einem String eine Zahl zu machen, wenn denn die Variable `s` eine String-Repräsentation einer Zahl ist.



Hinweis

Bei der Darstellung von Zahlen ist eine landestypische (länderspezifische) Formatierung sinnvoll. Das kann `printf()` leisten:

```
System.out.printf( "%f", 1000000. );      // 1000000,000000
System.out.printf( "%f", 1234.567 );       // 1234,567000
System.out.printf( "%,.3f", 1234.567 );     // 1.234,567
```

Der Formatspezifizierer für Fließkommazahlen ist `%f`. Die zusätzliche Angabe mit `,.3f` im letzten Fall führt zum Tausenderpunkt und zu drei Nachkommastellen.

toString() als Objekt- und Klassenmethode

Liegt ein Wrapper-Objekt vor, so liefert die Objektmethode `toString()` die String-Repräsentation des Wertes, den das Wrapper-Objekt speichert. Dass es gleichlautende statische Methoden `toString()` und eine Objektmethode `toString()` gibt, sollte uns nicht verwirren; während die Klassenmethode den Arbeitswert zur Konvertierung aus dem Argument zieht, nutzt die Objektmethode den gespeicherten Wert im Wrapper-Objekt.

Anweisungen, die ausschließlich zum Konvertieren über das Wrapper-Objekt gehen, wie `new Integer(v).toString()`, lassen sich problemlos umschreiben in `Integer.toString(v)`. Zudem bietet sich auch die überladene statische Methode `String.valueOf(v)` an, die – eben weil sie überladen ist – für alle möglichen Datentypen deklariert ist (doch nutzt `valueOf(v)` intern auch nur `WrapperKlasse.toString(v)`).

8.2.3 Die Basisklasse Number für numerische Wrapper-Objekte

Alle numerischen Wrapper-Klassen können den gespeicherten Wert in einem beliebigen anderen numerischen Typ liefern. Die Methodennamen setzen sich – wie zum Beispiel `doubleValue()` und `intValue()` – aus dem Namen des gewünschten Typs und `Value` zusammen. Technisch gesehen überschreiben die Wrapper-Klassen `Byte`, `Short`, `Integer`, `Long`, `Float` und `Double` aus einer Klasse `Number`³ die `xxxValue()`-Methoden⁴.

³ Zusätzlich erweitern `BigDecimal` und `BigInteger` die Klasse `Number` und haben damit ebenfalls die `xxxValue()`-Methoden. In Java 5 kamen `AtomicInteger` und `AtomicLong` hinzu, die aber nicht immutable sind wie die anderen Klassen.

⁴ Nur die Methoden `byteValue()` und `shortValue()` sind nicht abstrakt und müssen nicht überschrieben werden. Diese Methoden rufen `intValue()` auf und konvertieren den Wert über eine Typanpassung auf `byte` und `short`.

```
abstract class java.lang.Number
implements Serializable
```

- `byte byteValue()`
Liefert den Wert der Zahl als byte.
- `abstract double doubleValue()`
Liefert den Wert der Zahl als double.
- `abstract float floatValue()`
Liefert den Wert der Zahl als float.
- `abstract int intValue()`
Liefert den Wert der Zahl als int.
- `abstract long longValue()`
Liefert den Wert der Zahl als long.
- `short shortValue()`
Liefert den Wert der Zahl als short.

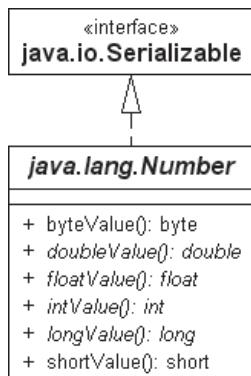


Abbildung 8.3: UML-Diagramm für Number

Hinweis

Wenn die Operandentypen beim Bedingungsoperator unterschiedlich sind, gibt es ganz automatisch eine Anpassung:

```

boolean b = true;
System.out.println( b ? 1 : 0.1 ); // 1.0
System.out.println( !b ? 1 : 0.1 ); // 0.1
  
```



Hinweis (Forts.)

Der Ergebnistyp ist double, sodass die Ganzzahl 1 als 1.0, also als Fließkommazahl, ausgegeben wird. Die gleiche Anpassung nimmt der Compiler bei Wrapper-Typen vor, die er unboxt und konvertiert:

```
Integer i = 1;
Double d = 0.1;
System.out.println( b ? i : d );    // 1.0
System.out.println( !b ? i : d );   // 0.1
```

Während diese Ausgabe eigentlich klar ist, kann es zu einem Missverständnis kommen, wenn das Ergebnis nicht einfach ausgegeben, sondern als Verweis auf das resultierende Wrapper-Objekt zwischengespeichert wird. Da der Typ im Beispiel entweder Integer oder Double ist, kann der Ergebnistyp nur der Obertyp Number sein:

```
Number n1 = b ? i : d;
System.out.println( n1 );           // 1.0
System.out.println( n1 == i );      // false
```

Die Programmlogik und Ausgabe ist natürlich genauso wie vorher, doch Entwickler könnten annehmen, dass der Compiler keine Konvertierung durchführt, sondern entweder das originale Integer- oder das Double-Objekt referenziert; das macht er aber nicht. Die Variable n1 referenziert hier ein Integer-unboxedes-double-konvertiertes-Double-unboxedes Objekt, und so sind die Referenzen von i und n1 überhaupt nicht identisch. Wenn der Compiler hier wirklich die Originalobjekte zurückliefern soll, muss entweder das Integer- oder das Double-Objekt explizit auf Number gebracht werden, sodass damit das Unboxing ausgeschaltet wird und der Bedingungsoperator nur noch von beliebigen nicht zu interpretierenden Referenzen ausgeht:

```
Number n2 = b ? (Number) i : d;    // oder Number n2 = b ? i : (Number) d;
System.out.println( n2 );          // 1
System.out.println( n2 == i );     // true
```

8.2.4 Vergleiche durchführen mit compare(), compareTo(), equals()

Haben wir zwei Ganzzahlen 1 und 2 vor uns, so ist es trivial zu sagen, dass 1 kleiner als 2 ist. Bei Fließkommazahlen ist das ein wenig komplizierter, da es hier »Sonderzahlen« wie Unendlich oder eine negative beziehungsweise positive 0 gibt. Da insbesondere

Vergleichsalgorithmen die Beantwortung der Frage, ob zwei Werte `a` und `b` kleiner, größer oder gleich sind, erwarten, gibt es zwei Typen von Methoden in den Wrapper-Klassen:

- Sie implementieren eine Objektmethode `compareTo()`. Die Methode ist nicht zufällig in der Klasse, denn Wrapper-Klassen implementieren die Schnittstelle `Comparable` (wir haben die Schnittstelle schon am Anfang des Kapitels kurz vorgestellt).
- Wrapper-Klassen besitzen statische `compare(x,y)`-Methoden.

Die Rückgabe der Methoden ist ein `int`, und es kodiert, ob ein Wert größer, kleiner oder gleich ist.

Beispiel

zB

Teste verschiedene Werte:

```
System.out.println( Integer.compare(1, 2) );           // -1
System.out.println( Integer.compare(1, 1) );           // 0
System.out.println( Integer.compare(2, 1) );           // 1

System.out.println( Double.compare(2.0, 2.1) );        // -1
System.out.println( Double.compare(Double.NaN, 0) ); // 1

System.out.println( Boolean.compare(true, false) );   // 1
System.out.println( Boolean.compare(false, true) );   // -1
```

Ein `true` ist »größer« als ein `false`.

Tabelle 8.2 fasst die Methoden der Wrapper-Klassen zusammen.

Klasse	Methode aus Comparable	Statische Methode compare()
Byte	<code>int compareTo(Byte anotherByte)</code>	<code>int compare(int x, int y)</code>
Short	<code>int compareTo(Short anotherShort)</code>	<code>int compare(short x, short y)</code>
Float	<code>int compareTo(Float anotherFloat)</code>	<code>int compare(float f1, float f2)</code>
Double	<code>int compareTo(Double anotherDouble)</code>	<code>int compare(double d1, double d2)</code>
Integer	<code>int compareTo(Integer anotherInteger)</code>	<code>int compare(int x, int y)</code>
Long	<code>int compareTo(Long anotherLong)</code>	<code>int compare(long x, long y)</code>

Tabelle 8.2: Methoden der Wrapper-Klassen

Klasse	Methode aus Comparable	Statische Methode compare()
Character	int compareTo(Character anotherChar)	int compare(char x, char y)
Boolean	int compareTo(Boolean b)	int compare(boolean x, boolean y)

Tabelle 8.2: Methoden der Wrapper-Klassen (Forts.)

Die Implementierung einer statischen Methode `WrapperKlasse.compare()` ist äquivalent zu `WrapperKlasse.valueOf(x).compareTo(WrapperKlasse.valueOf(y))`.



Hinweis

Nur die genannten Wrapper-Klassen besitzen eine statische `compare()`-Methode. Es ist kein allgemeingültiges Muster, dass, wenn eine Klasse `Number` erweitert und `Comparable` implementiert, sie dann auch eine statische `compare()`-Methode hat. So erweitern zum Beispiel die Klassen `BigInteger` und `BigDecimal` die Oberklasse `Number` und implementieren `Comparable`, aber eine statische `compare()`-Methode bieten sie trotzdem nicht.

Gleichheitstest über equals()

Alle Wrapper-Klassen überschreiben aus der Basisklasse `Object` die Methode `equals()`. So lässt sich testen, ob zwei Wrapper-Objekte den gleichen Wert haben, auch wenn die Wrapper-Objekte nicht identisch sind.



Beispiel

Die Ergebnisse einiger Gleichheitstests:

<code>Boolean.TRUE.equals(Boolean.TRUE)</code>	<code>true</code>
<code>Integer.valueOf(1).equals(Integer.valueOf(1))</code>	<code>true</code>
<code>Integer.valueOf(1).equals(Integer.valueOf(2))</code>	<code>false</code>
<code>Integer.valueOf(1).equals(Long.valueOf(1))</code>	<code>false</code>
<code>Integer.valueOf(1).equals(1L)</code>	<code>false</code>

Es ist wichtig zu wissen, dass der Parametertyp von `equals()` immer `Object` ist, aber die Typen gleich sein müssen, da andernfalls schon automatisch der Vergleich falsch ergibt. Das zeigen das vorletzte und das letzte Beispiel. Die `equals()`-Methode aus Zeile 3 und 4 lehnt jeden Vergleich mit einem Nicht-Integer ab, und ein `Long` ist eben kein `Integer`. In der letzten Zeile kommt Boxing zum Einsatz, daher sieht der Programmcode kürzer aus, aber entspricht dem aus der vorletzten Zeile.

Die Objektmethode `equals()` der Wrapper-Klassen ist auch eine kurze Alternative zu `wrapperObject.compareTo(anotherWrapperObjekt) == 0`.

Ausblick

Dass die Wrapper-Klassen `equals()` implementieren, ist gut, denn so können Wrapper-Objekte problemlos in Datenstrukturen wie einer `ArrayList` untergebracht und wieder gefunden werden. Und dass Wrapper-Objekte auch `Comparable` sind, ist ebenfalls prima für Datenstrukturen wie `TreeSet`, die – ohne extern gegebene `Comparator`-Klassen für Vergleiche – eine natürliche Ordnung der Elemente erwarten.



8.2.5 Die Klasse Integer

Die Klasse `Integer` kapselt den Wert einer Ganzzahl vom Typ `int` in einem Objekt und bietet Konstanten statische Methoden zur Konvertierung in einen String und zurück sowie weitere Hilfsmethoden mathematischer Natur an.

Um aus dem String eine Zahl zu machen, nutzen wir `Integer.parseInt(String)`.

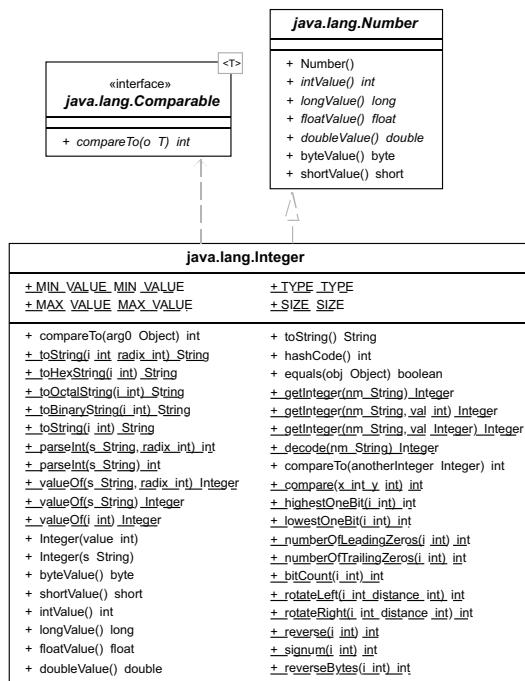


Abbildung 8.4: Klassendiagramm von Integer

zB Beispiel

Konvertiere die Ganzzahl 38.317, die als String vorliegt, in eine Ganzzahl:

```
String number = "38317";
int integer = 0;
try
{
    integer = Integer.parseInt( number );
}
catch ( NumberFormatException e )
{
    System.err.println( "Fehler beim Konvertieren von " + number );
}
System.out.println( integer );
```

Die NumberFormatException ist eine nicht-geprüfte Exception – Genauereres dazu liefert das Kapitel 6, »Exceptions« –, muss also nicht zwingend in einem try-Block stehen.

Die statische Methode `Integer.parseInt(String)` konvertiert einen String in `int`, und die Umkehrmethode `Integer.toString(int)` liefert einen String. Weitere Varianten mit unterschiedlicher Basis wurden schon in Abschnitt 4.5, »Konvertieren zwischen Primitiven und Strings«, vorgestellt.

```
final class java.lang.Integer
extends Number
implements Comparable<Integer>
```

- `static int parseInt(String s)`
Erzeugt aus der Zeichenkette die entsprechende Zahl. Die Basis ist 10.
- `static int parseInt(String s, int radix)`
Erzeugt die Zahl mit der gegebenen Basis.
- `static String toString(int i)`
Konvertiert die Ganzzahl in einen String und liefert sie zurück.

`parseInt()` erlaubt keine länderspezifischen Tausendertrennzeichen, etwa in Deutschland den Punkt oder im angelsächsischen Raum das Komma.

8.2.6 Die Klassen Double und Float für Fließkommazahlen

Die Klassen Double und Float haben wie die anderen Wrapper-Klassen eine Doppelfunktionalität. Sie kapseln zum einen eine Fließkommazahl als Objekt und bieten zum anderen statische Utility-Methoden. Wir kommen in Kapitel 18, »Bits und Bytes und Matematisches«, noch genauer auf die mathebezogenen Objekt- und Klassenmethoden zurück.

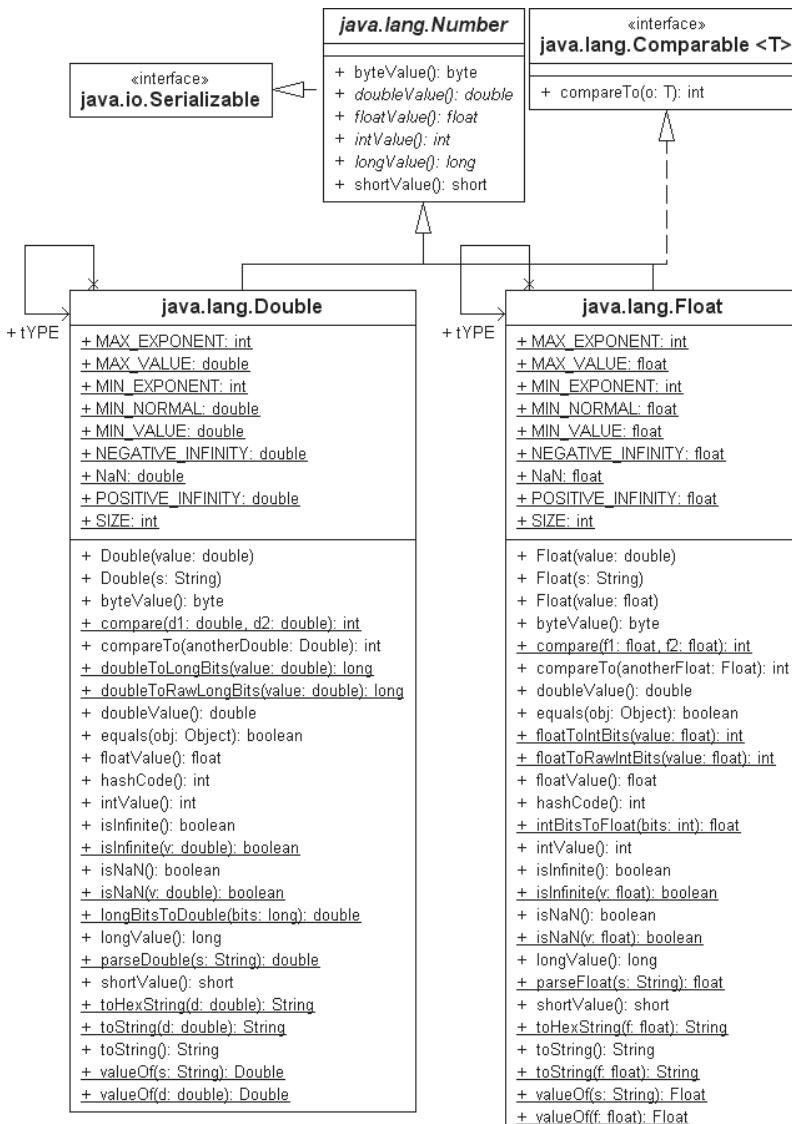


Abbildung 8.5: UML-Diagramm der Fließkommaklassen

8.2.7 Die Long-Klasse

Integer und Long sind im Prinzip API-gleich, nur ist der kleinere Datentyp int durch long ersetzt.

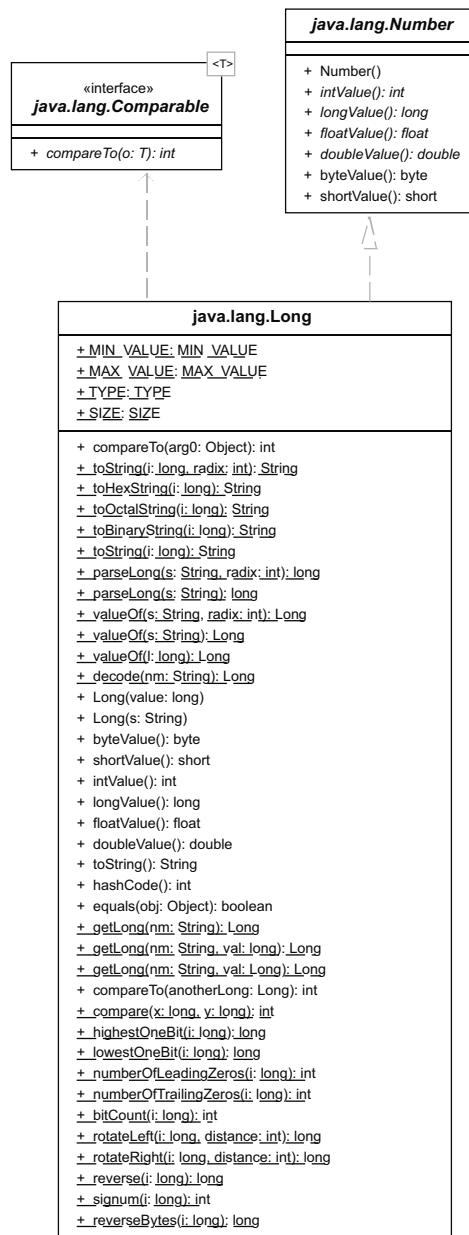


Abbildung 8.6: Klassendiagramm von Long

8.2.8 Die Boolean-Klasse

Die Klasse Boolean kapselt den Datentyp boolean. Sie deklariert als Konstanten zwei Boolean-Objekte: TRUE und FALSE.

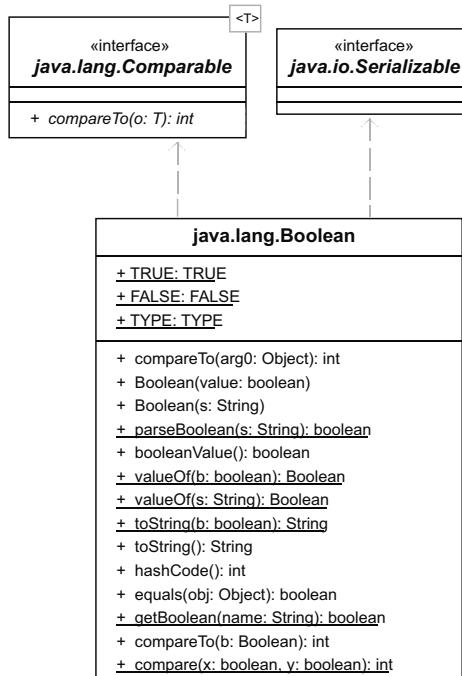


Abbildung 8.7: Klassendiagramm von Boolean

```

final class java.lang.Boolean
  implements Comparable<Boolean>, Serializable
  
```

- static final Boolean FALSE
- static final Boolean TRUE
Konstanten für Wahrheitswerte.
- Boolean(boolean value)
Erzeugt ein neues Boolean-Objekt. Dieser Konstruktor sollte nicht verwendet werden, stattdessen sollten Boolean.TRUE oder Boolean.FALSE eingesetzt werden. Boolean-Objekte sind immutable, und ein new Boolean(value) ist unnötig.
- Boolean(String s)
Parst den String und liefert ein neues Boolean-Objekt zurück.

- static Boolean valueOf(String s)

Parst den String und gibt die Wrapper-Typen Boolean.TRUE oder Boolean.FALSE zurück. Die statische Methode hat gegenüber dem Konstruktor Boolean(boolean) den Vorteil, dass sie immer das gleiche immutable Wahr- oder Falsch-Objekt (Boolean.TRUE oder Boolean.FALSE) zurückgibt, anstatt neue Objekte zu erzeugen. Daher ist es selten nötig, den Konstruktor aufzurufen und immer neue Boolean-Objekte aufzubauen.

- public static boolean parseBoolean(String s)

Parst den String und liefert entweder true oder false.

Der Konstruktor Boolean(String name) beziehungsweise die beiden statischen Methoden valueOf(String name) und parseBoolean(String name) nehmen Strings entgegen und führen im JDK den Test `name != null && name.equalsIgnoreCase("true")` durch. Das heißt zum einen, dass die Groß-/Kleinschreibung unwichtig ist, und zum anderen, dass Dinge wie »false« (mit Leerzeichen), »falsch« oder »Ostereier« automatisch false ergeben, wobei »TRUE« oder »True« dann true liefert.



Tipp

Würde jeder Entwickler ausschließlich die Konstanten Boolean.TRUE und Boolean.FALSE nutzen, so wären bei lediglich zwei Objekten Vergleiche mit == beziehungsweise != in Ordnung. Da es aber einen Konstruktor für Boolean-Objekte gibt – und es ist durchaus diskussionswürdig, warum es überhaupt Konstruktoren für Wrapper-Klassen gibt –, ist die sicherste Variante ein `boolean1.equals(boolean2)`. Wir können eben nicht wissen, ob eine Bibliotheksmethode wie Boolean.isNice() auf die zwei Konstanten zurückgreift oder immer wieder neue Boolean-Objekte aufbaut.

8.2.9 Autoboxing: Boxing und Unboxing

Neu seit Java 5 ist das *Autoboxing*. Dies bedeutet, dass primitive Datentypen und Wrapper-Objekte bei Bedarf ineinander umgewandelt werden. Ein Beispiel:

```
int      i = 4711;
Integer j = i;           // steht für j = Integer.valueOf(i)    (1)
int      k = j;           // steht für k = j.intValue()        (2)
```

Die Anweisung in (1) nennt sich *Boxing* und erstellt automatisch ein Wrapper-Objekt, sofern erforderlich. Schreibweise (2) ist das *Unboxing* und steht für das Beziehen des Elements aus dem Wrapper-Objekt. Das bedeutet: Überall dort, wo der Compiler ein pri-

mitives Element erwartet, aber ein Wrapper-Objekt vorhanden ist, entnimmt er den Wert mit einer passenden `xxxValue()`-Methode aus dem Wrapper.

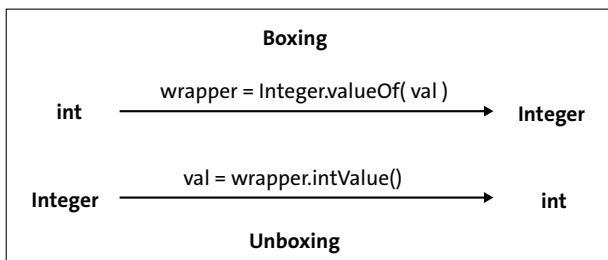


Abbildung 8.8: Autoboxing von int/Integer

Die Operatoren `++`, `--` *

Der Compiler konvertiert nach festen Regeln, und auch die Operatoren `++`, `--` sind erlaubt:

```

Integer i = 12;
i = i + 1;           // (1)
i++;                // (2)
System.out.println( i ); // 14

```

Wichtig ist, dass weder (1) noch (2) das Original-`Integer`-Objekt mit der 12 ändern (alle Wrapper-Objekte sind immutable), sondern `i` nur andere `Integer`-Objekte für 13 und 14 referenziert.

Boxing für dynamische Datenstrukturen (Ausblick)

Am angenehmsten ist die Schreibweise dann, wenn etwa in Datenstrukturen primitive Elemente abgelegt werden sollen:

```

List list = new ArrayList();
list.add( Math.sin(Math.PI / 4) );

```

Allerdings warnt der Compiler hier; er wünscht sich eine typisierte Liste, also:

```
List<Double> list = new ArrayList<Double>();
```

Leider ist es so, dass der Typ der Liste tatsächlich mit dem Wrapper-Typ `Double` festgelegt werden muss und nicht mit dem Primitivtyp `double`. Aber vielleicht ändert sich das ja noch irgendwann.

Keine Konvertierung der null-Referenz zu 0

Beim Unboxing führt der Compiler beziehungsweise die Laufzeitumgebung keine Konvertierung von `null` auf 0 durch. Mit anderen Worten: Bei der folgenden versuchten Zuweisung gibt es keinen Compilerfehler, aber zur Laufzeit eine `NullPointerException`:

```
int n = (Integer) null;           // ☣ java.lang.NullPointerException zur Laufzeit
```



Hinweis

In `switch`-Blöcken sind `int`, Aufzählungen und `Strings` als Typen erlaubt. Bei Ganzzahlen führt der Compiler automatisch Konvertierungen und Unboxing auf `int` durch. Beim Unboxing gibt es aber die Gefahr einer `NullPointerException`:

```
Integer integer = null;
switch ( integer ) // ☣ NullPointerException zur Laufzeit
{ }
```

Autoboxing bei Feldern?

Da primitive Datentypen und Wrapper-Objekte durch Autoboxing automatisch konvertiert werden, fällt im Alltag der Unterschied nicht so auf. Bei Feldern ist der Unterschied jedoch augenfällig, und hier kann Java keine automatische Konvertierung durchführen. Denn auch, wenn zum Beispiel `char` und `Character` automatisch ineinander umgewandelt werden, so sind Arrays nicht konvertierbar. Eine Feldinitialisierung der Art

```
Character[] chars = { 'S', 'h', 'a' };
```

enthält zwar rechts dreimal Boxing von `char` in `Character`, und eine automatische Umwandlung auf der Ebene der Elemente ist gültig, sodass

```
char first = chars[ 0 ];
```

natürlich gilt, aber die Feld-Objekte lassen sich nicht ineinander überführen. Folgendes ist nicht korrekt:

```
char[] sha = chars; // ☣ Compilerfehler!
```

Die Typen `char[]` und `Character[]` sind also zwei völlig unterschiedliche Typen, und eine Überführung ist nicht möglich (von den Problemen mit `null`-Referenzen einmal ganz abgesehen). So muss in der Praxis zwischen den unterschiedlichen Typen konvertiert werden, und bedauerlicherweise bietet die Java-Standardbibliothek hierfür keine Methoden. Die Lücke füllt etwa die Open-Source-Bibliothek *Apache Commons Lang*

(<http://commons.apache.org/lang/>) mit der Klasse `ArrayUtils`, die mit `toObject()` und `toPrimitive()` die Konvertierungen durchführt.

Mehr Probleme als Lösungen durch Autoboxing? *

Mit dem Autoboxing ist eine Reihe von Unregelmäßigkeiten verbunden, die der Programmierer beachten muss, um Fehler zu vermeiden. Eine Unregelmäßigkeit hängt mit dem Unboxing zusammen, das der Compiler immer dann vornimmt, wenn ein Ausdruck einen primitiven Wert erwartet. Wenn kein primitives Element erwartet wird, wird auch kein Unboxing vorgenommen:

Listing 8.7: com/tutego/insel/wrapper/Autoboxing.java, main() Teil 1

```
Integer i1 = new Integer( 1 );
Integer i2 = new Integer( 1 );

System.out.println( i1 >= i2 );    // true
System.out.println( i1 <= i2 );    // true
System.out.println( i1 == i2 );    // false
```

Der Vergleich mit `==` ist weiterhin ein Referenzvergleich, und es findet kein Unboxing auf primitive Werte statt, sodass es auf einen Vergleich von primitiven Werten hinausliefe. Daher muss bei zwei unterschiedlichen `Integer`-Objekten dieser Vergleich immer falsch sein. Das ist natürlich problematisch, da die alte mathematische Regel »aus $i \leq j$ und $i \geq j$ folgt automatisch $i == j$ « nicht mehr gilt. Wenn es die unterschiedlichen `Integer`-Objekte für gleiche Werte nicht gäbe, bestünde dieses Problem nicht.

Es ist interessant zu wissen, was nun genau passiert, wenn das Boxing eine Zahl in ein Wrapper-Objekt umwandelt. In dem Moment wird nicht der Konstruktor aufgerufen, sondern die statische `valueOf()`-Methode:

Listing 8.8: com/tutego/insel/wrapper/Autoboxing.java, main() Teil 2

```
Integer n1 = new Integer( 10 );
Integer n2 = Integer.valueOf( 10 );
Integer n3 = 10;
Integer n4 = 10;

System.out.println( n1 == n2 );    // false
System.out.println( n2 == n3 );    // true
System.out.println( n1 == n3 );    // false
System.out.println( n3 == n4 );    // true
```

Die Widersprüche hören aber damit nicht auf. Das JDK versucht das Problem mit dem `==` damit zu lösen, dass über Boxing gebildete Integer-Objekte einem Pool entstammen. Da jedoch nicht beliebig viele Wrapper-Objekte aus einem Pool kommen können, gilt die Gleichheit der über Boxing gebildeten Objekte nur in einem ausgewählten Wertebereich zwischen `-128` und `+127`, also dem Wertebereich eines Bytes:

Listing 8.9: com/tutego/insel/wrapper/Autoboxing.java, main() Teil 3

```
Integer j1 = 2;
Integer j2 = 2;
System.out.println( j1 == j2 );    // true
Integer k1 = 127;
Integer k2 = 127;
System.out.println( k1 == k2 );    // true
Integer l1 = 128;
Integer l2 = 128;
System.out.println( l1 == l2 );    // false
Integer m1 = 1000;
Integer m2 = 1000;
System.out.println( m1 == m2 );    // false
```

Wir betonten bereits, dass auch bei Wrapper-Objekten der Vergleich mit `==` immer ein Referenz-Vergleich ist. Da `2` und `127` im Wertebereich zwischen `-128` und `+127` liegen, entstammen die entsprechenden Integer-Objekte dem Pool. Das gilt für `128` und `1.000` nicht; sie sind immer neue Objekte. Damit ergibt auch der `==`-Vergleich `false`.



Abschlussfrage

Welche Ausgabe kommt auf den Bildschirm? Ändert sich etwas, wenn `i` und `j` auf `222` stehen?

```
Integer i = 1, j = 1;
boolean b = (i <= j && j <= i && i != j);
System.out.println( b );
```

8.3 Object ist die Mutter aller Klassen

`java.lang.Object` ist die oberste aller Eltern-Klassen. Somit spielt diese Klasse eine ganz besondere Rolle, da alle anderen Klassen automatisch Unterklassen sind und die Methoden erben beziehungsweise überschreiben.

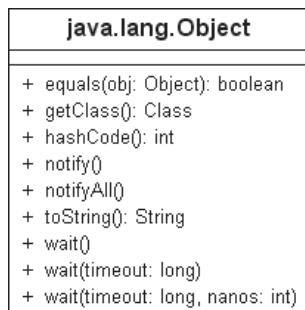


Abbildung 8.9: UML-Diagramm der absoluten Basisklasse Object

8.3.1 Klassenobjekte

Zwar ist jedes Objekt ein Exemplar einer Klasse – doch was ist eine Klasse? In einer Sprache wie C++ existieren Klassen nicht zur Laufzeit, und der Compiler übersetzt die Klassenstruktur in ein ausführbares Programm. Im absoluten Gegensatz dazu steht Smalltalk: Diese Laufzeitumgebung verwaltet Klassen selbst als Objekte. Diese Idee, Klassen als Objekte zu repräsentieren, übernimmt auch Java – Klassen sind Objekte vom Typ `java.lang.Class`.

`class java.lang.Object`

- final Class<? extends Object> getClass()

Liefert die Referenz auf das Klassenobjekt, die das Objekt konstruiert hat. Das `Class`-Objekt ist immer eindeutig in der JVM, sodass auch mehrere Anfragen an `getClass()` immer dasselbe `Class`-Objekt liefern.

Beispiel

zB

Die Objektmethode `getName()` eines `Class`-Objekts liefert den Namen der Klassen:

```
System.out.println( "Klaviklack".getClass().getName() ); // java.lang.String
```

Klassen-Literale

Ein *Klassen-Literal* (engl. *class literal*) ist ein Ausdruck der Form `Datentyp.class`, wobei Datentyp entweder eine Klasse, eine Schnittstelle, ein Feld oder ein primitiver Typ ist. Beispiele sind `String.class`, `Integer.class` oder `int.class` (was nicht mit `Integer.class` identisch ist). Der Ausdruck ist immer vom Typ `Class`. Bei primitiven Typen liefert die Schreibweise `primitiverTyp.class` das gleiche Ergebnis wie `WrapperTyp.TYPE`; es ist also `Integer.TYPE` identisch mit `int.class`. `Class`-Objekte spielen insbesondere bei dynamischen Abfragen über die sogenannte Reflection eine Rolle. Zur Laufzeit können so beliebige Klassen geladen, Objekte erzeugt und Methoden aufgerufen werden.

8.3.2 Objektidentifikation mit `toString()`

Jedes Objekt sollte sich durch die Methode `toString()` mit einer Zeichenkette identifizieren und den Inhalt der interessanten Attribute als Zeichenkette liefern.

zB Beispiel

Die Klasse `Point` implementiert `toString()` so, dass der Rückgabestring die Koordinaten enthält:

```
System.out.println( new java.awt.Point() ); // java.awt.Point[x=0,y=0]
```

Das Angenehme ist, dass `toString()` automatisch aufgerufen wird, wenn die Methoden `print()` oder `println()` mit einer Objektreferenz als Argument aufgerufen werden. Ähnliches gilt für den Zeichenkettenoperator `+` mit einer Objektreferenz als Operand:

Listing 8.10: com/tutego/insel/object/tostring/Player.java, Player

```
public class Player
{
    String name;
    int    age;

    @Override
    public String toString()
    {
        return getClass().getName() + "[name=" + name + ",age=" + age + "]";
    }
}
```

Die Ausgabe mit den Zeilen

Listing 8.11: com/tutego/insel/object/tostring/PlayerToStringDemo.java, main()

```
Player tinkerbelle = new Player();
tinkerbelle.name    = "Tinkerbelle";
tinkerbelle.age     = 32;
System.out.println( tinkerbelle.toString() );
System.out.println( tinkerbelle );
```

ist damit:

```
com.tutego.insel.object.tostring.Player[name=Tinkerbelle,age=32]
com.tutego.insel.object.tostring.Player[name=Tinkerbelle,age=32]
```

Bei einer eigenen Implementierung müssen wir darauf achten, dass die Sichtbarkeit public ist, da `toString()` in der Oberklasse `Object` öffentlich vorgegeben ist und wir in der Unterklassie die Sichtbarkeit nicht einschränken können. Zwar bringt die Spezifikation nicht deutlich zum Ausdruck, dass `toString()` nicht null als Rückgabe liefern darf, doch ist dann der Leerstring "" allemal besser. Die Annotation `@Override` macht das Überschreiben deutlich.

Standardimplementierung

Neue Klassen sollten `toString()` überschreiben. Ist dies nicht der Fall, gelangt das Programm zur Standardimplementierung in `Object`, wo lediglich der Klassenname und der wenig aussagekräftige Hash-Wert hexadezimal zusammengebunden werden.

```
public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Zur Methode:

class java.lang.Object

- `String toString()`
Liefert eine String-Repräsentation des Objekts aus Klassenname und Hash-Wert.

Zwar sagt der Hash-Wert selbst wenig aus, allerdings ist er ein erstes Indiz dafür, dass bei Klassen, die keine `toString()`- und `hashCode()`-Methoden überschreiben, zwei Referenzen nicht identisch sind.

zB Beispiel

Ein Objekt der class A {} wird gebildet, und `toString()` liefert die ID, die ausgegeben wird:

```
System.out.println( new A().toString() ); // A@923e30
System.out.println( new A().toString() ); // A@130c19b
```

`toString()`-Methode generieren lassen

Die Methode eignet sich gut zum Debugging, doch ist das manuelle Tippen der Methoden lästig. Zwei Lösungen vereinfachen das Implementieren der Methode `toString()`:

- Eclipse und NetBeans können standardmäßig über das Kontextmenü eine `toString()`-Methode anhand ausgewählter Attribute generieren. Das Gleiche gilt im Übrigen auch für `equals()` und `hashCode()`.
- Die Zustände werden automatisch über Reflection ausgelesen. Hier führt *Apache Commons Lang* (<http://jakarta.apache.org/commons/lang/>) auf den richtigen Weg.

8.3.3 Objektgleichheit mit `equals()` und Identität

Ob zwei Referenzen dasselbe Objekt repräsentieren, stellt der Vergleichsoperator `==` fest. Er testet die Identität, nicht jedoch automatisch die inhaltliche Gleichheit. Am Beispiel mit Zeichenketten ist das gut zu erkennen: Ein Vergleich mit `firstname == "Christian"` hat im Allgemeinen einen falschen, unbeabsichtigten Effekt, obwohl er syntaktisch korrekt ist. An dieser Stelle sollte der inhaltliche Vergleich stattfinden: Stimmen alle Zeichen der Zeichenkette überein?

Eine `equals()`-Methode sollte Objekte auf Gleichheit prüfen. So besitzt das `String`-Objekt eine Implementierung, die jedes Zeichen vergleicht:

```
String firstname = "Christian";
if ( firstname.equals( "Christian" ) )
    ...

```

```
class java.lang.Object
```

- `boolean equals(Object o)`

Testet, ob das andere Objekt gleich dem eigenen ist. Die Gleichheit definiert jede Klasse für sich anders, doch die Basisklasse vergleicht nur die Referenzen `o == this`.

equals()-Implementierung aus Object und Unterklassen

Die Standardimplementierung aus der absoluten Oberklasse `Object` kann über die Gleichheit von speziellen Objekten nichts wissen und testet lediglich die Referenzen:

```
public boolean equals( Object obj )
{
    return this == obj;
}
```

Überschreibt eine Klasse `equals()` nicht, ist das Ergebnis von `o1.equals(o2)` gleichwertig mit `o1 == o2`. Unterklassen überschreiben diese Methode, um einen inhaltlichen Vergleich mit ihren Zuständen vorzunehmen. Die Methode ist in Unterklassen gut aufgehoben, denn jede Klasse benötigt eine unterschiedliche Logik, um festzulegen, wann ein Objekt gleich einem anderen Objekt ist.

Nicht jede Klasse implementiert eine eigene `equals()`-Methode, sodass die Laufzeitumgebung unter Umständen ungewollt bei `Object` und seinem Referenzenvergleich landet. Dies hat ungeahnte Folgen, und diese Fehleinschätzung kommt leider bei Exemplaren der Klassen `StringBuffer` und `StringBuilder` vor, die kein eigenes `equals()` implementieren. Wir haben dies bereits in Kapitel 4, »Der Umgang mit Zeichenketten«, erläutert.

equals()-Methode überschreiben

Bei selbst deklarierten Methoden ist Vorsicht geboten, da wir genau auf die Signatur achten müssen. Die Methode muss ein `Object` akzeptieren und `boolean` zurückgeben. Wird diese Signatur falsch verwendet, kommt es statt zu einer *Überschreibung* der Methode zu einer *Überladung* und bei einer Rückgabe ungleich `boolean` zu einer zweiten Methode mit gleicher Signatur, was Java nicht zulässt (Java erlaubt bisher keine kovarianten Parametertypen). Um das Problem zu minimieren, sollte die Annotation `@Override` an `equals()` angeheftet sein.

Die `equals()`-Methode stellt einige Anforderungen:

- Heißt der Vergleich `equals(null)`, so ist das Ergebnis immer `false`.
- Kommt ein `this` hinein, lässt sich eine Abkürzung nehmen und `true` zurückliefern.
- Das Argument ist zwar vom Typ `Object`, aber dennoch vergleichen wir immer konkrete Typen. Eine `equals()`-Methode einer Klasse `X` wird sich daher nur mit Objekten vom Typ `X` vergleichen lassen. Eine spannende Frage ist, ob `equals()` auch Unterklassen von `X` beachten soll.

- Eine Implementierung von `equals()` sollte immer eine Implementierung von `hashCode()` bedeuten, denn wenn zwei Objekte `equals()`-gleich sind, müssen auch die Hashwerte gleich sein. Bei einer geerbten `hashCode()`-Methode aus `Object` ist das aber nicht in jedem Fall erfüllt.



Hinweis

Der Datentyp für den Parameter in der `equals()`-Methode ist immer `Object` und niemals etwas anderes, da sonst `equals()` nicht überschrieben, sondern überladen wird. Folgendes für eine Klasse `Player` ist also falsch:

```
public class Player
{
    private int age;
    public boolean equals( Player that ) { return this.age == that.age; }
}
```

Im Vokabular der Informatiker gesprochen: Java unterstützt bisher keine kovarianten Typ-Parameter, wohl aber seit Java 5 kovariante Rückgabetypen. Daher ist es gut, die Annotation `@Override` zu setzen, denn sie schlägt Alarm, falls wir glauben, eine Methode zu überschreiben, es dann aber doch nicht tun.

Beispiel einer eigenen `equals()`-Methode

Die beiden ersten Punkte sind leicht erfüllbar, und ein Beispiel für einen `Club` mit den Attributten `numberOfPersons` und `sm` (für die Quadratmeter) ist schnell implementiert:

```
@Override
public boolean equals( Object o )
{
    if ( o == null )
        return false;

    if ( o == this )
        return true;

    Club that = (Club) o;
```

```

        return this.numberOfPersons == that.numberOfPersons
        && this.sm == that.sm;
    }
}

```

Diese Lösung erscheint offensichtlich, führt aber spätestens bei einem Nicht-Club-Objekt zu einer `ClassCastException`. Das Problem scheint schnell behoben:

```

if ( ! o instanceof Club )
    return false;
}

```

Jetzt sind wir auf der sicheren Seite, aber das Ziel ist noch nicht ganz erreicht.

Hinweis

Die `equals()`-Methode sollte bei nicht passenden Typen immer `false` zurückgeben und keine Ausnahme auslösen.

Das Problem der Symmetrie *

Zwar funktioniert die aufgeführte Implementierung bei finalen Klassen schön, doch bei Unterklassen ist die Symmetrie gebrochen. Warum? Ganz einfach: `instanceof` testet Typen in der Hierarchie, liefert also auch dann `true`, wenn das an `equals()` übergebene Argument eine Unterkorrekte von `Club` ist. Diese Unterkorrekte wird wie die Oberkorrekte die gleichen Attribute haben, sodass – aus der Sicht von `Club` – alles in Ordnung ist. Nehmen wir einmal die Variablen `club` und `superClub` an, die die Typen `Club` und `SuperClub` – die fiktive Unterkorrekte von `Club` – besitzen. Sind beide Objekte gleich, so ergibt `club.equals(superClub)` das Ergebnis `true`. Drehen wir den Spieß um, und fragen wir, was `superClub.equals(club)` ergibt. Zwar haben wir `SuperClub` nicht implementiert, nehmen aber an, dass dort eine `equals()`-Methode steckt, die nach dem gleichen `instanceof`-Schema implementiert wurde wie `Club`. Dann wird dort bei einem Test Folgendes ausgeführt: `club instanceof superClub` – und das ist `false`. Damit wird aber die Fallunterscheidung mit `return false` beendet. Fassen wir zusammen:

```

club.equals( superClub ) == true
superClub.equals( club ) == false

```

Das darf nicht sein, und zur Lösung dürfen wir nicht `instanceof` verwenden, sondern müssen fragen, ob der Typ exakt ist. Das geht mit `getClass()`. Korrekt ist daher Folgendes:

Listing 8.12: com/tutego/insel/object>equals/Club.java, Club

```
public class Club
{
    int numberOfPersons;
    int sm;

    @Override
    public boolean equals( Object o )
    {
        if ( o == null )
            return false;

        if ( o == this )
            return true;

        if ( !o.getClass().equals(getClass()) )
            return false;

        Club that = (Club) o;

        return      this.numberOfPersons == that.numberOfPersons
                && this.sm    == that.sm;
    }

    @Override
    public int hashCode()
    {
        return (31 + numberOfPersons) * 31 + sm;
    }
}
```

Die hashCode()-Methode besprechen wir in Abschnitt 8.3.5, »Hashcodes über hashCode() liefern« – sie steht nur der Vollständigkeit halber hier, da equals() und hashCode() immer Hand in Hand gehen sollten.

Es ist günstig, bei erweiterten Klassen ein neues `equals()` anzugeben, sodass auch die neuen Attribute in den Test einbezogen werden. Bei `hashCode()`-Methoden müssen wir eine ähnliche Strategie anwenden, was wir hier nicht zeigen wollen.

Einmal gleich, immer gleich *

Ein weiterer Aspekt von `equals()`⁵ ist der folgende: Das Ergebnis muss während der gesamten Lebensdauer eines Objekts gleich bleiben. Ein kleines Problem steckt dabei in `equals()` der Klasse `URL`, die vergleicht, ob zwei URL-Adressen auf die gleiche Ressource zeigen. In der Dokumentation heißt es:

»Two URL objects are equal if they have the same protocol, reference equivalent hosts, have the same port number on the host, and the same file and fragment of the file.«

Hostnamen gelten als gleich, wenn entweder beide auf dieselbe IP-Adresse zeigen oder – falls eine nicht auflösbar ist – beide Hostnamen gleich (ohne Groß-/Kleinschreibung) oder `null` sind. Da hinter den URLs `http://tutego.de/` und `http://java-tutor.com/` aber letztendlich `http://www.tutego.com/` steckt, liefert `equals()` die Rückgabe `true`:

Listing 8.13: com/tutego/insel/object>equals/UrlEquals.java, main()

```
URL url1 = new URL( "http://tutego.com/" );
URL url2 = new URL( "http://www.tutego.com/" );
System.out.println( url1.equals(url2) ); // true
```

Die dynamische Abbildung der Hostnamen auf die IP-Adresse des Rechners kann aus mehreren Gründen problematisch sein:

- Der (menschliche) Leser erwartet intuitiv etwas anderes.
- Wenn keine Netzwerkverbindung besteht, wird keine Namensauflösung durchgeführt, und der Vergleich liefert `false`. Die Rückgabe sollte jedoch nicht davon abhängig sein, ob eine Netzwerkverbindung besteht.
- Dass die beiden URLs auf den gleichen Server zeigen, könnte sich zur Laufzeit ändern.

8.3.4 Klonen eines Objekts mit `clone()` *

Zum Replizieren eines Objekts gibt es oft zwei Möglichkeiten:

⁵ Eine korrekte Implementierung der Methode `equals()` bildet eine Äquivalenzrelation. Lassen wir die `null`-Referenz außen vor, ist sie reflexiv, symmetrisch und transitiv.

- einen Konstruktor (auch *Copy-Constructor* genannt), der ein vorhandenes Objekt als Vorlage nimmt, ein neues Objekt anlegt und die Zustände kopiert
- eine öffentliche `clone()`-Methode

Was eine Klasse nun anbietet, ist in der API-Dokumentation zu erfahren.

zB Beispiel

Erzeuge ein Punkt-Objekt, und klonen es:

```
java.awt.Point p = new java.awt.Point( 12, 23 );
java.awt.Point q = (java.awt.Point) p.clone();
System.out.println( q );                                // java.awt.Point[x=12,y=23]
```

Mehr als 300 Klassen der Java-Bibliothek unterstützen ein `clone()`, das ein neues Exemplar mit dem gleichen Zustand zurückgibt. Eine überschriebene Methode kann den Typ der Rückgabe dank kovalenter Rückgabetypen anpassen. Die `clone()`-Methode bei `java.awt.Point` bleibt allerdings bei `Object`.

Felder erlauben standardmäßig `clone()`. Speichern die Arrays jedoch nicht-primitive Werte, liefert `clone()` nur eine flache Kopie, was bedeutet, dass das neue Feldobjekt, der Klon, die exakt gleichen Objekte wie das Original referenziert und die Einträge selbst nicht klonen.

clone() aus `java.lang.Object`

Da `clone()` nicht automatisch unterstützt wird, stellt sich die Frage, wie wir `clone()` für unsere Klassen mit geringstem Aufwand umsetzen können. Einfach `clone()` aufzurufen funktioniert jedoch nicht, da die Methode `protected` ist, also erst einmal nicht sichtbar ist.

```
class java.lang.Object
```

- `protected Object clone() throws CloneNotSupportedException`
Liefert eine Kopie des Objekts.

Eine eigene `clone()`-Methode

Eigene Klassen überschreiben die `protected`-Methode `clone()` aus der Oberklasse `Object` und machen sie `public`. Für die Implementierung kommen zwei Möglichkeiten in Betracht:

- Wir könnten von Hand ein neues Objekt anlegen, alle Attribute kopieren und die Referenz auf das neue Objekt zurückgeben.
- Das Laufzeitsystem soll selbst eine Kopie anlegen, und diese geben wir zurück. Lösung zwei verkürzt die Entwicklungszeit und ist auch spannender.

Um das System zum Klonen zu bewegen, müssen zwei Dinge getan werden:

- Der Aufruf `super.clone()` stößt die Methode `clone()` aus `Object` an und veranlasst so die Laufzeitumgebung, ein neues Objekt zu bilden und die nicht-statischen Attribute zu kopieren. Die Methode kopiert elementweise die Daten des aktuellen Objekts in das neue. Die Methode ist in der Oberklasse `protected`, aber das ist der Trick: Nur Unterklassen können `clone()` aufrufen, keiner sonst.
- Die Klasse implementiert die Markierungsschnittstelle `Cloneable`. Falls von außen ein `clone()` auf einem Objekt aufgerufen wird, dessen Klasse nicht `Cloneable` implementiert, ist das Ergebnis eine `CloneNotSupportedException`. Natürlich implementiert `Object` die Schnittstelle `Cloneable` *nicht* selbst, denn sonst hätten ja Klassen schon automatisch diesen Typ, was sinnlos wäre.

`clone()` gibt eine Referenz auf das neue Objekt zurück, und wenn es keinen freien Speicher mehr gibt, folgt ein `OutOfMemoryError`.

Nehmen wir an, für ein Spiel sollen `Player` geklont werden:

Listing 8.14: com/tutego/insel/object/clone/Player.java

```
package com.tutego.insel.object.clone;

public class Player implements Cloneable
{
    public String name;
    public int    age;

    @Override
    public Player clone()
    {
        try
        {
            return (Player) super.clone();
        }
        catch (CloneNotSupportedException e) {
```

```
// Kann eigentlich nicht passieren, da Cloneable  
throw new InternalError();  
}  
}  
}
```

Da es seit Java 5 kovariante Rückgabetypen gibt, gibt `clone()` nicht lediglich `Object`, sondern den Untertyp `Player` zurück.

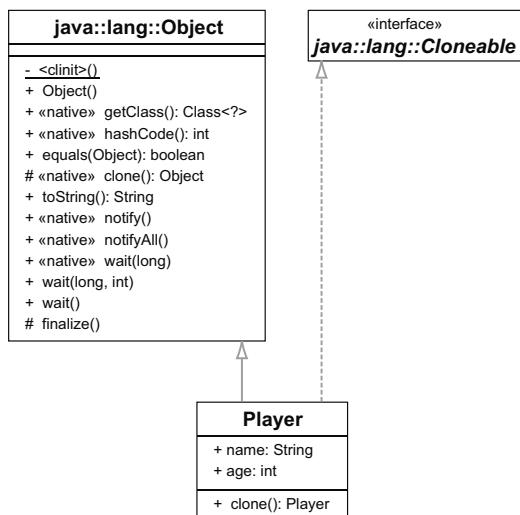


Abbildung 8.10: Player erweitert Object und implementiert Cloneable

Testen wir die Klasse etwa so:

Listing 8.15: com/tutego/insel/object/clone/PlayerCloneDemo.java, main()

```
Player susi = new Player();
susi.age  = 29;
susi.name = "Susi";
Player dolly = susi.clone();
System.out.println( dolly.name + " ist " + dolly.age ); // Susi ist 29
```



Hinweis

Erben wir von einer Klasse mit implementierter `clone()`-Methode, die ihrerseits mit `super.clone()` arbeitet, bekommen wir von oben gleich auch die eigenen Zustände kopiert.

clone() und equals()

Die Methode `clone()` und die Methode `equals()` hängen, wie auch `equals()` und `hashCode()`, miteinander zusammen. Wenn die `clone()`-Methode überschrieben wird, sollte auch `equals()` angepasst werden, denn ohne ein überschriebenes `equals()` bleibt Folgendes in `Object` stehen:

```
public boolean equals( Object obj )
{
    return (this == obj);
}
```

Das bedeutet aber, dass ein geklontes Objekt – das ja im Allgemeinen ein neues Objekt ist – durch seine neue Objektidentität nicht mehr `equals()`-gleich zu seinem Erzeuger ist. Formal heißt das: `o.clone().equals(o) == false`. Diese Semantik dürfte nicht erwünscht sein.

Flach oder tief?

Das `clone()` vom System erzeugt standardmäßig eine *flache Kopie* (engl. *shallow copy*). Bei untergeordneten Objekten werden nur die Referenzen kopiert, und das Originalobjekt sowie die Kopie verweisen anschließend auf dieselben untergeordneten Objekte (sie verwenden diese gemeinsam). Wenn zum Beispiel die Bedienung ein Attribut für einen Arbeitgeber besitzt und eine Kopie der Bedienung erzeugt wird, wird der Klon auf den gleichen Arbeitgeber zeigen. Bei einem Arbeitgeber mag das noch stimmig sein, aber bei Datenstrukturen ist mitunter eine *tiefe Kopie* (engl. *deep copy*) erwünscht. Bei dieser Variante werden rekursiv alle Unterobjekte ebenfalls geklont. Die Bibliotheksimplementierung hinter `Object` kann das nicht.

Keine Klonen bitte!

Wenn wir weder flach noch tief kopieren wollen, aber aus der Oberklasse eine `clone()`-Implementierung erben, ist folgende Lösung denkbar, um das Klonen zu unterbinden: Wir überschreiben `clone()`, lösen aber eine `CloneNotSupportedException` aus und signalisieren so, dass wir nicht geklont werden wollen. Allerdings gibt es ein Problem, wenn eine Klasse schon die `clone()`-Methode überschreibt und dabei die Signatur verändert. In `Object` sieht der Methodenkopf so aus:

```
public class Object
{
```

```

protected native Object clone() throws CloneNotSupportedException;
...
}

```

Eine Unterklasse überschreibt `clone()` und lässt in der Regel das `throws CloneNotSupportedException` weg. Bei `Point2D` (von der `Point` die `clone()`-Methode erbt) ist Folgendes abzulesen:

```

public abstract class Point2D implements Cloneable
{
    public Object clone()
    ...
}

public class Point extends Point2D implements java.io.Serializable {
    ...
}

```

Erbt eine Klasse eine `clone()`-Methode, von der `throws CloneNotSupportedException` entfernt wurde, so kann sie diese nicht mehr einführen – Unterklassen können `throws`-Klausen weglassen aber nicht hinzufügen. Folgendes ist daher *nicht* möglich:

```

public class PointSubclass extends java.awt.Point
{
    @Override // aus Point2D
    public Object clone() throws CloneNotSupportedException // ☹ Compilerfehler!
    ...
}

```

Da die Signatur keine Exception-Klausel mehr aufnehmen kann, müssen wir einen Trick nutzen und die `CloneNotSupportedException` in eine Laufzeitausnahme verpacken:

Listing 8.16: com/tutego/insel/object/clone/ColoredPoint.java, ColoredPoint

```

public class ColoredPoint extends java.awt.Point
{
    public int rgb;

    @Override // aus Point2D
    public Object clone()
    {

```

```

        throw new RuntimeException( new CloneNotSupportedException() );
    }
}

```

Ein Klonversuch führt zu etwas wie:

```

Exception in thread "main" java.lang.RuntimeException:
java.lang.CloneNotSupportedException
at com.tutego.insel.object.clone.ColoredPoint.clone(ColoredPoint.java:10)
at ...
Caused by: java.lang.CloneNotSupportedException
... 2 more

```

8

Technisch löst es dann unser Problem, allerdings sollten wir uns bewusst sein, dass wir ein Verhalten, das vorher erlaubt war, nun »abschalten«. Unterklassen sollten Verhalten nicht wegnehmen.

8.3.5 Hashcodes über hashCode() liefern *

Die Methode `hashCode()` soll zu jedem Objekt eine möglichst eindeutige Integerzahl (so-wohl positiv als auch negativ) liefern, die das Objekt identifiziert. Die Ganzzahl heißt *Hashcode* beziehungsweise *Hash-Wert*, und `hashCode()` ist die Implementierung einer *Hash-Funktion*. Nötig sind Hashcodes, wenn die Objekte in speziellen Datenstrukturen untergebracht werden, die nach dem Hashing-Verfahren arbeiten. Datenstrukturen mit Hashing-Algorithmen bieten einen effizienten Zugriff auf ihre Elemente. Die Klasse `java.util.HashMap` implementiert eine solche Datenstruktur.

<pre> class java.lang.Object ■ int hashCode() Liefert den Hash-Wert eines Objekts. Die Basisklasse Object implementiert die Methode nativ. </pre>

Spieler mit Hash-Funktion

Im folgenden Beispiel soll die Klasse `Player` die Methode `hashCode()` aus `Object` überschreiben. Um die Objekte erfolgreich in einem Assoziativspeicher abzulegen, ist ebenfalls `equals()` nötig, was die Klasse `Player` ebenfalls implementiert:

Listing 8.17: com/tutego/insel/object/hashcode/Player.java

```
package com.tutego.insel.object.hashcode;

public class Player
{
    String name;
    int age;
    double weight;

    /**
     * Returns a hash code value for this {@code Player} object.
     *
     * @return A hash code value for this object.
     *
     * @see java.lang.Object#equals(java.lang.Object)
     * @see java.util.HashMap
     */
    @Override public int hashCode()
    {
        int result = 31 + age;
        result = 31 * result + ((name == null) ? 0 : name.hashCode());
        long temp = Double.doubleToLongBits( weight );
        result = 31 * result + (int) (temp ^ (temp >>> 32));

        return result;
    }

    /**
     * Determines whether or not two players are equal. Two instances of
     * {@code Player} are equal if the values of their {@code name}, {@code age}
     * and {@code weight} member fields are the same.
     *
     * @param that an object to be compared with this {@code Player}
     *
     * @return {@code true} if the object to be compared is an instance of
     *         {@code Player} and has the same values; {@code false} otherwise.
     */
}
```

```
*/  
@Override public boolean equals( Object that )  
{  
    if ( this == that )  
        return true;  
  
    if ( that == null )  
        return false;  
  
    if ( getClass() != that.getClass() )  
        return false;  
  
    if ( age != ((Player)that).age )  
        return false;  
  
    if ( name == null )  
        if ( ((Player)that).name != null )  
            return false;  
        else if ( !name.equals( ((Player)that).name ) )  
            return false;  
  
    return !( Double.doubleToLongBits( weight ) !=  
              Double.doubleToLongBits( ((Player)that).weight ) );  
}  
}
```

Testen können wir die Klasse etwa mit den folgenden Zeilen:

Listing 8.18: com/tutego/insel/object/hashcode/PlayerHashCodeDemo.java, main()

```
Player bruceWants = new Player();  
bruceWants.name = "Bruce Wants";  
bruceWants.age = 32;  
bruceWants.weight = 70.3;
```

```
Player brucelii = new Player();  
brucelii.name = "Bruce Lii";
```

```

bruceLii.age = 32;
bruceLii.weight = 70.3;;

System.out.println( bruceWants.hashCode() );           // -340931147
System.out.println( bruceLii.hashCode() );             // 301931244
System.out.println( System.identityHashCode( bruceWants ) ); // 1671711
System.out.println( System.identityHashCode( bruceLii ) ); // 11394033
System.out.println( bruceLii.equals( bruceWants ) );   // false

bruceWants.name = "Bruce Lii";
System.out.println( bruceWants.hashCode() );           // 301931244
System.out.println( bruceLii.equals( bruceWants ) );   // true

```

Die statische Methode `System.identityHashCode()` liefert für ein Objekt den Hashcode, wie es die Standard-Implementierung von `Object` liefern würde, wenn wir sie nicht überschrieben hätten.



Hinweis

Da der Hashcode negativ sein kann, sind Ausdrücke wie `array[o.hashCode() % array.length()]` problematisch. Ist `o.hashCode()` negativ, ist auch das Ergebnis vom Restwert negativ, und die Folge ist eine `ArrayIndexOutOfBoundsException`.



Eclipse kann die Methoden `hashCode()` und `equals()` automatisch generieren, wenn wir im Kontextmenü unter `SOURCE • GENERATE HASHCODE() AND EQUALS()` auswählen.

Tiefe oder flache Vergleiche/Hash-Werte

Referenziert ein Objekt Unterobjekte (etwa eine Person ein String-Objekt für den Namen – keine primitiven Datentypen), so geben die Methoden `equals()` und `hashCode()` den Vergleich beziehungsweise die Berechnung des Hashcodes an das referenzierte Unterobjekt weiter (wenn es denn nicht `null` ist). Ablesen können wir das an folgendem Ausschnitt unserer `equals()`-Methode:

Listing 8.19: com/tutego/insel/object/hashcode/Player.java, equals() Ausschnitt

```

if ( name == null )
if ( ((Player)that).name != null )
    return false;

```

```

else if ( !name.equals( ((Player)that).name ) )
    return false;

```

Es ist demnach die Aufgabe der String-Klasse (`name` ist vom Typ `String`), den Gleichheitstest vorzunehmen. Das heißt, dass zwei Personen problemlos `equals()`-gleich sein können, auch wenn sie zwei nicht-identische, aber `equals()`-gleiche String-Objekte referenzieren.

Auch bei `hashCode()` ist diese Delegation an das referenzierte Unterobjekt abzulesen:

Listing 8.20: com/tutego/insel/object/hashcode/Player.java, `hashCode()` Ausschnitt

```
result = 31 * result + ((name == null) ? 0 : name.hashCode());
```

Dass eine `equals()`-Methode beziehungsweise `hashCode()` einer Klasse den Vergleich beziehungsweise die Hashcode-Berechnung nicht an die Unterobjekte delegiert, sondern selbst umsetzt, ist unüblich.

equals()- und hashCode()-Berechnung bei (mehrdimensionalen) Arrays

Einen gewissen Sonderfall bei `equals()/hashCode()` nehmen mehrdimensionale Arrays ein. Mehrdimensionale Arrays sind nichts anderes als Arrays von Arrays. Das erste Array für die erste Dimension referenziert jeweils auf Unterarrays für die zweite Dimension. Wichtig wird diese Realisierung bei der Frage, wie diese Verweise der ersten Dimension nun bei `equals()` betrachtet werden sollen. Denn hier stellt sich die Frage, ob die Unterarrays von zwei zu testenden Arrays nur identisch oder auch gleich sein dürfen. Diese Frage hatten wir schon in Abschnitt 3.8.19, »Die Klasse Arrays zum Vergleichen, Füllen, Suchen, Sortieren nutzen«, angesprochen.

Enthält unsere Klasse ein Array und soll es in einem `equals()` mit berücksichtigt werden, so sind prinzipiell drei Varianten zum Umgang mit diesem Array möglich. Felder selbst einfach mit `==` wie primitive Werte zu vergleichen ist keine gute Lösung, da Arrays Objekte sind, die wie Strings nicht einfach mit `==` zu vergleichen sind. Während allerdings Objekte ein `equals()` haben, bieten Arrays keine eigene `equals()`-Methode, sondern diese ist in die Utility-Klasse `Arrays` gewandert. Hier gibt es jedoch zwei Methoden, die infrage kämen. `Arrays.equals(Object[] a, Object[] a2)` geht jedes Element von `a`, also bei mehrdimensionalen Arrays jede Referenz auf ein Unterarray durch, und testet, ob es identisch mit einem zweiten Feld `a2` ist. Wenn also zwei gleiche, aber nicht-identische Hauptarrays identische Unterarrays besitzen, liefert `Arrays.equals()` die Rückgabe `true`, aber nicht, wenn die Unterarrays zwar gleich, aber nicht identisch sind. Spielt das eine Rolle, so ist `Arrays.deepEquals()` die passende Methode, denn sie fragt immer mit `equals()` die Unterarrays ab.

Bei der Berechnung des Hash-Werts gibt es eine vergleichbare Frage. Die `Arrays`-Klasse bietet zur Berechnung des Hash-Werts eines ganzen Arrays die Methoden `Arrays.hashCode()` und `Arrays.deepHashCode()`. Die erste Methode fragt jedes Unterelement über die von `Object` angebotene Methode `hashCode()` nach dem Hash-Wert. Nehmen wir ein mehrdimensionales Array an. Dann ist das Unterelement ebenfalls ein Feld. `Arrays.hashCode()` wird dann wie erwähnt nur die `hashCode()`-Methode auf dem Feld-Objekt aufrufen, während `Arrays.deepHashCode()` auch in das Unterarray hinabsteigt und so lange `Arrays.deepHashCode()` auf allen Unterfeldern aufruft, bis ein `equals()`-Vergleich auf einem Nicht-Feld möglich ist.

Was heißt das nun für unsere `equals()/hashCode()`-Methode? Üblich ist der Einsatz von `Arrays.equals()` und nicht von `Arrays.deepEquals()`, genauso wie `Arrays.hashCode()` üblicher als `Arrays.deepHashCode()` ist.

Das folgende Beispiel zeigt das in der Anwendung. Die Methoden wurden von Eclipse generiert und etwas kompakter geschrieben:

Listing 8.21: com/tutego/insel/object/hashcode/Chess.java, Chess

```
char[][] chessboard = new char[8][8];

@Override public int hashCode()
{
    return 31 + Arrays.hashCode( chessboard );
}

@Override public boolean equals( Object obj )
{
    if ( this == obj )
        return true;
    if ( obj == null )
        return false;
    if ( getClass() != obj.getClass() )
        return false;
    if ( !Arrays.equals( chessboard, ((Chess) obj).chessboard ) )
        return false;
    return true;
}
```

Fließkommazahlen im Hashcode

Abhängig von den Datentypen sehen die Berechnungen immer etwas unterschiedlich aus. Während Ganzzahlen direkt in einen Ganzzahlausdruck für den Hashcode eingebracht werden können, sind im Fall von `double` die statischen Konvertierungsmethoden `Double.doubleToLongBits()` beziehungsweise `Float.floatToIntBits()` im Einsatz.

Die Datentypen `double` und `float` haben eine weitere Spezialität, da `NaN` und das Vorzeichen der `0` zu beachten sind, wie Kapitel 18, »Bits und Bytes und Mathematisches«, näher ausführt. Fazit: Sind $x = +0.0$ und $y = -0.0$, gilt $x == y$, aber `Double.doubleToLongBits(x) != Double.doubleToLongBits(y)`. Sind $x = y = Double.NaN$, gilt $x != y$, aber `Double.doubleToLongBits(x) == Double.doubleToLongBits(y)`. Wollen wir die beiden Nullen *nicht* unterschiedlich behandeln, sondern als gleich werten, ist Folgendes ein übliches Idiom:

```
x == 0.0 ? 0L : Double.doubleToLongBits( x )
```

`Double.doubleToLongBits(0.0)` liefert die Rückgabe `0`, aber der Aufruf `Double.doubleToLongBits(-0.0)` gibt `-9223372036854775808` zurück.

Equals, die Null und das Hashen

Inhaltlich gleiche Objekte (gemäß der Methode `equals()`) müssen denselben Wert bekommen.

Die beiden Methoden `hashCode()` und `equals()` hängen miteinander zusammen, sodass in der Regel bei der Implementierung einer Methode auch eine Implementierung der anderen notwendig wird. Denn es gilt, dass bei Gleichheit natürlich auch die Hash-Werte übereinstimmen müssen. Formal gesehen heißt das:

$$x.equals(y) \Rightarrow x.hashCode() == y.hashCode()$$

So berechnet sich der Hashcode bei Point-Objekten aus den Koordinaten. Zwei Punkt-Objekte, die inhaltlich gleich sind, haben die gleichen Koordinaten und damit auch den gleichen Hashcode. Wenn Objekte den gleichen Hashcode aufweisen, aber nicht gleich sind, handelt es sich um eine Kollision und den Fall, dass in der Gleichung nicht die Äquivalenz gilt.

8.3.6 System.identityHashCode() und das Problem der nicht-eindeutigen Objektverweise *

Die Gleichheit von Objekten wird mit der Methode `equals()` neu definiert. Wenn `equals()` neu implementiert wird, dann gilt das in der Regel auch für die Methode `hash-`

`Code()`, die ebenfalls überschrieben werden soll. So wird `hashCode()` bei unterschiedlichen Objektzuständen unterschiedliche Werte zurückgeben, und gleiche Objektinhalte müssen den gleichen Hashwert liefern.

Die Standardimplementierung von `Object` sieht nun so aus, dass auch bei Objekten, die gleiche Werte annehmen, unterschiedliche Hashwerte herauskommen – das ist auch der Grund dafür, warum wir `hashCode()` überschreiben sollten. Doch was liefert denn `hashCode()` von `Object` eigentlich? Es sieht so aus, als ob dies eine Objekt-ID wäre, die das Objekt eindeutig kennzeichnet. Die Ur-ID geht verloren, wenn `hashCode()` neu implementiert wird. Doch interessiert der ursprüngliche `hashCode()`-Wert, so bietet sich `System.identityHashCode()` an.



Hinweis

Obwohl die Hashwerte zu zwei Objekten gleich sind, liefert `identityHashCode()` in der Regel unterschiedliche Werte:

```
Point p = new Point( 0, 0 );
Point q = new Point( 0, 0 );
System.out.println( System.identityHashCode(p) ); // z. B. 16032330
System.out.println( System.identityHashCode(q) ); // z. B. 13288040
System.out.println( p.hashCode() );           // 0
System.out.println( q.hashCode() );           // 0
```

Wenn `hashCode()` nicht überschrieben wird, dann stimmt der Hashwert mit dem `identityHashCode()` überein.



zB Beispiel

Die Klasse `StringBuffer` überschreibt `hashCode()` nicht, sodass `identityHashCode()` gleich dem Hashwert ist:

```
StringBuffer sb1 = new StringBuffer();
StringBuffer sb2 = new StringBuffer();
System.out.println( System.identityHashCode(sb1) +
    " " + sb1.hashCode() ); // z. B. 7439041 7439041
System.out.println( System.identityHashCode(sb2) +
    " " + sb2.hashCode() ); // z. B. 4152583 4152583
```

Diese statische Methode `identityHashCode()` liefert den Original-Identifizierer der Objekte. Auf den ersten Blick sieht sie nach einer eindeutigen ID aus, das stimmt aber nicht immer. Es kann durchaus zwei unterschiedliche Objekte im Speicher geben, für die `System.identityHashCode()` gleich ist. Wir werden gleich ein Beispiel sehen.

Abilden von Objektverweisen in Datenbanken oder Dateien

Stellen wir uns vor, wir hätten eine Objekthierarchie im Speicher, die zum Beispiel jeder Socke einen Besitzer zuspricht. Wenn wir im Speicher Assoziationen abbilden, dann sollen diese Verweise auch noch nach dem Tod des Programms überleben. Eine Lösung ist die Serialisierung, eine andere eine Objekt-Datenbank oder aber auch eine XML-Datei. Doch überlegen wir selbst, wo bei der Abbildung auf eine Datenbank oder eine Datei das Problem besteht. Zunächst stehen ganz unterschiedliche Objekte mit ihren Eigenschaften im Speicher. Das Speichern der Zustände ist kein Problem, denn nur die Attribute müssen abgespeichert werden. Doch wenn ein Objekt auf ein anderes verweist, muss dieser Verweis gesichert werden. Aber in Java ist ein Verweis durch eine Referenz gegeben, und was sollte es da zu speichern geben? Eine Lösung für das Problem ist, jedem Objekt im Speicher einen Zähler zu geben und beim Speichern etwa zu sagen: »Der Besitzer 2 kennt Socke 5«.

Der Identifizierer für die Objekte muss eindeutig sein, und wir können überlegen, `System.identityHashCode()` zu nutzen. In der Implementierung der virtuellen Maschine von Oracle geht in den Wert von `identityHashCode()` die Information über den wahren Ort des Objekts im Speicher ein. Bei einer 64-Bit-Implementierung würden auch 32 Bit abgeschnitten, und die Eindeutigkeit wäre somit automatisch nicht mehr gewährleistet. Ein weiteres Problem besteht darin, dass zwar die Implementierung von Oracles `identityHashCode()` auf die eindeutige Objektspeicheradresse abbildet, aber dass das nicht jeder Hersteller so machen muss. Damit ist `identityHashCode()` nicht überall gesichert unterschiedlich. Zudem ist es prinzipiell denkbar, dass die Speicherverwaltung die Objekte verschiebt. Was sollte `identityHashCode()` dann machen? Wenn die neue Speicheradresse dahinter steckt, würde sich der Hashcode ändern, und das darf nicht sein. Es käme ebenfalls zu einem Problem, wenn mehr als `Integer.MAX_INTEGER` viele Objekte im Speicher stünden. (Doch wenn wir uns die große Zahl $2^{32} = 4.294.967.296$ vor Augen halten, dann es ist unwahrscheinlich, dass sich mehr als 4 Milliarden Objekte im Speicher tummeln. Zudem bräuchten wir 4 Gigabyte Speicher, wenn jedes Objekt auch nur 1 Byte kosten würde.)

Es ist gar nicht so schwierig, zwei unterschiedliche Objekte mit gleichen `identityHashCode()`-Resultat zu bekommen. Wir erzeugen ein paar `String`-Objekte und testen, jedes mit jedem, ob `identityHashCode()` den gleichen Wert ergibt:

Listing 8.22: com/tutego/insel/object/hashcode/IdentityHashCode.java, main()

```
String[] strings = new String[5000];

for ( int i = 0; i < strings.length; i++ )
    strings[i] = Integer.toString( i );

int cnt = 0;

for ( int i = 0; i < strings.length; i++ )
{
    for ( int j = i + 1; j < strings.length; j++ )
    {
        int id1 = System.identityHashCode( strings[i] );
        int id2 = System.identityHashCode( strings[j] );

        if ( id1 == id2 )
        {
            out.println( "Zwei Objekte mit identityHashCode() = " + id1 );
            out.println( " Objekt 1: \"" + strings[i] + "\"" );
            out.println( " Objekt 2: \"" + strings[j] + "\"" );
            out.println( " Object1.hashCode(): " + strings[i].hashCode() );
            out.println( " Object2.hashCode(): " + strings[j].hashCode() );
            out.println( " Object1.equals(Object2): " + strings[i].equals( strings[j] ) );

            cnt++;
        }
    }
}

System.out.println( cnt + " Objekte mit gleichem identityHashCode() gefunden." );
```

Ein Durchlauf bringt schnell Ergebnisse wie:

```
Zwei Objekte mit identityHashCode() = 9578500
Objekt 1: "541"
Objekt 2: "2066"
Object1.hashCode(): 52594
Object2.hashCode(): 1537406
Object1.equals(Object2): false
Zwei Objekte mit identityHashCode() = 14850080
Objekt 1: "2085"
Objekt 2: "2365"
Object1.hashCode(): 1537467
Object2.hashCode(): 1540288
Object1.equals(Object2): false
2 Objekte mit gleichem identityHashCode() gefunden.
```

Das Ergebnis ist also, dass `identityHashCode()` nicht sicher bei der Vergabe von Identifizierern ist. Um wirklich allen Problemen aus dem Weg zu gehen, ist ein Zählerobjekt oder eine ID über zum Beispiel die Klasse `java.util.UUID` nötig.

8.3.7 Aufräumen mit `finalize()` *

Einen *Destruktor*, der wie in C++ am Ende eines Gültigkeitsbereichs einer Variable aufgerufen wird, gibt es in Java nicht. Wohl ist es möglich, eine Methode `finalize()` für Aufräumarbeiten zu überschreiben, die *Finalizer* genannt wird (ein Finalizer hat nichts mit dem `finally`-Block einer Exception-Behandlung zu tun). Der Garbage-Collector ruft die Methode immer dann auf, wenn er ein Objekt entfernen möchte. Es kann allerdings sein, dass `finalize()` überhaupt nicht aufgerufen wird, und zwar dann, wenn die virtuelle Maschine Fantastillionen Megabyte an Speicher hat und dann beendet wird – in dem Fall gibt sie den Heap-Speicher als Ganzes dem Betriebssystem zurück. Ohne Garbage-Collector (GC) als Grabträger gibt es auch kein `finalize()`! Und wann der Garbage-Collector in Aktion tritt, ist auch nicht vorhersehbar, sodass im Gegensatz zu C++ in Java keine Aussage über den Zeitpunkt möglich ist, zu dem das Laufzeitsystem `finalize()` aufruft. Es ist von der Implementierung des GC abhängig. Üblicherweise werden aber Objekte mit `finalize()` von einem extra GC behandelt, und der arbeitet langsamer als der normale GC.

```
class java.lang.Object
```

- `protected void finalize() throws Throwable`

Wird vom GC aufgerufen, wenn es auf dieses Objekt keinen Verweis mehr gibt. Die Methode ist geschützt, weil sie von uns nicht aufgerufen wird. Auch wenn wir die Methode überschreiben, sollten wir die Sichtbarkeit nicht erhöhen, also `public` setzen.



Hinweis

Klassen sollten `finalize()` überschreiben, um wichtige Ressourcen zur Not freizugeben, etwa File-Handles via `close()` oder Grafik-Kontexte des Betriebssystems, wenn der Entwickler das vergessen hat. Alle diese Freigaben müssten eigentlich vom Entwickler angestoßen werden, und `finalize()` ist nur ein Helfer, derrettend eingreifen kann. Da der GC `finalize()` nur dann aufruft, wenn er tote Objekte freigeben möchte, dürfen wir uns nicht auf die Ausführung verlassen. Gehen zum Beispiel die File-Handles aus, wird der GC nicht aktiv; es erfolgen keine `finalize()`-Aufrufe, und nicht mehr erreichbare, aber noch nicht weggeräumte Objekte belegen weiter die knappen File-Handles.

Einmal Finalizer, vielleicht mehrmals der GC

Objekte von Klassen, die eine `finalize()`-Methode besitzen, kann Oracles JVM nicht so schnell erzeugen und entfernen wie Klassen ohne `finalize()`. Das liegt auch daran, dass der GC vielleicht mehrmals laufen muss, um das Objekt zu löschen. Es gilt zwar, dass der GC aus dem Grund `finalize()` aufruft, weil das Objekt nicht mehr benötigt wird, es kann aber sein, dass aus der `finalize()`-Methode die `this`-Referenz nach außen gegeben wurde, sodass das Objekt wegen einer bestehenden Referenz nicht gelöscht werden kann. Das Objekt wird zwar irgendwann entfernt, aber der Finalizer läuft nur einmal und nicht immer pro GC-Versuch. Einige Hintergründe erfährt der Leser unter <http://www.iecc.com/gclist/GC-lang.html#Finalization>.

Löst eine Anweisung in `finalize()` eine Ausnahme aus, so wird diese ignoriert. Das bedeutet aber, dass die Finalisierung des Objekts stehen bleibt. Den GC beeinflusst das in seiner Arbeit aber nicht.

`super.finalize()`

Überschreiben wir in einer Unterklasse `finalize()`, dann müssen wir auch gewährleisten, dass die Methode `finalize()` der Oberklasse aufgerufen wird. So besitzt zum Beispiel die Klasse `Font` ein `finalize()`, das durch eine eigene Implementierung nicht ver-

schwinden darf. Wir müssen daher in unserer Implementierung `super.finalize()` aufrufen (es wäre gut, wenn der Compiler das wie beim Konstruktorauftrag immer automatisch machen würde). Leere `finalize()`-Methoden ergeben im Allgemeinen keinen Sinn, es sei denn, das `finalize()` der Oberklasse soll explizit übergangen werden:

Listing 8.23: com/tutego/insel/object/finalize/SuperFont.java, `finalize()`

```
@Override protected void finalize() throws Throwable
{
    try {
        // ...
    }
    finally {
        super.finalize();
    }
}
```

Der Block vom `finally` wird immer ausgeführt, auch wenn es im oberen Teil eine Ausnahme gab.

Die Methode von Hand aufzurufen, ist ebenfalls keine gute Idee, denn das kann zu Problemen führen, wenn der GC-Thread die Methode auch gerade aufruft. Um das Aufrufen von außen einzuschränken, sollte die Sichtbarkeit von `protected` bleiben und nicht erhöht werden.

Hinweis

Da beim Programmende vielleicht nicht alle `finalize()`-Methoden abgearbeitet wurden, haben die Entwickler schon früh einen Methodenaufruf `System.runFinalizersOnExit(true)`; vorgesehen. Mittlerweile ist die Methode veraltet und sollte auf keinen Fall aufgerufen werden. Die API-Dokumentation erklärt:

»It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.«

Dazu auch Joshua Bloch, Autor des ausgezeichneten Buchs »Effective Java Programming Language Guide«:

»Never call `System.runFinalizersOnExit` or `Runtime.runFinalizersOnExit` for any reason: they are among the most dangerous methods in the Java libraries.«

8.3.8 Synchronisation *

Threads können miteinander kommunizieren und dabei Daten teilen. Sie können außerdem auf das Eintreten bestimmter Bedingungen warten, zum Beispiel auf neue Eingabedaten. Die Klasse `Object` deklariert insgesamt fünf Versionen der Methoden `wait()`, `notify()` und `notifyAll()` zur Beendigungssynchronisation von Threads.

8.4 Die Utility-Klasse `java.util.Objects`

In Java 7 ist die Klasse `Objects` hinzugekommen, die einige statische Utility-Funktionen bereithält. Sie führen in erster Linie `null`-Tests durch, um eine `NullPointerException` beim Aufruf von Objektmethoden zu vermeiden.

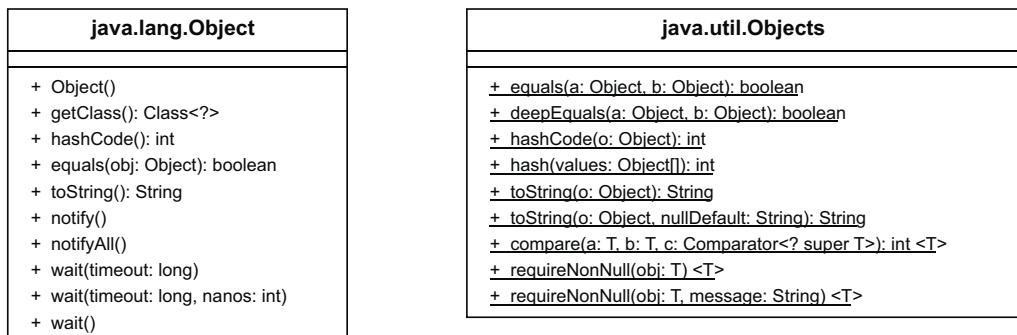


Abbildung 8.11: UML-Diagramm von Objects und Object

Eingebaute null-Tests für `equals()/hashCode()`

Ist zum Beispiel eine Objektvariable `name` einer Person `null`, so kann nicht einfach `name.hashCode()` aufgerufen werden, ohne dass eine `NullPointerException` folgt. Drei Methoden von `Objects` führen `null`-Tests durch, bevor sie an die `Object`-Methode `equals()`/ `hashCode()`/`toString()` weiterleiten. Eine zusätzliche Hilfsmethode arbeitet mit Comparatoren.

```
class java.util.Objects
■ static boolean equals(Object a, Object b)
Liefert true, wenn beide Argumente entweder null sind oder a.equals(b) ebenfalls
true ergibt, andernfalls liefert es false. Dass Objects.equals(null, null) die Rückgabe
true ergibt, ist sinnvoll, und so erspart die Methode einige händische Tests.
```

- static int hashCode(Object o)
Liefert 0, wenn o gleich null ist, sonst o.hashCode().
- static int hash(Object... values)
Ruft hashCode() auf jedem Objekt der Sammlung values auf und verbindet es zu einem neuen Hashcode. Die Implementierung ist einfach ein return Arrays.hashCode(values);.
- static <T> int compare(T a, T b, Comparator<? super T> c)
Liefert 0, wenn a und b beide entweder null sind, oder der Comparator die Objekte a und b für gleich erklärt. Sind a und b beide ungleich null, so ist die Rückgabe c.compare(a, b). Ist nur a oder b gleich null, so hängt das Ergebnis vom Comparator und der Reihenfolge der Parameter ab.

Beispiel

zB

Erinnern wir uns an die Methode hashCode() vom Spieler, bei der der Spielername in den Hashcode eingehen soll:

Listing 8.24: com/tutego/insel/object/hashcode/Player.java, hashCode() Ausschnitt

```
result = 31 * result + ((name == null) ? 0 : name.hashCode());
```

Mit Objects.hashCode() kann der null-Test entfallen, da er in schon in der statischen Methode vorgenommen wird:

```
result = 31 * result + Objects.hashCode( name.hashCode() );
```

Objects.toString()

Eine weitere statische Methode ist Objects.toString(). Sie ist aus Symmetriegründen in der Klasse, da `toString()` zu den Standard-Methoden der Klasse `Object` zählt. Genutzt werden muss die Methode nicht, da es mit `String.valueOf()` schon eine entsprechende Methode gibt.

```
class java.util.Objects
```

- static String toString(Object o)
Liefert den String »null«, wenn das Argument null ist, sonst o.toString().



Hinweis

Die Methode `String.valueOf()` ist überladen und ist somit insbesondere für primitive Argumente viel besser geeignet als `Objects.toString(Object)`, bei der immer erst Wrapper-Objekte aufgebaut werden müssen. Zwar sehen `String.valueOf(3.14)` und `Objects.toString(3.14)` gleich aus, aber im zweiten Fall kommt ein Wrapper-Double-Objekt mit ins Spiel.

null-Prüfungen mit eingebauter Ausnahmebehandlung

Zu den drei statischen Methoden kommen zwei hinzu, die null-Prüfungen übernehmen und im Fehlerfall eine Ausnahme auslösen. Das ist praktisch bei Konstruktoren oder Settern, die Werte initialisieren sollen, aber verhindern möchten, dass null durchgeleitet wird.

zB

Beispiel

Die Methode `setName()` soll kein name-Argument gleich null erlauben:

```
public void setName( String name )
{
    this.name = Objects.requireNonNull( name );
}
```

Alternativ ist eine Fehlermeldung möglich:

```
public void setName( String name )
{
    this.name = Objects.requireNonNull( name, "Name darf nicht null sein!" );
}
```

class java.util.Objects

- static <T> T requireNonNull(T obj)

Löst eine `NullPointerException` aus, wenn obj gleich null ist. Sonst liefert sie obj als Rückgabe. Die Deklaration ist generisch und so zu verstehen, dass der Parametertyp gleich dem Rückgabetypr ist.

- static <T> T requireNonNull(T obj, String message)

Wie `requireNonNull(obj)`, nur dass die Meldung der `NullPointerException` bestimmt wird.

8.5 Die Spezial-Oberklasse Enum

Jedes Aufzählungsobjekt erbt von der Spezialklasse `Enum`. Nehmen wir erneut die Woche ntage:

Listing 8.25: com/tutego/insel/enumeration/Weekday.java, Weekday

```
public enum Weekday
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Der Compiler übersetzt dies in eine Klasse, die etwa so aussieht:

```
class Weekday extends Enum
{
    public static final Weekday MONDAY = new Weekday( "MONDAY", 0 );
    public static final Weekday TUESDAY = new Weekday( "TUESDAY ", 1 );
    // weitere Konstanten ...

    private Weekday( String s, int i )
    {
        super( s, i );
    }

    // weitere Methoden ...
}
```

8.5.1 Methoden auf Enum-Objekten

Jedes `Enum`-Objekt besitzt automatisch einige Standardmethoden, die von der Oberklasse `java.lang.Enum` kommen. Das sind zum einen überschriebene Methoden aus `java.lang.Object`, einige neue Objektmethoden und einige statische Methoden.

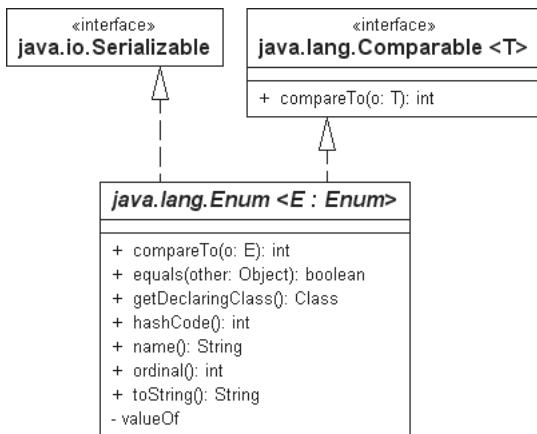


Abbildung 8.12: Typbeziehung von Enum

String-Repräsentation

Jedes Enum-Objekt liefert über die Methode `name()` den Namen der Konstante. Dazu gesellt sich die bekannte `toString()`-Methode, die standardmäßig `name()` aufruft, aber überschrieben werden kann. Die Methode `name()` lässt sich nicht überschreiben.

Eine vom Compiler generierte Enum-Klasse bietet eine statische `valueOf(String)`-Methode, die das Enum-Objekt liefert, das zur `name()`-Repräsentation passt. Wird bei `valueOf()` ein String übergeben, zu dem es kein Enum gibt, folgt eine `IllegalArgumentException`. Dazu kommt eine weitere statische Methode, die jedoch selbst schon in der Klasse `Enum` deklariert wird (die Basisklasse der vom Compiler erzeugten Enum-Klassen): `Enum.valueOf(Class<T> enumType, String s)`.

zB Beispiel

Die Konvertierung in den String und vom String in das entsprechende Enum-Objekt:

```

System.out.println( Weekday.MONDAY.toString() );           // MONDAY
System.out.println( Weekday.MONDAY.name() );               // MONDAY
System.out.println( Weekday.valueOf( "MONDAY" ).name() ); // MONDAY
System.out.println( Enum.valueOf( Weekday.class, "MONDAY" ).name() ); // MONDAY
  
```

Der Unterschied zu den `valueOf()`-Methoden ist wichtig: Während es `Enum.valueOf(Class, String)` nur einmal gibt, existieren statische `valueOf(String)`-Methoden einmal in jeder vom Compiler generierten Aufzählungsklasse. Da die Methode also compilergeneriert ist, taucht sie in der folgenden Aufzählung nicht auf.

```
abstract class java.lang.Enum<E extends Enum<E>>
    implements Comparable<E>, Serializable
```

- `final String name()`
Liefert den Namen der Konstanten. Da die Methode – wie viele andere der Klasse – `final` ist, lässt sich der Name nicht ändern.
- `String toString()`
Liefert den Namen der Konstanten. Die Methode ruft standardisiert `name()` auf, weil sie aber nicht `final` ist, kann sie überschrieben werden.
- `static <T extends Enum<T>> T valueOf(Class<T> enumType, String s)`
Ermöglicht das Suchen von `Enum`-Objekten zu einem Konstantennamen und einer `Enum`-Klasse. Sie liefert das `Enum`-Objekt für die gegebene Zeichenfolge oder löst eine `IllegalArgumentException` aus, wenn dem String kein `Enum`-Objekt zuzuordnen ist.

Alle Konstanten der Klasse aufzählen

Eine praktische statische Methode ist `values()`. Sie liefert ein Feld von `Enum`-Objekten. Nützlich ist das für das erweiterte `for`, das alle Konstanten aufzählen soll. Eine Alternative mit dem gleichen Ergebnis ist die `Class`-Methode `getEnumConstants()`:

Listing 8.26: com/tutego/insel/enumeration/WeekdayDemo.java, Ausschnitt main()

```
for ( Weekday day : Weekday.values() ) // oder Weekday.class.getEnumConstants()
    System.out.println( "Name=" + day.name() );
```

Liefert Zeilen mit Name=MONDAY, ...

Ordinalzahl

Von der Oberklasse `Enum` erbt jede Aufzählung einen geschützten parametrisierten Konstruktor, der den Namen der Konstanten sowie einen assoziierten Zähler erwartet. So wird aus jedem Element der Aufzählung ein Objekt vom Basistyp `Enum`, das einen Namen und eine ID, die sogenannte *Ordinalzahl*, speichert. Natürlich kann es auch nach seinem Namen und Zähler gefragt werden.

Beispiel

zB

Eine Methode, die die Ordinalzahl eines Elements der Aufzählung liefert oder `-1`, wenn die Konstante nicht existiert:

zB Beispiel (Forts.)

Listing 8.27: com/tutego/insel/enumeration/WeekdayDemo.java, getOrdinal()

```
static int getOrdinal( String name )
{
    try {
        return Weekday.valueOf( name ).ordinal();
    }
    catch ( IllegalArgumentException e ) {
        return -1;
    }
}
```

Damit liefert unser getOrdinal("MONDAY") == 0 und getOrdinal("WOCHENTAG") == -1.

Die Ordinalzahl gibt die Position in der Deklaration an und ist auch Ordnungskriterium der `compareTo()`-Methode. Die Ordinalzahl lässt sich nicht ändern und repräsentiert immer die Reihenfolge der deklarierten Konstanten.

zB Beispiel

Kommt Montag wirklich vor Freitag?

```
System.out.println( Weekday.MONDAY.compareTo( Weekday.FRIDAY ) ); // -4
System.out.println( Weekday.MONDAY.compareTo( Weekday.MONDAY ) ); // 0
System.out.println( Weekday.FRIDAY.compareTo( Weekday.MONDAY ) ); // 4
```

Negative Rückgaben bei `compareTo()` geben immer an, dass das erste Objekt »kleiner« als das zweite aus dem Argument ist.

```
abstract class java.lang.Enum<E extends Enum<E>>
implements Comparable<E>, Serializable
```

- `final int ordinal()`

Liefert die zur Konstante gehörige ID. Im Allgemeinen ist diese Ordinalzahl nicht wichtig, aber besondere Datenstrukturen wie `EnumSet` oder `EnumMap` nutzen diese eindeutige ID. Die Reihenfolge der Zahlen ist durch die Reihenfolge der Angabe gegeben.

- `public final boolean equals(Object other)`

Die Oberklasse `Enum` überschreibt `equals()` mit der Logik wie in `Object` – also den Vergleich der Referenzen –, um sie als `final` zu markieren.

- `protected final Object clone() throws CloneNotSupportedException`

Die Methode `clone()` ist final protected und kann also weder überschrieben noch von außen aufgerufen werden. So kann es keine Kopien der `Enum`-Objekte geben, die die Identität gefährden könnten. Grundsätzlich ist es aber erlaubt, wenn eigene Implementierungen von `clone()` die `this`-Referenz liefern.

- `final int compareTo(E o)`

Da die `Enum`-Klasse die Schnittstelle `Comparable` implementiert, gibt es auch die Methode `compareTo()`. Sie vergleicht anhand der Ordinalzahlen. Vergleiche sind nur innerhalb eines `Enum`-Typs erlaubt.

- `final Class<E> getDeclaringClass()`

Liefert das `Class`-Objekt zu einem konkreten `Enum`.

Hinweis



Die Methode `getDeclaringClass()` liefert auf der Aufzählungsklasse selbst `null` und nur auf den Elementen der Aufzählung einen sinnvollen Wert:

```
System.out.println( Weekday.class.getDeclaringClass() ); // null
System.out.println( Weekday.MONDAY.getDeclaringClass() );
// class com.tutego.weekday.Weekday
```

8.5.2 enum mit eigenen Konstruktoren und Methoden *

Da ein `enum`-Typ eine besondere Form der Klassendeklaration ist, kann er ebenso Attribute und Methoden deklarieren. Geben wir einer Aufzählung `Country` eine Methode, die den ISO-3166-2-Landescode des jeweiligen Aufzählungselements liefert:

Listing 8.28: com/tutego/insel/enumeration/Country.java, Country

```
public enum Country
{
    GERMANY, UK, CHINA;

    public String getISO3Country()
    {
        if ( this == GERMANY )
            return Locale.GERMANY.getISO3Country();
        else if ( this == UK )
            return Locale.UK.getISO3Country();
        else
            return Locale.CHINA.getISO3Country();
    }
}
```

```

        return Locale.UK.getISO3Country();
        return Locale.CHINA.getISO3Country();
    }
}

```

Die Methode `getISO3Country()` kann nun auf den `Enum`-Objekten aufgerufen werden:

```
System.out.println( Country.CHINA.getISO3Country() ); // CHN
```

Da `switch` auf `enum` erlaubt ist, können wir Folgendes schreiben:

Listing 8.29: `com/tutego/insel/enumeration/CountryEnumDemo.java`, Ausschnitt

```
Country c = Country.GERMANY;
```

```

switch ( c )
{
    case GERMANY:
        System.out.println( "Aha. Ein Krauti" );           // Aha. Ein Krauti
        System.out.println( c.getISO3Country() );          // DEU
        break;
    default:      System.out.println( "Anderes Land" );
}

```

enum für Singleton nutzen

Ein Singleton ist ein Objekt, das es in der Applikation nur einmal gibt.⁶ Javas `enum` ist dafür perfekt geeignet, denn die Aufzählungsobjekte gibt es in der Tat nur einmal, und die Bibliothek implementiert einige Tricks, um das Objekt auch möglichst nur einmal zu erzeugen, etwa dann wenn die Aufzählung serialisiert über die Leitung geht.

Ein Beispiel dazu. Ein `enum MainFrame` soll genau eine Konstante `INSTANCE` deklarieren. Da `enum`-Typen Attribute deklarieren können, soll unser `MainFrame` eine Objektvariable `JFrame` bekommen. Das bedeutet dann, dass mit dem Exemplar `INSTANCE` ein Swing-Fenster assoziiert ist. Da es nur ein Aufzählungselement in der `enum` gibt, kann auch nur ein `JFrame`-Exemplar gebildet werden:

⁶ Pro Klassenlader, um das etwas genauer auszudrücken.

Listing 8.30: com/tutego/insel/enumeration/MainFrame.java, MainFrame

```
public enum MainFrame
{
    INSTANCE;

    private JFrame f = new JFrame();

    public JFrame getFrame()
    {
        return f;
    }
}
```

Damit ist `INSTANCE` ein Exemplar vom Typ `MainFrame` und hat ein privates Attribut und eine öffentliche Zugriffsmethode. Da es nur eine Konstante gibt, gibt es auch nur ein Fenster. Eine Anwendung sieht etwa so aus:

Listing 8.31: com/tutego/insel/enumeration/MainFrameDemo.java, main()

```
MainFrame.INSTANCE.getFrame().setTitle( "Singleton" );
MainFrame.INSTANCE.getFrame().setBounds( 100, 100, 300, 400 );
MainFrame.INSTANCE.getFrame().setVisible( true );
```

Aus jedem Teil der Anwendung ist `MainFrame.INSTANCE` zugänglich und repräsentiert dieses eine Exemplar. Auch kann dieses Exemplar übergeben werden, weil es ein Objekt ist, wie jedes andere auch.

enum mit Konstruktoren

Neben dieser Variante wollen wir eine zweite Implementierung nutzen und nun Konstruktoren hinzuziehen, um das gleiche Problem auf andere Weise zu lösen:

Listing 8.32: com/tutego/insel/enumeration/Country.java, Country

```
public enum Country
{
    GERMANY( Locale.GERMANY ),
    UK( Locale.UK ),
    CHINA( Locale.CHINA );
```

```

private Locale country;

private Country( Locale country )
{
    this.country = country;
}

public String getISO3Country()
{
    return country.getISO3Country();
}
}

```

Bei der Deklaration der Konstanten wird in runden Klammern ein Argument für den Konstruktor übergeben. Der Konstruktor speichert das zugehörige `Locale`-Objekt in der internen Variablen `country`, auf die dann `getISO3Country()` Bezug nimmt.

enum mit überschriebenen Methoden

In dem `Enum`-Typ lassen sich nicht nur Methoden hinzufügen, sondern auch Methoden überschreiben. Beginnen wir mit einer lokalisierten und überladenen Methode `toString()`:

Listing 8.33: com/tutego/insel/enumeration/WeekdayInternational.java, WeekdayInternational

```

public enum WeekdayInternational
{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

    @Override
    public String toString()
    {
        return new SimpleDateFormat().getDateFormatSymbols()
            .getWeekdays()[ ordinal() + 1 ];
    }
}

```

```

public String toString( Locale l )
{
    return new SimpleDateFormat( "", l ).getDateFormatSymbols()
        .getWeekdays()[ ordinal() + 1 ];
}
}

```

Die erste Methode ist aus unserer Oberklasse `Object` überschrieben, die zweite als überladene Methode hinzugefügt. Ein Beispiel macht den Aufruf und die Funktionsweise klar:

8

Listing 8.34: com/tutego/insel/enumeration/WeekdayInternationalDemo.java, main()

```

System.out.println( WeekdayInternational.SATURDAY );                                // Samstag
System.out.println( WeekdayInternational.SATURDAY.toString() );                      // Samstag
System.out.println( WeekdayInternational.SATURDAY.toString(Locale.FRANCE) ); // samedi
System.out.println( WeekdayInternational.SATURDAY.toString(Locale.ITALY) );  // sabato

```

An dieser Stelle hören die Möglichkeiten der `enum`-Syntax aber noch nicht auf. Ähnlich wie die Syntax von inneren anonymen Klassen, die es erlauben, Methoden zu überschreiben, bieten Aufzählungstypen eine vergleichbare Syntax, um gezielt Methoden für eine spezielle Konstante zu überschreiben.

Nehmen wir an, in einem Spiel gibt es eine eigene Währung, den Ponro-Dollar. Nun soll dieser aber mit einer Referenzwährung, dem Euro, in Beziehung gesetzt werden; der Wechselkurs ist einfach 1:2:

Listing 8.35: com/tutego/insel/enumeration/GameCurrency.java, GameCurrency

```

public enum GameCurrency
{
    EURO() {
        @Override double convertTo( GameCurrency targetCurrency, double value )
        {
            return targetCurrency == EURO ? value : value / 2;
        }
    },
    PONRODOLLAR() {
        @Override double convertTo( GameCurrency targetCurrency, double value )
        {

```

```

        return targetCurrency == PONRODOLLAR ? value : value * 2;
    }
};

abstract double convertTo( GameCurrency targetCurrency, double value );
}

```

Der interessante Teil ist die Deklaration der abstrakten `convertTo()`-Methode und die Implementierung lokal bei den einzelnen Konstanten. (Natürlich müssen wir nicht jede Methode im `enum` abstrakt machen, sondern sie kann auch konkret sein. Dann muss nicht jedes `enum`-Element die abstrakte Methode implementieren.)

Mit einem statischen Import für die Aufzählung lässt sich die Nutzung und Funktionalität schnell zeigen:

Listing 8.36: com/tutego/insel/enumeration/GameCurrencyDemo.java, main()

```

System.out.println( EURO.convertTo( EURO, 12 ) );           // 12.0
System.out.println( EURO.convertTo( PONRODOLLAR, 12 ) );   // 6.0
System.out.println( PONRODOLLAR.convertTo( EURO, 12 ) );   // 24.0
System.out.println( PONRODOLLAR.convertTo( PONRODOLLAR, 12 ) ); // 12.0

```

enum kann Schnittstellen implementieren

Die API-Dokumentation von `Enum` zeigt an, dass die abstrakte Klasse zwei Schnittstellen implementiert: `Comparable` und `Serializable`. Jede in `enum` deklarierte Konstante ist Unterklasse von `Enums`, also immer vergleichbar und standardmäßig serialisierbar. Neben diesen Standardschnittstellen kann ein `enum` andere Schnittstellen implementieren. Das ist sehr nützlich, denn so schreibt es für alle Aufzählungselemente ein bestimmtes Verhalten vor – jedes Aufzählungselement bietet dann diese Operationen. Die Operationen der Schnittstelle können auf zwei Arten realisiert werden: Das `enum` selbst implementiert die Operationen der Schnittstelle im Rumpf, oder die einzelnen Aufzählungselemente realisieren die Implementierungen jeweils unterschiedlich. Oftmals dürfte es so sein, dass die Elemente unterschiedliche Implementierungen bereitstellen.

Unser nächstes kleines Beispiel für eine `enum DefaultIcons` implementiert die Schnittstelle `Icon` für grafische Symbole. Da die Symbole alle die gleichen Ausmaße haben, ist die `Icon`-Operation `getIconWidth()` und `getIconHeight()` immer gleich und wird nur einmal implementiert; die tatsächlichen `paintIcon()`-Implementierungen (die hier nur angedeutet werden) unterscheiden sich.

Listing 8.37: com/tutego/insel/enumeration/DefaultIcons.java, DefaultIcons

```

public enum DefaultIcons implements Icon
{
    WARNING {
        @Override public void paintIcon( Component c, Graphics g, int x, int y )
        {
            // g.drawXXX()
        }
    },
    ERROR {
        @Override public void paintIcon( Component c, Graphics g, int x, int y )
        {
            // g.drawXXX()
        }
    };

    @Override public int getIconWidth() { return 16; }

    @Override public int getIconHeight() { return 16; }
}

```

Der Zugriff DefaultIcons.ERROR gibt ein Objekt, das unter anderem vom Typ Icon ist und an allen Stellen übergeben werden kann, an denen ein Icon gewünscht ist.

8.6 Erweitertes for und Iterable

Bisher haben wir das erweiterte for für kleine Beispiele eingesetzt, in denen es darum ging, ein Feld (Array) von Elementen abzulaufen:

Listing 8.38: com/tutego/insel/iterable/SimpleIterable.java, main()

```

for ( String s : new String[]{ "Eclipse", "NetBeans", "IntelliJ" } )
    System.out.printf( "%s ist toll.%n", s );

```

8.6.1 Die Schnittstelle Iterable

Die erweiterte for-Schleife läuft nicht nur Felder ab, sondern alles, was vom Typ Iterable ist. Die Schnittstelle schreibt nur die Methode iterator() vor, die einen java.util.Iterator liefert, den das erweiterte for zum Durchlaufen verwendet.

```
interface java.lang.Iterable<T>
```

- Iterator<T> iterator()

Liefert einen Iterator, der über alle Elemente vom Typ T iteriert.

Insbesondere die Datenstrukturklassen implementieren diese Schnittstelle, sodass mit dem erweiterten `for` praktisch durch Ergebnismengen iteriert werden kann.

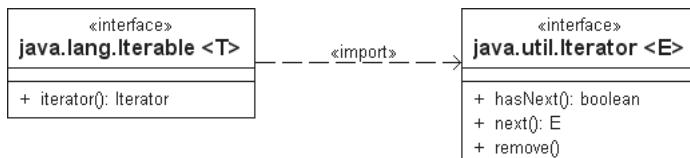


Abbildung 8.13: Klassendiagramm von Iterable und Iterator

8.6.2 Einen eigenen Iterable implementieren *

Möchten wir selbst rechts neben dem Doppelpunkt vom erweiterten `for` stehen, müssen wir ein Objekt angeben, dessen Klasse `Iterable` implementiert und somit eine `iterator()`-Methode besitzt. `iterator()` muss dann einen passenden `Iterator` zurückgeben. Der wiederum muss die Methoden `hasNext()` und `next()` implementieren, um das nächste Element in der Aufzählung anzugeben und das Ende anzuzeigen. Zwar schreibt der `Iterator` auch `remove()` vor, doch das wird leer implementiert.

Unser Beispiel soll einen praktischen `Iterable` implementieren, um über Wörter eines Satzes zu gehen. Als grundlegende Implementierung dient der `StringTokenizer`, der über `hasToken()` die nächsten Teilstufen und über `hasMoreTokens()` meldet, ob weitere Tokens ausgelesen werden können.

Beginnen wir mit dem ersten Teil, der Klasse `WordIterable`, die erst einmal `Iterable` implementieren muss, um auf der rechten Seite vom Punkt stehen zu können. Dann muss dieses Exemplar über `iterator()` einen `Iterator` zurückgeben, der über alle Wörter läuft. Dieser `Iterator` kann als eigene Klasse implementiert werden, doch wir implementieren die Klasse `WordIterable` so, dass sie `Iterable` und `Iterator` gleichzeitig verkörpert; daher ist nur ein Exemplar nötig:

Listing 8.39: com/tutego/insel/iterable/WordIterable.java

```
package com.tutego.insel.iterable;
```

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

```
class WordIterable implements Iterable<String>, Iterator<String>
{
    private StringTokenizer st;

    public WordIterable( String s )
    {
        st = new StringTokenizer( s );
    }

    // Method from interface Iterable

    @Override public Iterator<String> iterator()
    {
        return this;
    }

    // Methods from interface Iterator

    @Override public boolean hasNext()
    {
        return st.hasMoreTokens();
    }

    @Override public String next()
    {
        return st.nextToken();
    }

    @Override public void remove()
    {
        // No remove.
    }
}
```

Im Beispiel:

Listing 8.40: com/tutego/insel/iterable/WordIterableDemo.java, main()

```
String s = "Am Anfang war das Wort – am Ende die Phrase. (Stanislaw Jerzy Lec)";

for ( String word : new WordIterable(s) )
    System.out.println( word );
```

Die erweiterte for-Schleife baut der (Eclipse)-Compiler um zu:

```
String word;
WordIterable word iterable;
for ( Iterator iterator = (word iterable = new WordIterable(s)).iterator();
      iterator.hasNext(); )
{
    word = (String) iterator.next();
    System.out.println( word );
}
```

8.7 Zum Weiterlesen

Die API-Dokumentation der Standardklassen aus dem `java.lang`-Paket ist sehr hilfreich und sollte komplett studiert werden.



Kapitel 9

Generics<T>

»Irdisches Glück heißt: Das Unglück besucht uns nicht zu regelmäßig.«
– Karl Gutzkow (1811–1878)

9

9.1 Einführung in Java Generics

Generics zählen zu den komplexesten Sprachkonstrukten in Java. Wir wollen uns Generics in zwei Schritten nähern: von der Seite des Nutzers und von der Seite des API-Designers. Das Nutzen von generisch deklarierten Typen ist deutlich einfacher, sodass wir diese niedrig hängende Frucht zuerst pflücken wollen. Das Java-Buch für Fortgeschrittene dokumentiert sehr detailliert Generics aus der Sicht des API-Designers; die gepflückten Früchte werden dann veredelt.

9.1.1 Mensch versus Maschine: Typprüfung des Compilers und der Laufzeitumgebung

Eine wichtige Eigenschaft von Java ist, dass der Compiler die Typen prüft und so weiß, welche Eigenschaften vorhanden sind und welche nicht. Hier unterscheidet sich Java von dynamischen Programmiersprachen wie Python oder PHP, die erst spät eine Prüfung zur Laufzeit vornehmen.

In Java gibt es zwei Instanzen, die die Typen prüfen, und diese sind unterschiedlich schlau. Wir haben die JVM mit der absoluten Typ-Intelligenz, die unsere Anwendung ausführt und als letzte Instanz prüft, ob wir ein Objekt nicht einem falschen Typ zuweisen. Dann haben wir noch den Compiler, der zwar gut prüft, aber teilweise etwas zu gutgläubig ist und dem Entwickler folgt. Macht der Entwickler Fehler, kann dieser die JVM ins Verderben stürzen und zu einer `Exception` führen. Alles hat mit der expliziten Typanpassung zu tun.

Ein zunächst unkompliziertes Beispiel:

```
Object o = "String";
String s = (String) o;
```

Dem Compiler wird über den expliziten Typecast das Object o für ein String verkauft. Das ist in Ordnung, weil ja o tatsächlich ein String-Objekt referenziert. Problematisch wird es, wenn der Typ *nicht* auf String gebracht werden kann, wir dem Compiler aber eine Typanpassung anweisen:

```
Object o = Integer.valueOf( 42 );           // oder mit Autoboxing: Object o = 42;
String s = (String) o;
```

Der Compiler akzeptiert die Typanpassung, und es folgt kein Fehler zur Übersetzungszeit. Es ist jedoch klar, dass diese Anpassung von der JVM nicht durchgeführt werden kann – daher folgt zur Laufzeit eine ClassCastException, da eben ein Integer nicht auf String gebracht werden kann.

Bei Generics geht es nun darum, dem Compiler mehr Informationen über die Typen zu geben und ClassCastException-Fehler zu vermeiden.

9.1.2 Taschen

In unseren vorangehenden Beispielen drehte sich alles um Spieler und in einem Raum platzierte Spielobjekte. Stellen wir uns vor, der Spieler hat eine Tasche (engl. *pocket*), die etwas enthält. Da nicht bekannt ist, was genau er in der Tasche hat, müssen wir einen Basistyp nehmen, der alle möglichen Objekttypen repräsentiert. Das soll in unserem ersten Beispiel der allgemeinste Basistyp Object sein, sodass der Benutzer alles in seiner Tasche tragen kann:¹

Listing 9.1: com/tutego/insel/nongeneric/Pocket.java, Pocket

```
public class Pocket
{
    private Object value;
    public Pocket() {}
    public Pocket( Object value ) { this.value = value; }
    public void set( Object value ) { this.value = value; }
```

¹ Primitive Datentypen können über Wrapper-Objekte gespeichert werden, was seit Java 5 dank Autoboxing leicht möglich ist.

```

public Object get() { return value; }
public boolean isEmpty() { return value == null; }
public void empty() { value = null; }
}

```

Es gibt einen Standard- sowie einen parametrisierten Konstruktor. Mit `set()` lassen sich Objekte in die Tasche setzen und über die Zugriffsmethode `get()` wieder auslesen.

Geben wir einem Spieler eine rechte und eine linke Tasche:

Listing 9.2: com/tutego/insel/nongeneric/Player.java, Player

```

public class Player
{
    public String name;
    public Pocket rightPocket;
    public Pocket leftPocket;
}

```

9

Zusammen mit einem Spieler, der eine rechte und eine linke Tasche hat, ist ein Beispiel schnell geschrieben. Unser Spieler `michael` soll sich in beide Taschen Zahlen legen. Dann wollen wir sehen, in welcher Tasche er die größere Zahl versteckt hat.

Listing 9.3: com/tutego/insel/nongeneric/PlayerPocketDemo.java, main()

```

Player michael = new Player();
michael.name = "Omar Arnold";
Pocket pocket = new Pocket();
Long aBigNumber = 11111111111111L;
pocket.set( aBigNumber );                      // (1)
michael.leftPocket = pocket;
michael.rightPocket = new Pocket( 2222222222222222L );

System.out.println( michael.name + " hat in den Taschen " +
                    michael.leftPocket.get() + " und " + michael.rightPocket.get() );

Long val1 = (Long) michael.leftPocket.get();    // (2)
Long val2 = (Long) michael.rightPocket.get();

System.out.println( val1.compareTo( val2 ) > 0 ? "Links" : "Rechts" );

```

Das Beispiel hat keine besonderen Fallen, allerdings fallen zwei Sachen auf, die prinzipiell unschön sind. Die haben damit zu tun, dass die Klasse `Pocket` mit dem Typ `Object` zum Speichern der Tascheninhalte sehr allgemein deklariert wurde und alles aufnehmen kann:

- Beim Initialisieren wäre es gut, zu sagen, dass die Tasche nur einen bestimmten Typ (etwa `Long`) aufnehmen kann. Wäre eine solche Einschränkung möglich, dann lassen sich wie in Zeile (1) auch wirklich nur `Long`-Objekte in die Tasche setzen und nichts anderes, etwa `Integer`-Objekte.
- Beim Entnehmen (2) des Tascheninhalts mit `get()` müssen wir uns daran erinnern, was wir hineingelegt haben. Fordern Datenstrukturen besondere Typen, dann sollte dies auch dokumentiert sein. Doch wenn der Compiler wüsste, dass in der Tasche auf jeden Fall ein `Long` ist, dann könnte die Typanpassung wegfallen und der Programmcode wäre kürzer. Auch könnte uns der Compiler warnen, wenn wir versuchen würden, das `Long` als `Integer` aus der Tasche zu ziehen. Unser Wissen möchten wir gerne dem Compiler geben! Denn wenn in der Tasche ein `Long`-Objekt ist, wir es aber als `Integer` annehmen und eine explizite Typanpassung auf `Integer` setzen, meldet der Compiler zwar keinen Fehler, aber zur Laufzeit gibt es eine böse `ClassCastException`.

Um es auf den Punkt zu bringen: Der Compiler berücksichtigt im oberen Beispiel die Typsicherheit nicht ausreichend. Explizite Typanpassungen sind in der Regel unschön und sollten vermieden werden. Aber wie können wir die Taschen typischer machen?

Eine Lösung wäre, eine neue Klasse für jeden in der Tasche zu speichernden Typ zu deklarieren, also einmal eine `PocketLong` für den Datentyp `long`, dann vielleicht `PocketInteger` für `int`, `PocketString` für `String` usw. Das Problem bei diesem Ansatz ist, dass viel Code kopiert wird – fast identischer Code. Das ist keine vernünftige Lösung; wir können nicht für jeden Datentyp eine neue Klasse schreiben, und die Logik bleibt die gleiche. Wir wollen wenig schreiben, aber Typsicherheit beim Compilieren bekommen und nicht erst die Typsicherheit zur Laufzeit, wo uns vielleicht eine `ClassCastException` überrascht. Es wäre gut, wenn wir den Typ bei der Deklaration frei, allgemein, also »generisch« halten könnten, und sobald wir die Tasche benutzen, den Compiler dazu bringen könnten, auf diesen dann angegebenen Typ zu achten und die Korrektheit der Nutzung sicherzustellen.

Die Lösung für dieses Problem heißt *Generics*.² Diese Technik wurde in Java 5 eingeführt. Sie bietet Entwicklern ganz neue Möglichkeiten, um Datenstrukturen und Algorithmen zu programmieren, die von einem Datentyp unabhängig, also *generisch* sind.

9.1.3 Generische Typen deklarieren

Wollen wir `Pocket` in einen *generischen Typ* umbauen, so müssen wir an den Stellen, an denen `Object` vorkam, einen Typstellvertreter, einen sogenannten *formalen Typparameter* einsetzen, der durch eine *Typvariable* repräsentiert wird. Der Name der Typvariablen muss in der Klassendeklaration angegeben werden.

Die Syntax für den generischen Typ von `Pocket` ist folgende:

Listing 9.4: com/tutego/insel/generic/Pocket.java, Pocket

```
public class Pocket<T>
{
    private T value;
    public Pocket() {}
    public Pocket( T value ) { this.value = value; }
    public void set( T value ) { this.value = value; }
    public T get() { return value; }
    public boolean isEmpty() { return value != null; }
    public void empty() { value = null; }
}
```

Wir haben die Typvariable `T` definiert und verwenden diese jetzt anstelle von `Object` in der `Pocket`-Klasse.

Bei generischen Typen steht die Angabe der Typvariable nur einmal zu Beginn der Klassendeklaration in spitzen Klammern hinter dem Klassennamen. Der Typparameter kann nun fast³ überall dort genutzt werden, wo auch ein herkömmlicher Typ stand. In unserem Beispiel ersetzen wir direkt `Object` durch `T`, und fertig ist die generische Klasse.

Namenskonvention

Formale Typparameter sind in der Regel einzelne Großbuchstaben wie `T` (steht für Typ), `E` (Element), `K` (Key/Schlüssel), `V` (Value/Wert). Sie sind nur Platzhalter und keine wirklichen Typen. Möglich wäre etwa auch Folgendes, doch davon ist absolut abzuraten, da `Elf` viel zu sehr nach einem echten Klassentyp als nach einem formalen Typparameter aussieht:



2 In C++ werden diese Typen von Klassen *parametisierte Klassen* oder *Templates* (Schablonen) genannt.

3 `T t = new T();` ist zum Beispiel nicht möglich.

**Namenskonvention (Forts.)**

```
public class Pocket<Elf>
{
    private Elf value;
    public void set( Elf value ) { this.value = value; }
    public Elf get() { return value; }
}
```

Es dürfen nicht nur Elfen in die Klasse, sondern alle Typen.

Wofür Generics noch gut ist

Es gibt eine ganze Reihe von Beispielen, in denen Speicherstrukturen wie unsere Tasche nicht nur für einen Datentyp `Long` sinnvoll sind, sondern grundsätzlich für alle Typen, wobei aber die Implementierung (relativ) unabhängig vom Typ der Elemente ist. Das gilt zum Beispiel für einen Sortieralgorithmus, der mit der Ordnung der Elemente arbeitet. Wenn zwei Elemente größer oder kleiner sein können, muss ein Algorithmus lediglich diese Eigenschaft nutzen können. Es ist dabei egal, ob es Zahlen vom Typ `Long`, `Double` oder auch `Strings` oder Kunden sind – der Algorithmus selbst ist davon nicht betroffen. Der häufigste Einsatz von Generics sind Container, die typsicher gestaltet werden sollen.

**Geschichtsstunde**

Die Idee, Generics in Java einzuführen, ist schon älter und geht auf das Projekt *Pizza* beziehungsweise das Teilprojekt *GJ* (A Generic Java Language Extension) von Martin Odersky (der auch der Schöpfer der Programmiersprache *Scala* ist), Gilad Bracha, David Stoutamire und Philip Wadler zurück. *GJ* wurde dann die Basis des JSR 14, »Add Generic Types To The Java Programming Language«.

9.1.4 Generics nutzen

Um die neue `Pocket`-Klasse nutzen zu können, müssen wir sie zusammen mit einem Typparameter angeben; es entstehen hier zwei *parametisierte Typen*:

Listing 9.5: com/tutego/insel/generic/PocketPlayer.java, main() Teil 1

```
Pocket<Integer> intPocket = new Pocket<Integer>();
Pocket<String> stringPocket = new Pocket<String>();
```

Der konkrete Typ steht immer hinter dem Klassen-/Schnittstellennamen in spitzen Klammern.⁴ Die Tasche `intPocket` ist eine Instanz eines generischen Typs mit dem konkreten Typargument `Integer`. Diese Tasche kann jetzt offiziell nur `Integer`-Werte enthalten, und die Tasche `stringPocket` enthält nur Zeichenketten. Das prüft der Compiler auch, und wir benötigen keine Typanpassung mehr:

Listing 9.6: com/tutego/insel/generic/PocketPlayer.java, main() Teil 2

```
intPocket.set( 1 );
int x = intPocket.get();           // Keine Typanpassung mehr nötig
stringPocket.set( "Selbstzerstörungsauslösungsschalterhintergrundbeleuchtung" );
String s = stringPocket.get();
```

Der Entwickler macht so im Programmcode sehr deutlich, dass die Taschen einen `Integer` enthalten und nichts anderes. Da Programmcode häufiger gelesen als geschrieben wird, sollten Autoren immer so viele Informationen wie möglich über den Kontext in den Programmcode legen. Zwar leidet die Lesbarkeit etwas, da insbesondere beim Instanziieren der Typ sowohl rechts wie auch links angegeben werden muss und die Syntax bei geschachtelten Generics lang werden kann, doch wie wir später sehen werden, lässt sich das ab Java 7 abkürzen.

Das Schöne für die Typsicherheit ist, dass nun alle Eigenschaften mit dem angegebenen Typ geprüft werden. Wenn wir etwa aus `intPocket` mit `get()` auf das Element zugreifen, ist es vom Typ `Integer` (und durch Unboxing gleich `int`), und `set()` erlaubt auch nur ein `Integer`. Das macht den Programmcode robuster und durch den Wegfall der Typanpassungen kürzer und lesbarer.

Begriff	Beispiel
Generischer Typ (engl. <i>generic type</i>)	<code>Pocket<T></code>
Typvariable oder formaler Typparameter (engl. <i>formal type parameter</i>)	<code>T</code>
Parametrisierter Typ (engl. <i>parameterized type</i>)	<code>Pocket<Long></code>

Tabelle 9.1: Zusammenfassung der bisherigen Generics-Begriffe

⁴ Dass auch XML in spitzen Klammern daherkommt und XML als groß und aufgeblättert gilt, wollen wir nicht als Parallele zu Javas Generics sehen.

Begriff	Beispiel
Typparameter (engl. <i>actual type parameter</i>)	Long
Originaltyp (engl. <i>raw type</i>)	Pocket

Tabelle 9.1: Zusammenfassung der bisherigen Generics-Begriffe (Forts.)



Keine Primitiven

Typparameter können in Java Klassen, Schnittstellen, Aufzählungen und Arrays davon sein, aber keine primitiven Datentypen. Das schränkt die Möglichkeiten zwar ein, doch da es Autoboxing gibt, lässt sich damit leben. Und wenn `null` in der `Pocket<Integer>` liegt, führt ein Unboxing zur Laufzeit zur `NullPointerException`.

Geschachtelte Generics

Ist ein generischer Typ wie `Pocket<T>` gegeben, gibt es erst einmal keine Einschränkung für `T`. So beschränkt sich `T` nicht auf einfache Klassen- oder Schnittstellentypen, sondern kann auch wieder ein generischer Typ sein. Das ist logisch, denn jeder generische Typ ist ja ein eigenständiger Typ, der (fast) wie jeder andere Typ genutzt werden kann:

Listing 9.7: com/tutego/insel/generic/PocketPlayer.java, main() Teil 3

```
Pocket<Pocket<String>> pocketOfPockets = new Pocket<Pocket<String>>();
pocketOfPockets.set( new Pocket<String>() );
pocketOfPockets.get().set( "Inner Pocket<String>" );
System.out.println( pocketOfPockets.get().get() ); // Inner Pocket<String>
```

Hier enthält die Tasche eine Innentasche, die eine Zeichenkette `"Inner Pocket<String>"` speichert.

Bei Dingen wie diesen ist schnell offensichtlich, wie hilfreich Generics für den Compiler (und uns) sind. Ohne Generics sähen eben alle Taschen gleich aus.

Präzise mit Generics	Unpräzise ohne Generics
<code>Pocket<String> stringPocket;</code>	<code>Pocket stringPocket;</code>
<code>Pocket<Integer> intPocket;</code>	<code>Pocket intPocket;</code>
<code>Pocket<Pocket<String>> pocketOfPockets;</code>	<code>Pocket pocketOfPockets;</code>

Tabelle 9.2: Präzisierung durch Generics

Nur ein gut gewählter Name und eine präzise Dokumentation können bei nicht-generisch deklarierten Variablen helfen. Vor Java 5 haben sich Entwickler damit geholfen, mithilfe eines Blockkommentars Generics anzudeuten, etwa in `Pocket/*<String>*/ stringPocket.`

Keine Arrays von parametrisierten Typen



Die folgende Anweisung bereitet eine einzige Tasche mit einem Feld von Strings vor:

```
Pocket<String[]> pocketForArray = new Pocket<String[]>();
```

Aber lässt sich auch ein Array von mehreren Taschen, die jeweils Strings enthalten, deklarieren? Ja. Doch während die Deklaration noch möglich ist, ist die Initialisierung schlichtweg ungültig:

```
Pocket<String>[] arrayOfPocket;
arrayOfPocket = new Pocket<String>[2]; // ☹ Compilerfehler
```

Der Grund liegt in der Umsetzung in Bytecode verborgen, und die beste Lösung ist, die komfortablen Datenstrukturen aus dem `java.util`-Paket zu nutzen. Für Entwickler ist ein `List<Pocket<String>>` sowieso sexyer als ein `Pocket<String>[]`. Laufzeiteinbußen sind kaum zu erwarten. Da Arrays aber vom Compiler automatisch bei variablen Argumentlisten eingesetzt werden, gibt es ein Problem, wenn die Parametervariable eine Typvariable ist. Bei Signaturen wie `f(T... params)` hilft die Annotation `@SafeVarargs`, die Compiler-Meldung zu unterdrücken.

9

9.1.5 Diamonds are forever

Bei der Initialisierung einer Variablen, deren Typ generisch ist, fällt auf, dass der Typparameter zweimal angegeben werden muss. Bei geschachtelten Generics fällt die Mehrarbeit unangenehm auf. Nehmen wir eine Liste, die Maps enthält, wobei der Assoziativspeicher Datumswerte mit Strings verbindet:

```
List<Map<Date, String>> listOfMaps;
listOfMaps = new ArrayList<Map<Date, String>>();
```

Der Typparameter `Map<Date, String>` steht einmal auf der Seite der Variablen-deklaration und einmal beim `new`-Operator.

Seit Java 7 erlaubt der Compiler eine verkürzte Schreibweise, denn er kann aus dem Kontext den Typ ableiten; diese Eigenschaft – *Typ-Inferenz* (engl. *type inference*) genannt – wird uns noch einmal über den Weg laufen.

Der Diamantoperator

Verfügt der Compiler über alle Typinformationen, so können beim new-Operator die Typparameter entfallen, und es bleibt lediglich ein Pärchen spitzer Klammern:

zB Beispiel

Ab Java 7 wird aus

```
List<Map<Date, String>> listOfMaps = new ArrayList<Map<Date, String>>();
```

einfach:

```
List<Map<Date, String>> listOfMaps = new ArrayList<>();
```

Wegen des Aussehens der spitzen Klammern `<>` nennt sich der Typ, für den die spitzen Klammern stehen, auch *Diamanttyp* (engl. *diamond type*). Das Pärchen `<>` wird auch *Diamantoperator* (engl. *diamond operator*) genannt, und es ist ein Operator, weil er den Typ herausfindet, weshalb er auch *Diamant-Typ-Inferenz-Operator* genannt wird.



Randnotiz

Es ist ungewöhnlich, dass der Java-Compiler hier den Typ der linken Seite betrachtet – denn bei `long val = 10000000000;` macht er das auch nicht. Doch darüber müssen wir uns keine so großen Gedanken machen, denn dies ist nicht das einzige Problem in der Java-Grammatik ...

Einsatzgebiete des Diamanten

Der Diamant in unserem Beispiel ersetzt den gesamten Typparameter `Map<String, String>`. Es ist nicht möglich, ihn nur zum Teil bei geschachtelten Generics einzusetzen. So schlägt `new ArrayList<Map<>>()` fehl. Auch ist nur bei `new` der neue Diamant-Operator erlaubt, und es wäre falsch, ihn auch auf der linken Seite bei der Variablen Deklaration einzusetzen und ihn etwa auf der rechten Seite bei der Bildung des Exemplars zu nutzen. Eine Deklaration wie `List<> listOfMaps;` führt somit zum Compilerfehler, denn der Compiler würde nicht bei jeder folgenden Nutzung irgendwelche Typen ableiten können.

Da der Diamant bei `new` eingesetzt wird, kann er – bis auf einige Ausnahmen, die wir uns später anschauen – immer dort eingesetzt werden, wo Exemplare gebildet werden. Das nächste Nonsense-Beispiel zeigt vier Einsatzgebiete:

```

import java.util.*;

public class WereToUseTheDiamond
{
    public static List<String> foo( List<String> list )
    {
        return new ArrayList<>();
    }

    public static void main( String[] args )
    {
        List<String> list = new ArrayList<>();
        list = new ArrayList<>();
        foo( new ArrayList<>( list ) );
    }
}

```

Die Einsatzorte sind:

- bei Deklarationen und der Initialisierung von Attributen und lokalen Variablen
- bei der Initialisierung von Attributen, lokalen Variablen bzw. Parametervariablen
- als Argument bei Methoden-/Konstruktoraufrufen
- bei Methodenrückgaben

Ohne Frage ist der erste und zweite Fall der sinnvollste. Fast überall kann der Diamant die Schreibweise abkürzen. Besonders im ersten Fall spricht nichts Grundsätzliches gegen den Einsatz, bei den anderen drei Punkten muss berücksichtigt werden, ob nicht vielleicht die Lesbarkeit des Programmcodes leidet. Wenn zum Beispiel in mitten in einer Methode eine Datenstruktur mit `list = new ArrayList<>()` initialisiert wird, aber die Variablendeklaration nicht auf der gleichen Bildschirmseite liegt, ist mitunter für den Leser nicht sofort sichtbar, was denn genau für Typen in der Liste sind.⁵

⁵ Um den Diamanten zu testen, haben die Entwickler ein Tool geschrieben, das durch das JDK läuft und schaut, welche generisch genutzten news durch den Diamanten vereinfacht werden könnten. (Heraus kamen etwa 5000 Stellen.) Nicht jedes Team hat jede erlaubte Konvertierung hin zum Diamanten akzeptiert. So wollte das Team, das die Java-Security-Bibliotheken pflegt, weiterhin die explizite Schreibweise der Generics bei Zuweisungen beibehalten.

Diamant nicht möglich

Es gibt Situationen, in denen die Typableitung nicht so funktioniert wie erwartet. Oftmals hat das mit dem Einsatz des Diamanten bei Methodenaufrufen zu tun, sodass anzuraten ist – auch schon aus Gründen der Programmverständlichkeit –, bei Methodenaufrufen grundsätzlich auf Diamanten zu verzichten.

Ein Beispiel mit zwei für den Compiler unlösbaren Fällen:

```
class NoDiamondsForYou
{
    static void out( List<String> list ) { }

    public static void main( String[] args )
    {
        out( new ArrayList<>() ); // ☹ Compilerfehler
        List<String> list = new ArrayList<>().subList(0, 1); // ☹ Compilerfehler
    }
}
```

Die Typinferenz ist komplex,⁶ und glücklicherweise muss ein Entwickler sich nicht um die interne Arbeitsweise kümmern. Wenn der Diamant, wie im Beispiel, nicht möglich ist, löst eine explizite Typangabe das Problem.

9.1.6 Generische Schnittstellen

Eine Schnittstelle kann genauso als generischer Typ deklariert werden wie eine Klasse. Werfen wir einen Blick auf die Schnittstellen `java.lang.Comparable` und einen Ausschnitt von `java.util.Set` (Schnittstelle, die Operationen für Mengenoperationen vorschreibt, mehr dazu in Kapitel 13, »Einführung in Datenstrukturen und Algorithmen«), die beide seit Java 5 mit einer Typvariablen ausgestattet sind.

⁶ Für Java 7 standen zwei Algorithmen zur Typauswahl zur Auswahl: simpel und komplex. Der komplexe Ansatz bezieht neben den Typeninformationen, die eine Zuweisung liefert, noch den Argumenttyp mit ein. Zunächst verwendete das Team den einfachen Algorithmus, wechselte ihn jedoch später, da der komplexe Ansatz auf Algorithmen zurückgreift, die der Compiler auch an deren Stellen einsetzt. Ein paar mehr Details geben die Präsentation http://blogs.oracle.com/darcy/resource/JavaOne/J1_2010-ProjectCoin.pdf und Beiträge in der Mailingliste <http://mail.openjdk.java.net/pipermail/coin-dev/2009-November/002393.html>.

<pre>public interface Comparable<T> { public int compareTo(T o); }</pre>	<pre>public interface Set<E> extends Collection<E> { int size(); boolean isEmpty(); boolean contains(Object o); Iterator<E> iterator(); Object[] toArray(); <T> T[] toArray(T[] a); boolean add(E e); ... }</pre>
------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Tabelle 9.3: Generische Deklaration der Schnittstellen Comparable und Set

Wie bekannt, greifen die Methoden auf die Typvariablen T und E zurück. Bei `Set` ist weiterhin zu erkennen, dass sie selbst eine generisch deklarierte Schnittstelle erweitert.

Beim Einsatz von generischen Schnittstellen lassen sich die folgenden zwei Benutzungsmuster ableiten:

- Ein nicht-generischer Klassentyp löst Generics bei der Implementierung auf.
- Ein generischer Klassentyp implementiert eine generische Schnittstelle und gibt die Parametervariable weiter.

Nicht-generischer Klassentyp löst Generics bei der Implementierung auf

Im ersten Fall implementiert eine Klasse die generisch deklarierte Schnittstelle und gibt einen konkreten Typ an. Alle numerischen Wrapper-Klassen implementieren zum Beispiel `Comparable` und füllen den Typparameter genau mit dem Typ der Wrapper-Klasse:

```
public final class Integer extends Number implements Comparable<Integer>
{
    public int compareTo( Integer anotherInteger ) { ... }
    ...
}
```

Durch diese Nutzung wird für den Anwender die Klasse `Integer` Generics-frei.

Generischer Klassentyp implementiert generische Schnittstelle und gibt die Parametervariable weiter

Die Schnittstelle Set schreibt Operationen für Mengen vor. Eine Klasse, die Set implementiert, ist zum Beispiel HashSet. Der Kopf der Typdeklaration ist folgender:

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

Es ist abzulesen, dass Set eine Typvariable E deklariert, die HashSet nicht konkretisiert. Der Grund ist, dass die Datenstruktur Set vom Anwender als parametrisierter Typ verwendet wird und nicht aufgelöst werden soll.



Hinweis

In manchen Situationen wird auch Void als Typparameter eingesetzt. Wenn etwa interface I<T> { T foo(); } eine Typvariable T deklariert, ohne dass es bei der Implementierung von I etwas zurückzugeben gibt, dann kann der Typparameter Void sein:

```
class C implements I<Void> {
    @Override public Void foo() { return null; }
}
```

Allerdings sind void und Void unterschiedlich, denn bei Void muss es eine Rückgabe geben, was ein return null notwendig macht.

9.1.7 Generische Methoden/Konstruktoren und Typ-Inferenz

Die bisher genannten generischen Konstruktionen sahen im Kern wie folgt aus:

- class Klassenname<T> { ... }
- interface Schnittstellename<T> { ... }

Eine an der Klassen- oder Schnittstellendeklaration angegebene Typvariable kann in allen nicht-statischen Eigenschaften des Typs angesprochen werden.

zB

Beispiel

Folgendes führt zu einem Fehler:

```
class Pocket<T> {
    static void foo( T t ) { };      // ☹ Compilerfehler
}
```

Der Eclipse-Compiler meldet: »Cannot make a static reference to the non-static type T«.

Doch was machen wir, wenn

- statische Methoden eine eigene Typvariable nutzen wollen?
- unterschiedliche statische Methoden unterschiedliche Typvariablen nutzen möchten?

Eine Klasse kann auch ohne Generics deklariert werden, aber *generische Methoden* besitzen. Ganz allgemein kann jeder Konstruktor, jede Objektmethode und jede Klassenmethode einen oder mehrere formale Typparameter deklarieren. Sie stehen dann nicht mehr an der Klasse, sondern an der Methoden-/Konstruktordeklaration und sind »lokal« für die Methode beziehungsweise den Konstruktor. Das allgemeine Format ist:

Modifizierer <Typvariable(n)> Rückgabetyp Methodename(Parameter) throws-Klausel

Ganz zufällig das eine oder andere Argument

Interessant sind generische Methoden insbesondere für Utility-Klassen, die nur statische Methoden anbieten, aber selbst nicht als Objekt vorliegen. Das folgende Beispiel zeigt das anhand einer Methode `random()`:

Listing 9.8: com/tutego/insel/generic/GenericMethods.java, GenericMethods

```
public class GenericMethods
{
    public static <T> T random( T m, T n )
    {
        return Math.random() > 0.5 ? m : n;
    }

    public static void main( String[] args )
    {
        String s = random( "Analogkäse", "Gel-Schinken" );
    }
}
```

```

        System.out.println( s );
    }
}

```

Dabei deklariert `<T> T random(T m, T n)` eine generische Methode, wobei der Rückgabetyp und Parametertyp durch eine Typvariable `T` bestimmt wird. Die Angabe von `<T>` beim Klassennamen ist bei dieser Syntax entfallen und wurde auf die Deklaration der Methode verschoben.



Hinweis

Natürlich kann eine Klasse als generischer Typ und eine darin enthaltene Methode als generische Methode mit unterschiedlichem Typ deklariert werden. In diesem Fall sollten die Typvariablen unterschiedlich benannt sein, um den Leser nicht zu verwirren. So bezieht sich im Folgenden `T` bei `sit()` eben nicht auf die Parametervariable der Klasse Lupilu, sondern auf die der Methode:

```

class Lupilu<T> { <T> void sit( T val ); } // Verwirrend
class Lupilu<T> { <V> void sit( V val ); } // Besser

```

Der Compiler auf der Suche nach Gemeinsamkeiten

Den Typ (der wichtig für die Rückgabe ist) leitet der Compiler also automatisch aus dem Kontext, das heißt aus den Argumenten, ab. Diese Eigenschaft nennt sich *Typ-Inferenz* (engl. *type inference*). Das hat weitreichende Konsequenzen.

Bei der Deklaration `<T> T random(T m, T n)` sieht es vielleicht auf den ersten Blick so aus, als ob die Variablentypen `m` und `n` absolut gleich sein müssen. Das stimmt aber nicht, denn bei den Typen geht der Compiler in der Typhierarchie so weit nach oben, bis er einen gemeinsamen Typ findet.

Aufruf	Identifizierte Typen	Gemeinsame Basistypen
<code>random("Essen", 1)</code>	String, Integer	Object, Serializable, Comparable
<code>random(1L, 1D)</code>	Long, Double	Object, Number, Comparable
<code>random(new Point(), new StringBuilder())</code>	Point, StringBuilder	Object, Serializable, Cloneable

Tabelle 9.4: Gemeinsame Basistypen

Es fällt auf, aber überrascht nicht, dass `Object` immer in die Gruppe gehört.

Die Schnittmenge der Typen bildet im Fall von `random()` die gültigen Rückgabetypen. Erlaubt sind demnach für die Parametertypen `String` und `Integer`:

```
Object      s1 = random( "Essen", 1 );
Serializable s2 = random( "Essen", 1 );
Comparable   s3 = random( "Essen", 1 );
```

Knappe Fabrikmethoden

Der bei Java 7 eingeführte Diamant-Typ kürzt Variablen Deklarationen mit Initialisierung einer Referenzvariablen angenehm ab. Heißt es unter Java 6 noch zwingend

```
Pocket<String> p = new Pocket<String>();
```

und musste `<String>` zweimal angegeben werden, so erlaubt der Diamant (`»<>«`) ab Java 7 Folgendes:

```
Pocket<String> p = new Pocket<>();
```

Mit der Typ-Inferenz gibt es eine alternative Lösung auch für Java 5 und Java 6. Geben wir unserer Klasse `Pocket` eine Fabrikmethode

```
public static <T> Pocket<T> newInstance()
{
    return new Pocket<T>();
}
```

so ist folgende Alternative möglich:

```
Pocket<String> p = Pocket.newInstance();
```

Aus dem Ergebnistyp `Pocket<String>` leitet der Compiler den tatsächlichen Typparameter `String` für die Tasche ab. Und ist bei einem einfachen Typ wie `String` die Schreibersparnis noch gering, wird der Code bei verschachtelten Datenstrukturen kürzer. Soll die Tasche einen Assoziativspeicher aufnehmen, der eine Zeichenkette mit einer Liste von Zahlen assoziiert, so schreiben wir kompakt:

```
Pocket<Map<String, List<Integer>>> p = Pocket.newInstance();
```

Natürlich müssen entweder die Klassen selbst oder Utility-Klassen diese Fabrikmethoden anbieten, denn von selbst sind sie nicht vorhanden. Da unter Java 7 der Diamant aber den Programmcode bei den Initialisierungen verkürzt, ist es unwahrscheinlich, dass Bibliotheksautoren sie jetzt nachrüsten. Die populäre Google-Collections-Biblio-

theke (<http://code.google.com/p/guava-libraries/>) bietet Utility-Klassen wie Lists, Maps, Sets mit genau den statischen Fabrikfunktionen wie `newArrayList()`, `newLinkedList()`, `newTreeMap()`, `newHashSet()`, die von der Typ-Inferenz leben.

Generische Methoden mit explizitem Typparameter *

Es gibt Situationen, in denen der Compiler nicht aus dem Kontext über Typ-Inferenz den richtigen Typ ableiten kann. Zum Beispiel ist Folgendes nicht möglich:

```
boolean hasPocket = true;
Pocket<String> pocket = hasPocket ? Pocket.newInstance() : null;
```

Der Eclipse-Compiler meldet »Type mismatch: cannot convert from Pocket<Object> to Pocket<String>«.

Die Lösung: Wir müssen bei `Pocket.newInstance()` den Typparameter `String` explizit angeben:

```
Pocket<String> pocket = hasPocket ? Pocket.<String>.newInstance() : null;
```

Die Syntax ist etwas gewöhnungsbedürftig, doch in der Praxis ist die explizite Angabe selten nötig.

zB

Beispiel

Ist das Argument der statischen Methode `Arrays.asList()` ein Feld, dann ist der explizite Typparameter nötig, da der Compiler nicht erkennen kann, ob das Feld selbst das eine Element der Rückgabe-List ist oder ob das Feld die Vararg-Umsetzung ist und alle Elemente des Feldes in die Rückgabeliste kommen:

```
List<String> list11 = Arrays.asList( new String[] { "A", "B" } );
List<String> list12 = Arrays.asList( "A", "B" ); // Parameter ist als vararg
definiert
System.out.println( list11 ); // [A, B]
System.out.println( list12 ); // [A, B]
List<String> list21 = Arrays.<String>asList( new String[] { "A", "B" } );
List<String> list22 = Arrays.<String>asList( "A", "B" );
System.out.println( list21 ); // [A, B]
System.out.println( list22 ); // [A, B]
List<String[]> list31 = Arrays.<String[]>asList( new String[] { "A", "B" } );
```

zB

9

Beispiel (Forts.)

```
// List<String[]> list32 = Arrays.<String[]>asList( "A", "B" );
System.out.println( list31 ); // [[Ljava.lang.String;@69b332]
```

Zunächst gilt es festzuhalten, dass die Ergebnisse für list11, list12, list21 und list22 identisch sind. Der Compiler setzt ein Vararg automatisch als Feld um und übergibt das Feld der asList()-Methode. Im Bytecode sehen daher die Aufrufe gleich aus.

Bei list21 und list22 ist der Typparameter jeweils explizit angegeben, aber nicht wirklich nötig, da ja das Ergebnis wie list11 bzw. list12 ist. Doch der Typparameter String macht deutlich, dass die Elemente im Feld, also die Vararg-Argumente, Strings sind. Spannend wird es bei list31. Zunächst zum Problem: Ist new String[]{"A", "B"} das Argument einer Vararg-Methode, so ist das mehrdeutig, weil genau dieses Feld das erste Element des vom Compiler automatisch aufgebauten Varargs-Feldes sein könnte (dann wäre es ein Feld im Feld) oder – und das ist die interne Standardumsetzung – der Java-Compiler das übergebene Feld als die Vararg-Umsetzung interpretiert. Diese Doppeldeutigkeit löst <String[]>, da in dem Fall klar ist, dass das von uns aufgebaute String-Feld das einzige Element eines neuen Varargs-Feldes sein muss. Und Arrays.<String[]> asList() stellt heraus, dass der Typ der Feldelemente String[] ist. Daher funktioniert auch die letzte Variablendeclaration nicht, denn bei asList("A", "B") ist der Elementtyp String, aber nicht String[].

9.2 Umsetzen der Generics, Typlösung und Raw-Types

Zum Verständnis der Generics und um zu erfahren, was zur Laufzeit an Informationen vorhanden ist, lohnt es sich, sich anzuschauen, wie der Compiler Generics in Bytecode übersetzt.

9.2.1 Realisierungsmöglichkeiten

Im Allgemeinen gibt es zwei Möglichkeiten, um generische Typen zu realisieren:

- **Heterogene Variante:** Für jeden Typ (etwa String, Integer, Point) wird individueller Code erzeugt, also drei Klassendateien. Die Variante nennt sich auch *Code-Spezialisierung*.

- **Homogene Übersetzung:** Aus der parametrisierten Klasse wird eine Klasse erzeugt, die anstelle des Typparameters nur `Object` einsetzt. Für den konkreten Typparameter werden Typanpassungen in die Anweisungen eingebaut.
- Java nutzt die homogene Übersetzung, und der Compiler erzeugt nur eine Klassendatei. Es gibt keine multiplen Kopien der Klasse – weder im Bytecode noch im Speicher.

9.2.2 Typlösung (Type Erasure)

Übersetzt der Java-Compiler die generischen Anwendungen, so löscht er dabei alle Typinformationen, da die Java-Laufzeitumgebung keine Generics im Typsystem hat. Das nennt sich *Typlösung* (engl. *type erasure*). Wir können uns das so vorstellen, dass alles wegfällt, was in spitzen Klammern steht, und dass jede Typvariable zu `Object` wird.⁷

Mit Generics	Nach der Typlösung
<pre>public class Pocket<T> { private T value; public void set(T value) { this.value = value; } public T get() { return value; } }</pre>	<pre>public class Pocket { private Object value; public void set(Object value) { this.value = value; } public Object get() { return value; } }</pre>

Tabelle 9.5: Generische Klasse im Quellcode und wie sie nach der Typlösung aussieht

So entspricht der Programmcode nach der Typlösung genau dem, was wir selbst auch ohne Generics am Anfang programmiert haben. Auch bei der Nutzung wird gelöscht:

Mit Generics	Nach der Typlösung
<pre>Pocket<Integer> p = new Pocket<Integer>(1); p.set(1); Integer i = p.get();</pre>	<pre>Pocket p = new Pocket(1); p.set(1); Integer i = (Integer) p.get();</pre>

Tabelle 9.6: Nutzung generischer Klassen und wie es nach der Typlösung aussieht

⁷ Sind *Bounds* im Spiel – eine Typeinschränkung, die später noch vorgestellt wird –, wird ein präziserer Typ statt `Object` genutzt.

Beim Herausholen über `get()` fügt der Compiler genau die explizite Typanpassung ein, die wir in unserem ersten Beispiel noch von Hand eingesetzt haben.

Aber...



Wenn der Compiler Bytecode erzeugt, der auch für ältere JVMs keine Probleme bereitet, so stellt sich die Frage, wo denn die Informationen abgespeichert sind, dass ein Typ generisch deklariert wurde oder nicht. Irgendwo muss das stehen, denn der Compiler weiß das ja. Die Antwort ist, dass der Compiler diese Typinformationen, die nicht Teil des Typsystems der JVM sind, als Signatur-Attribute in den Konstantenpool des Bytecodes legt. Das Attribut ist ein UTF-8 Text, der von älteren Compilern als Kommentar überlesen wird. Mit dem Diassembler `javap` und dem Schalter `-verbose` lassen sich diese Informationen anzeigen. Interessierte bekommen weitere Informationen unter:
http://java.sun.com/docs/books/jvms/second_edition/jvms-clarify.html.

Das große Ziel: Interoperabilität

Interoperabilität stand bei der Einführung der Generics ganz oben auf der Wunschliste. Zwei wichtige Anforderungen sind:

- Die neuen mit Generics deklarierten Klassen – wie `List<E>` – müssen auf jeden Fall noch von altem Programmcode, der zum Beispiel mit einem Java 1.4-Compiler erzeugt wurde, nutzbar sein. Das funktioniert so, dass generische deklarierte Klassen im Bytecode für einen »alten« Compiler so aussehen, als gäbe es keine Generics. Wir sprechen von *Typlösung*. Hätte Sun sich nicht dieses Kompatibilitätsziel auf die Fahnen geschrieben, hätte die Umsetzung auch anders ausfallen können. Denn die Konsequenz der Typlösung ist, dass es keine Informationen über den Typparameter zur Laufzeit gibt. Das führt zu Überraschungen und Einschränkungen (insbesondere bei Arrays), die wir uns gleich anschauen werden. Was wir hier vor uns haben, ist der Wunsch nach *Bytecode-Kompatibilität*.
- Auf der anderen Seite gibt es neben der Bytecode-Kompatibilität auch noch die *Quellcode-Kompatibilität*. Alter Programmcode, der zum Beispiel Listen als `List list;` statt zum Beispiel als `List<String> list;` nutzt, soll immer noch übersetzbare sein, auch wenn er die überarbeiteten Datenstrukturen nicht generisch nutzt. Warnungen sind akzeptabel, aber keine Compilerfehler. Es gibt Millionen Zeilen alten Quellcodes, die Listen ohne Generics nutzen, ohne dass sofort ein Team alle Programmstellen anfasst und Typparameter einführt.



Java Generics und C++-Templates

Java Generics gehen bei den Typbeschreibungen weit über das hinaus, was C++-Templates bieten. In C++ kann ein beliebiger Typparameter eingesetzt werden – was zu unglaublichen Fehlermeldungen führt. Der C++-Compiler führt somit eher eine einfache Ersetzung durch. Doch durch die heterogene Umsetzung generiert der C++-Compiler für jeden genutzten Template-Typ unterschiedlichen (und wunderbar optimierten) Maschinencode. Im Fall von Java würde die heterogene Variante zu sehr vielen sehr ähnlichen Klassen führen, die sich nur in ein paar Typanpassungen unterschieden. Und da in Java sowieso nur Referenzen als Typvariablen möglich sind und keine primitiven Typen, ist auch eine besondere Optimierung an dieser Stelle nicht möglich. Durch die Code-Spezialisierung sind aber andere Dinge in C++ machbar, die in Java unmöglich sind, zum Beispiel Template-Metaprogramming. Der Compiler wird in diesem Fall als eine Art Interpreter für rekursive Template-Aufrufe genutzt, um später optimalen Programmcode zu generieren. Das ist funktionale Programmierung mit einem Compiler ...

9.2.3 Probleme aus der Typlösung

Typlösung ist für die Laufzeitumgebung praktisch, weil sie überhaupt nicht an die Generics angepasst werden muss. So sehen zum Beispiel die seit Java 5 generisch deklarierten Datenstrukturen nach dem Übersetzungsvorgang genauso aus wie unter Java 1.4 und sind damit voll kompatibel. Sonst aber stellt die Typlösung ein riesiges Problem dar, weil die Typinformationen zur Laufzeit nicht vorhanden sind.⁸



Reified Generics

Für Java 7 stand auf der Aufgabenliste, die generischen Parameter auch zur Laufzeit zugänglich zu machen. Das wurde jedoch verschoben und kommt vielleicht irgendwann, in Java 8, Java 9, Java 2020, ... Das Stichwort dazu ist *Reified Generics*, also generische Informationen, die auch zur Laufzeit komplett zugänglich sind.

⁸ Dass diese Typinformationen nicht vorliegen, wird auch damit begründet, dass die Laufzeit leiden könnte. Microsoft war das hingegen egal, dort besteht Generizität in der *Common Language Runtime* (CLR), also auch in der Laufzeitumgebung. Microsoft ist damit einen klaren Schritt voraus. Doch gab es Generics (*Parametric Polymorphism* ist der offizielle Name) auch wie in Java nicht von Anfang an; es zog erst in Version 2 in die Sprache und CLR ein. Die alten Datenstrukturen wurden einfach als veraltet markiert, und die Entwickler waren gezwungen, auf die neuen generischen Varianten umzusteigen.

Kein new T

Da durch die Typlösung bei Deklarationen wie `Pocket<T>` die Parametervariable durch `Object` ersetzt wird, lässt sich zum Beispiel in der Tasche *nicht* Folgendes schreiben, um ein neues Exemplar eines Tascheninhalts zu erzeugen:

Gedacht: Mit Generics (Compilerfehler!)	Konsequenz aus Typlösung
<pre>class Pocket<T> { T newPocketContent() { return new T(); } }</pre>	<pre>class Pocket<T> { Object newPocketContent() { return new Object(); } }</pre>

Tabelle 9.7: Warum »new T()« nicht funktionieren kann: nur ein »new Object()« würde gebildet

Als Aufrufer von `newPocketContent()` erwarten wir aber nicht immer ein lächerliches `Object`, sondern ein Objekt vom Typ `T`.

Kein instanceof

Der `instanceof`-Operator ist bei parametrisierten Typen ungültig, auch wenn das praktisch wäre, um zum Beispiel aufgrund der tatsächlichen Typen eine Fallunterscheidung vornehmen zu können:

```
void printType( Pocket<?> p )
{
    if ( p instanceof Pocket<Number> )           // ❌ illegal generic type for instanceof
        System.out.println( "Pocket mit Number" );
    else if ( p instanceof Pocket<String> )        // ❌ illegal generic type for instanceof
        System.out.println( "Pocket mit String" );
}
```

Der Compiler meldet zu Recht einen Fehler – nicht nur eine Warnung –, weil es die Typen `Pocket<String>` und `Pocket<Number>` zur Laufzeit gar nicht gibt: Es sind nur typgelöschte `Pocket`-Objekte. Nach der Typlösung würde unsinniger Code entstehen:

```
void printType( Pocket p )
{
    if ( p instanceof Pocket )
    ...
}
```

```

else if ( p instanceof Pocket )
    ...
}

```

Keine Typanpassungen auf parametrisierten Typ

Typanpassungen wie

```
Pocket<String> p = (Pocket<String>) new Pocket<Integer>(); // ☹ Compilerfehler
```

sind illegal. Wir haben ja extra Generics, damit der Compiler die Typen testet. Und durch die Typlöschung verschwindet der Typparameter, sodass der Compiler Folgendes erzeugen würde:

```
Pocket p = (Pocket) new Pocket();
```

Kein .class für generische Typen und keine Class-Objekte mit Typparameter zur Laufzeit

Ein hinter einen Typ gesetztes .class liefert das Class-Objekt zum jeweiligen Typ.

```
Class<Object> objectClass = Object.class;
Class<String> stringClass = String.class;
```

Class selbst ist als generischer Typ deklariert.

Bei generischen Typen ist das .class nicht erlaubt. Zwar ist noch (mit Warnung) Folgendes gültig:

```
Class<Pocket> pocketClass = Pocket.class;
```

aber dies nicht mehr:

```
Class<Pocket<String>> pocketClass = Pocket<String>.class; // ☹ Compilerfehler
```

Der Grund ist die Typlöschung: Alle Class-Objekte für einen Typ sind gleich und haben zur Laufzeit keine Information über den Typparameter:

```
Pocket<String> p1 = new Pocket<String>();
Pocket<Integer> p2 = new Pocket<Integer>();
System.out.println( p1.getClass() == p2.getClass() ); // true
```

Alle Exemplare von generischen Typen werden zur Laufzeit vom gleichen Class-Objekt repräsentiert. Hinter `Pocket<String>` und `Pocket<Integer>` steckt also immer nur `Pocket`. Kurz gesagt: Alles in eckigen Klammern verschwindet zur Laufzeit.

Keine generischen Ausnahmen

Grundsätzlich ist eine Konstruktion wie `class MyClass<T> extends SuperClass` erlaubt. Aber der Compiler enthält eine spezielle Regel, die verhindert, dass eine generische Klasse `Throwable` (`Exception` und `Error` sind Unterklassen von `Throwable`) erweitern kann. Wäre zum Beispiel

```
class MyException<T> extends Exception { } // ☹ Compilerfehler
```

erlaubt, könnte im Quellcode vielleicht ein

```
try { }
catch ( MyException<Typ1> e ) { }
catch ( MyException<Typ2> e ) { }
```

stehen, doch durch die Typlösung würde das auf zwei identische `catch`-Blöcke hinauslaufen, was nicht erlaubt ist.

Keine statischen Eigenschaften

Statische Eigenschaften hängen nicht an einzelnen Objekten, sondern an Klassen. `Pocket` kann zum Beispiel einmal als parametrisierter Typ `Pocket<String>` und einmal als `Pocket<Integer>` auftauchen, also als zwei Instanzen. Aber kann `Pocket` auch eine statische Methode deklarieren, die auf den formalen Typparameter der Klasse zurückgreift? Nein, das geht nicht. Würden wir in `Pocket` etwa die folgende statische Methode einsetzen

```
public static boolean isEmpty( T value ) { return value == null; } // ☹
```

so gäbe es bei `T` die Fehlermeldung »Cannot make a static reference to the non-static type `T`«.

Statische Variablen und die Parameter/Rückgaben von statischen Methoden sind nicht an ein Exemplar gebunden. Eine Typvariable jedoch, so wie wir sie bisher verwendet haben, ist immer mit dem Exemplar verbunden. Das `T` für den `value` ist ja erst immer dann festgelegt, wenn wir zum Beispiel `Pocket<String>` oder `Pocket<Integer>` mit einem Exemplar verbinden. Bei `Pocket.isEmpty("")`; zum Beispiel kann der Compiler nicht wissen, was für ein Typ gemeint ist, da für statische Methodenaufrufe ja keine Exemplare nötig

sind, also nie ein parametrisierter Typ festgelegt werde. Das Nutzen von Code wie `Pocket<String>.isEmpty("")` führt zu einem Compilerfehler, denn die Syntax ist nicht erlaubt.

Statische generische Methoden sind natürlich möglich, wie wir schon gesehen haben; sie haben dann eine eigene Typvariable.

Kein Überladen mit Typvariablen

Kommt nach der Typlöschung einfach nur `Object` heraus, kann natürlich keine Methode einmal mit einer Typvariablen und einmal mit `Object` parametrisiert sein. Folgendes ist nicht erlaubt:

```
public class Pocket<T>
{
    public T value;
    public void set( T value ) { this.value = value; }
    public void set( Object value ) { this.value = value; } // ☹ Compilerfehler!
}
```

Der Compiler liefert: »Method `set(T)` has the same erasure `set(Object)` as another method in type `Pocket<T>`«.

Ist der Typ spezieller, also etwa `String`, sieht das wieder anders aus. Dann taucht die Frage auf, welche Methode bei `Pocket<String>` aufgerufen wird. Die Leser dürfen das gerne prüfen.

Es lassen sich keine Arrays generischer Klassen bilden

Die Nutzung von Generics bei Arrays schränkt der Compiler ebenfalls ein. Während

```
Pocket[] pockets = new Pocket[1];
```

gültig ist und mit einer Warnung versehen wird, führt bei

```
Pocket<String>[] pockets; // (1)
pockets = new Pocket<String>[1]; // (2) ☹ Compilerfehler
```

nicht die erste, aber die zweite Zeile zum Compilerfehler »Cannot create a generic array of `Pocket<String>`«.

Typsicher kann das nicht genutzt werden, aber drei schnelle Lösungen sind denkbar:

- auf Generics ganz zu verzichten und ein `@SuppressWarnings("unchecked")` an die Feldvariable zu setzen
- den Typ durch ein Wildcard ersetzen, sodass es etwa zu einem `Pocket<?>[] pockets = new Pocket<?>[1];` kommt. Wildcards sind Platzhalter, die später noch detaillierter vorgestellt werden.
- gleich auf Datenstrukturen der Collection-API umsteigen, bei denen ein `Collection<String> pockets = new ArrayList<String>();` keine Probleme bereitet

Als Zusammenfassung lässt sich festhalten, dass Array-Variablen von generischen Typen zwar deklariert (1), dass aber keine Array-Objekte gebaut werden können (2). Mit einem Trick funktioniert es:

```
class PocketFullOfMoney extends Pocket<BigInteger> {}  
Pocket<BigInteger>[] pockets = new PocketFullOfMoney[1];
```

Hübsch ist das nicht, denn es muss extra eine temporäre Klasse angelegt werden.

9.2.4 Raw-Type

Generische Klassen müssen nicht unbedingt parametrisiert werden, doch es ist einleuchtend, dass wir dem Compiler so viel Typinformation wie möglich geben sollten. Auf die Typparameter zu verzichten ist nur für die Rückwärtskompatibilität wichtig, da sonst viele parametrisierte neue Klassen nicht mehr mit altem Programmcode verwendet werden könnten. Wenn zum Beispiel `Pocket` unter Java 1.4 deklariert und mit den Sprachmitteln von Java 5 zu einem generischen Typ verfeinert wurde, kann es immer noch alten Programmcode geben, der wie folgt aussieht:

```
Pocket p = new Pocket();           // Gefährlich, wie wir gleich sehen werden  
p.set( "Drei Pleitegeier, die Taschen voller Sand" );  
String content = (String) p.get();
```

Ein generischer Typ, der nicht als parametrisierter Typ, also ohne Typargument, genutzt wird, heißt *Raw-Type*. In unserem Beispiel ist `Pocket` der Raw-Type von `Pocket<T>`. Bei einem Raw-Type kann der Compiler die Typkonformität nicht mehr prüfen, denn es ist der Typ nach der Typlösung; `get()` liefert `Object`, und `set(Object)` kann alles annehmen.

Ein unter Java 1.4 geschriebenes Programm nutzt also nur Raw-Types. Trifft ein Java 5-Compiler auf Programmcode, der einen generischen Typ nicht als parametrisierten Typ nutzt, fängt er an zu meckern, denn er wünscht, dass der Typ generisch verwendet wird.

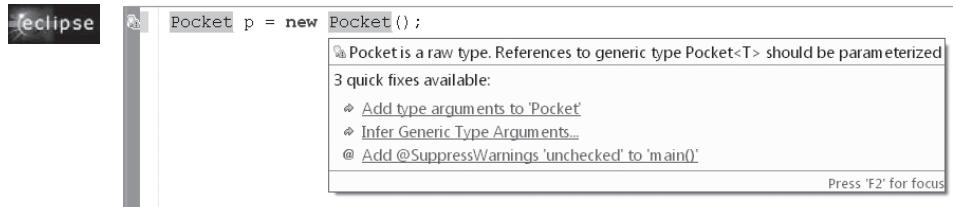


Abbildung 9.1: Eclipse warnt Raw-Types standardmäßig an

Auch bei `set()` gibt der Compiler eine Warnung aus, denn er sieht eine Gefahr für die Typsicherheit. Die Methode `set()` ist so entworfen, dass sie ein Argument von dem Typ akzeptiert, mit dem sie parametrisiert wurde. Fehlt durch die Verwendung des Raw-Types der konkrete Typ, bleibt `Object`, und der Compiler gibt bei den sonst mit einem Typ präzisierten Methoden eine Warnung aus:

```
p.set( "Type safety: The method set(Object) belongs to the " +
        "raw type Pocket. References to generic type " +
        "Pocket<T> should be parameterized" );
```

Der Hinweis besagt, dass die Tasche hätte typisiert werden müssen. Wenn wir nicht darauf achten, kann das schnell zu Problemen führen:

```
Pocket<String> p1 = new Pocket<String>();
Pocket p2 = p1;                      // Compiler-Warnung
p2.set( new java.util.Date() );       // Compiler-Warnung
String string = p1.get();             // ☹ ClassCastException
System.out.println( string );
```

Der Compiler gibt keinen Fehler, aber Warnungen aus. Die dritte Zeile ist hochgradig problematisch, denn über die nicht parametrisierte Tasche können wir beliebige Objekte eintüten. Da aber das Objekt hinter `p2` und dem typgelöschten `p1` identisch ist, haben wir ein Typproblem, das zur Laufzeit zu einer `ClassCastException` führt:

```
Exception in thread "main" java.lang.ClassCastException: java.util.Date cannot be cast to java.lang.String
```

Es kann also nur die Empfehlung ausgesprochen werden, Raw-Types in neuen Programmen zu vermeiden, da ihre Verwendung zu Ausnahmen führen kann, die erst zur Laufzeit auffallen.

Typanpassungen

Ein Raw-Type lässt sich automatisch in eine speziellere Form bringen, wobei es natürlich Warnungen vom Compiler gibt.

```
Pocket p = new Pocket();           // (1) Warnung
p.set( "Roh macht nicht froh" );   // (2) Warnung
Pocket<String> stringPocket = p;    // (3) Warnung
String result = stringPocket.get(); // (4)
```

Bei der Variablen p, die wir über den Raw-Type nutzen (2), prüft der Compiler gar keine Typen in set(), denn er hat sie ja nie kennengelernt. Zeile (3) verkauft dem Compiler den Raw-Type als parametrisierten Typ. Eine explizite Typanpassung ist nicht nötig, denn Casts sind nur zwischen »echten« Typen gültig, wie Object auf Pocket, nicht aber von Pocket auf Pocket<String>, da Pocket<String> ja der gleiche Class-Typ ist (siehe »Kein .class für generische Typen und keine Class-Objekte mit Typparameter zur Laufzeit«). Eine Anweisung wie (4), die einen Nicht-String-Typ in die Tasche setzt, bringt keinen Fehler zur Übersetzungszeit, und so kann auch über diese Hintertür ein falscher Typ in die Tasche kommen.

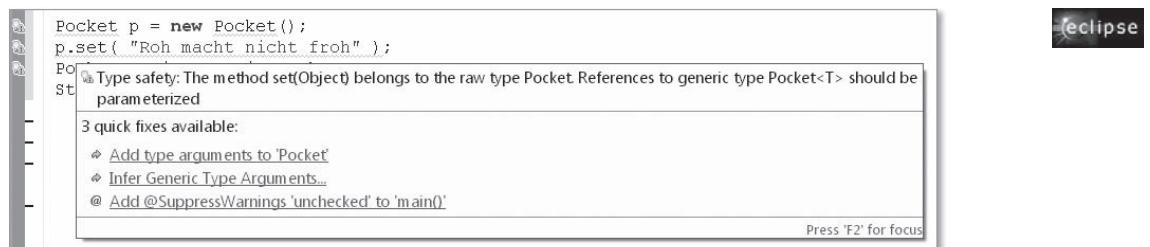


Abbildung 9.2: Warnung von Eclipse bei Raw-Types

Annotation SuppressWarnings

In seltenen Fällen muss in den Typ konvertiert werden. Als Beispiel soll cast() dienen:

```
public <T> T cast( Object obj )
{
    return (T) obj; // Compilerwarnung: Type safety: Unchecked cast from Object to T
}
```

Lässt sich der Cast nicht vermeiden, um dem Compiler den Typ zu geben und ihn somit glücklich zu machen, setzen wir eine @SuppressWarnings-Annotation:

```
@SuppressWarnings("unchecked")
public <T> T cast( Object obj )
{
    return (T) obj;
}
```

Die Generics bieten uns Möglichkeiten, den Quellcode sicherer zu machen. Wir sollten diese Sicherheit nicht durch Raw-Types kaputtmachen.



Unter den PREFERENCES von Eclipse können drei Typen von Hinweisen für die Nutzung von Raw-Types angegeben werden: Der Compiler gibt einen harten Compilerfehler aus, eine Warnung, oder er ignoriert sie. Dass er Warnungen ausgibt, ist voreingestellt, und diese Vorgabe ist ganz gut.

9.3 Einschränken der Typen über Bounds

Bei generischen Angaben können die Typen weiter eingeschränkt werden. Das ist nützlich, da ein beliebiger Typ oft zu allgemein ist. Unsere Deklaration von `random()` sah keine Einschränkungen für die Typen vor:

```
public static <T> T random( T m, T n )
{
    return Math.random() > 0.5 ? m : n;
}
```

So ist auch Folgendes möglich:

```
Object o1 = new Object();
Object o2 = new Point();
System.out.println( random( o1, o2 ) );
```

Da der Typ beliebig ist, können auch Objekte übergeben werden, die vielleicht wenig Sinn ergeben, insbesondere in ihrer Kombination.

9.3.1 Einfache Einschränkungen mit extends

Bei der Deklaration eines generischen Typs kann vorgeschrieben werden, dass der spätere parametrisierte Typ eine bestimmte Klasse erweitert oder eine konkrete Schnitt-

stelle implementiert. Soll unsere statische `random()`-Methode zum Beispiel nur Objekte vom Typ `CharSequence` (also Zeichenfolgen wie `String` und `StringBuffer/StringBuilder`) akzeptieren, so schreiben wir das in die Deklaration mit hinein:

Listing 9.9: com/tutego/insel/generic/BondageBounds.java, BondageBounds

```
public class BondageBounds
{
    public static <T extends CharSequence> T random( T m, T n )
    {
        return Math.random() > 0.5 ? m : n;
    }

    public static void main( String[] args )
    {
        String random1 = random( "Shinju", "Karada" );
        System.out.println( random1 );

        CharSequence random2 = random( "Ushiro", new StringBuilder("Takatekote") );
        System.out.println( random2 );
    }
}
```

Einen Aufruf mit zwei `Strings` lässt der Compiler korrekterweise durch, genauso wie mit `String` und `StringBuilder`, wobei der Rückgabetyp dann nur noch `CharSequence` ist.

Ein Fehler ist leicht zu provozieren. Dazu muss der Methode `random()` nur etwa ein `Point` übergeben werden – `Point` ist nicht vom Typ `CharSequence`. So führt

```
System.out.println( random( "", new Point() ) ); // ☹ Compilerfehler
                                                // »Bound mismatch«
```

zu der Fehlermeldung: »`Bound mismatch: The generic method random(T, T) of type BondageBounds is not applicable for the arguments (String, Point). The inferred type Serializable is not a valid substitute for the bounded parameter <T extends CharSequence>«.`

Der Compiler führt eine Typ-Inferenz durch; das heißt, er schaut sich an, welche gemeinsamen Typen die Argumente `""` und `Point` haben, und kommt auf `Serializable`. Der Typ hilft jedoch nicht weiter, denn wir wollten bloß `CharSequences` einsetzen können.

Typeeinschränkung für gemeinsame Methoden

Eine Typeeinschränkung wie `<T extends CharSequence>` ist interessant, da wir so wissen, dass ein konkreter Typparameter (etwa `String` oder `StringBuffer`) mindestens die Methoden der Schnittstelle `CharSequence` hat. Das ist logisch, denn bei einer Einschränkung des Typs wird der Compiler sicherstellen, dass die konkreten Typen die vorgeschriebene Schnittstelle implementieren (oder die Klasse erweitern) und damit die Methoden existieren.

Nehmen wir an, wir wollten ein typsicheres `max()` implementieren. Es soll den größeren der beiden Werte zurückgeben. Vergleiche lassen sich einfach tätigen, wenn die Objekte `Comparable` implementieren, denn `compareTo()` liefert einen Rückgabewert, der aussagt, welches Objekt nach der definierten Metrik kleiner, größer oder gleich ist.

Listing 9.10: com/tutego/insel/generic/BondageBounds.java, max()

```
public static <T extends Comparable<T>> T max( T m, T n )
{
    return m.compareTo( n ) > 0 ? m : n;
}
```

Die Nutzung ist einfach:

```
System.out.println( max( "Kino", "Lesen" ) );           // Lesen
System.out.println( max( 12, 100 ) );                   // 100
```



Hinweis

Ohne Typ-Bound können nur die Methoden von `Object` verwendet werden, aber immerhin noch Methoden wie `equals()`, `hashCode()` und `toString()`.

Betrachten wir noch einen Fehlerfall. Intuitiv ist anzunehmen, dass alles, was vom Typ `Comparable` ist, auch ein gültiges Argument für `max()` ist. Das ist aber nicht ganz präzise, denn wir schreiben `<T extends Comparable<T>> T max(T m, T n)`, was bedeutet, dass der *gemeinsame* Typ für `m` und `n` laut Typ-Inferenz `Comparable` sein muss, nicht nur jeder *einzelne*. Was das bedeutet, zeigt die folgende Anweisung, die der Compiler mit einem Fehler ablehnt:

```
System.out.println( max( 12L, 100F ) ); // ☹ Compilerfehler »Bound mismatch«
```

Nach dem Boxing leitet der Compiler aus dem `Long 12` und dem `Float 100` den gemeinsamen Typ `Number` ab. (Zur Erinnerung: Der Compiler geht in der Typhierarchie so lange

nach oben, bis ein gemeinsamer Typ gefunden wurde, der für T eingesetzt werden kann. Das ist Number.) Aber Number ist nicht vom Typ Comparable, und so folgt eine Fehlermeldung: »Bound mismatch: The generic method max(T, T) of type BondageBounds is not applicable for the arguments (Long, Float). The inferred type Number&Comparable<?> is not a valid substitute for the bounded parameter <T extends Comparable<T>>«.

Hinweis

Bei `<T extends Comparable<T>>` handelt es sich um einen sogenannten *rekursiven Type-Bound*. Er kommt selten vor und soll hier auch nicht weiter vertieft werden. Bei Interesse gibt <http://tutego.de/go/getthistrick> weitere Hinweise. Wird die Methode `max()` zum Beispiel fälschlicherweise mit

```
static <T extends Comparable/*Hier fehlt was*/> T max( T m, T n ) { ... }
```

deklariert, so gibt der Compiler die Warnung »Comparable is a raw type. References to generic type Comparable<T> should be parameterized« aus. Ignorieren wir die Warnung, so lässt sich `max(12L, 100F)` tatsächlich aufrufen, doch es folgt eine ClassCastException mit »java.lang.Float cannot be cast to java.lang.Long«.

9

9.3.2 Weitere Obertypen mit &

Soll der konkrete Typ zu mehreren Typen passen, lassen sich mit einem & weitere Obertypen hinzunehmen. Wichtig ist aber, dass nur eine Klassen-Vererbungsangabe stattfinden kann, also nur ein `extends` stehen darf, da Java keine Mehrfachvererbung auf Klassenebene unterstützt. Bei dem Rest muss es sich um implementierte Schnittstellen handeln. Die allgemeine Notation (für eine Klasse C und Schnittstellen I₁ bis I_n) ist:

`T extends C & I1 & I2 & ... & In`

Nehmen wir eine fiktive Oberklasse Endeavour und die Schnittstellen Serializable und Comparable an.⁹ Dann sind die folgenden Deklarationen prinzipiell erlaubt:

- `<T extends Endeavour>`
- `<T extends Serializable & Comparable>`
- `<T extends Endeavour & Serializable>`
- `<T extends Endeavour & Comparable & Serializable>`

⁹ Comparable bekommt selbst einen Typparameter, was das Beispiel aus Gründen der Übersichtlichkeit auslässt.

Syntaktisch falsch wäre etwa `<T extends Endeavour & T extends Comparable>`, da das Schlüsselwort `extends` nur einmal vorkommen darf.

9.4 Typparameter in der throws-Klausel *

Wir haben im Abschnitt »Keine generischen Ausnahmen« gesehen, dass durch die Typlösung eine Konstruktion wie `class MyException<T> extends Exception` nicht möglich ist. Allerdings ist ein Typparameter in der `throws`-Klausel erlaubt. Das gibt interessante Möglichkeiten für Klassen, die je nach Anwendungsfall einmal geprüfte oder ungeprüfte Ausnahmen auslösen können.

9.4.1 Deklaration einer Klasse mit Typvariable `<E extends Exception>`

Unsere Schnittstelle `CharIterable` soll von Klassen implementiert werden, die einen Strom von Zeichen liefern. `CharIterable` ist ein generischer Schnittstellentyp mit einem formalen Typparameter, der später eine Unterklasse von `Exception` sein muss:

Listing 9.11: com/tutego/insel/generic/CharIterable.java, CharIterable

```
public interface CharIterable<E extends Exception>
{
    boolean hasNext() throws E;
    char     next()     throws E;
}
```

Zeichen können etwa aus einer Datei, von einer Internetressource oder von einem String kommen, doch die Nutzung sieht immer gleich aus:

```
while ( iter.hasNext() )
    System.out.print( iter.next() );
```

9.4.2 Parametrisierter Typ bei Typvariable `<E extends Exception>`

Kommen wir zu den Klassen, die `CharIterable` implementieren, sodass Nutzer mit der gerade vorgestellten Schleife die Zeichen ablaufen können. Die Deklaration der Schnittstelle `CharIterable<E extends Exception>` enthält eine auf `Exception` eingeschränkte Typvariable, was zum Beispiel die folgenden Implementierungen zulässt:

- class StringIterable implements CharIterable<RuntimeException>
- class WebIterable implements CharIterable<IOException>

Kommen im Fall von StringIterable die Zeichen aus einem String, ist keine Ein-/Ausgabe-Ausnahme zu erwarten, daher ist der Typparameter RuntimeException. Beim Lesen aus Dateien oder Internetressourcen kann es jedoch zu IOExceptions kommen, sodass WebIterable den Typparameter IOException wählt.

Beispielimplementierungen für den parametrisierten Typ

Implementieren wir die beiden Klassen StringIterable und WebIterable. Da StringIterable bei der Implementierung der Schnittstelle den Typparameter RuntimeException wählt, führt das zu einem throws RuntimeException, was wiederum optional ist und weg gelassen werden kann.

Listing 9.12: com/tutego/insel/generic/StringIterable.java, StringIterable

```
public class StringIterable implements CharIterable<RuntimeException>
{
    private final String string;
    private int    pos;

    public StringIterable( String string )
    {
        this.string = string;
    }

    @Override public boolean hasNext()
    {
        return pos < string.length();
    }

    @Override public char next()
    {
        return string.charAt( pos++ );
    }
}
```

Bei `WebIterable` sieht das anders an. Hier ist der Typparameter `IOException`, und somit ist ein `throws IOException` an der Methodensignatur nötig.

Listing 9.13: com/tutego/insel/generic/WebIterable.java, `WebIterable`

```
public class WebIterable implements CharIterable<IOException>
{
    private final Reader reader;

    public WebIterable( String url ) throws IOException
    {
        reader = new InputStreamReader( new URL( url ).openStream() );
    }

    @Override public boolean hasNext() throws IOException
    {
        return reader.ready();
    }

    @Override public char next() throws IOException
    {
        return (char) reader.read();
    }
}
```

Nutzen von `StringIterable` und `WebIterable`

Das folgende Beispiel zeigt, dass beim Ablaufen eines Strings keine Ausnahmebehandlung nötig ist, beim Lesen von Zeichen aus dem Internet aber schon:

Listing 9.14: com/tutego/insel/generic/CharReadableExample.java, `main()`

```
StringIterable iter1 = new StringIterable( "Shasha" ); // try ist unnötig
while ( iter1.hasNext() )
    System.out.print( iter1.next() );

System.out.println();

try
```

```

{
    WebIterable iter2 = new WebIterable( "http://java-tutor.com/aufgaben/bond.txt" );
    while ( iter2.hasNext() )
        System.out.print( iter2.next() );
}
catch ( IOException e )
{
    e.printStackTrace();
}

```

Statt `StringIterable iter1 = new StringIterable...` **hätten wir natürlich auch** `CharIterable<RuntimeException> iter1...` **schreiben können und analog** `CharIterable<IOException> iter2` **statt** `WebIterable iter2.`

Zusammenfassung

Das Beispiel macht deutlich, dass ein Typparameter `RuntimeException` selbst so elementare Dinge wie geprüfte Ausnahmen ausschaltet. Die Besonderheit liegt beim Compiler, dass er Dinge wie `throws E` zulässt und dass `E` dann einmal eine geprüfte oder ungeprüfte Ausnahme sein kann. Exakt so hatten wir `CharIterable` deklariert:

```

public interface CharIterable<E extends Exception>
{
    boolean hasNext() throws E;
    char    next()    throws E;
}

```

Das ist sehr praktisch, denn unser Anwendungsfall macht deutlich, dass es gut ist, einmal geprüfte Ausnahmen zu verwenden, denn geprüfte Aufnahmen verlangen ja immer etwas mehr Aufwand und sind immer nötig, wie das Ablaufen von Strings zeigt.

API-Design der Klasse Scanner



Scanner ist ein Beispiel für eine Klasse, in der die Java-API-Designer geprüfte Ausnahmen bei den `next()`-Methoden nicht haben wollten. Die Klasse kann normale Strings zerlegen, bei denen `next()` keine `IOException` auslösen kann. Aber Scanner kann auch einen Eingabestrom bekommen, und dann sind Ein-/Ausgabeausnahmen durchaus möglich.



API-Design der Klasse Scanner (Forts.)

Was also tun? Entweder bei den next()-Methoden immer eine IOException auslösen oder nie? Lösen sie keine Ausnahme aus (das ist das Design jetzt), so bleiben die Fehler auf der Strecke, die beim Einlesen aus dem Datenstrom auftreten können. Trügen die next()-Methoden jedoch ein throws IOException, dann wäre das lästig beim Zerlegen von puren Strings – und das wollten die Entwickler nicht. Daher fällt die IOException bei next() unter den Tisch und muss explizit über die Methode IOException() erfragt werden. Das steht so ganz im Gegensatz zu der Idee, bei Ein-/Ausgabefehlern immer geprüfte Ausnahmen zu verwenden. Beim PrintWriter ist das übrigens genauso, die write() und printXXX()-Methoden lösen keine IOException aus, sondern Entwickler fragen später mit checkError() nach, ob es Probleme gab. Leser können überlegen, ob Scanner<E extends Exception> und Methoden wie next() throws E das Problem lösen würden.

9.5 Generics und Vererbung, Invarianz

Vererbung und Substitution ist für Java-Entwickler alltäglich, sodass diese Eigenschaft nicht weiter verwunderlich ist. Die `toString()`-Methode zum Beispiel wird ganz natürlich auf allen Objekten aufgerufen, und Entwicklern ist klar, dass der Aufruf dynamisch gebunden ist. Genauso lässt sich bei `String.toString(Object o)` jedes Objekt übergeben, und die statische Methode ruft die Objektmethode `toString()` auf.

9.5.1 Arrays sind invariant

Nehmen wir als folgendes Beispiel die Hierarchie der bekannten Wrapper-Klassen. Natürlich steht `Object` oben. Die numerischen Wrapper-Klassen implementieren alle `Number`. Darunter stehen dann etwa `Integer`, `Double` und die anderen numerischen Wrapper. Folgendes bereitet keine Kopfschmerzen:

```
Number number = Integer.valueOf( 10 );
number = Double.valueOf( 1.1 );
```

Einmal zeigt `number` auf ein `Integer`, dann auf ein `Double`-Objekt.

Wie verhält es sich nun mit Arrays? Da ist ein Number-Array der Basistyp eines Double-Arrays:

```
Number[] numbers = new Double[ 100 ];
numbers[ 0 ] = 1.1;
```

Dass ein Array vom Typ Double[] ein Untertyp von Number[] ist, nennt sich *Invarianz*. Doch lässt sich das auch auf Generics übertragen?

9.5.2 Generics sind kovariant

Es funktioniert, Folgendes zu schreiben:

```
Set<String> set = new HashSet<String>();
```

Ein HashSet mit Strings ist eine Art von Set mit Strings. Aber ein HashSet mit Strings ist kein HashSet mit Objects. Damit wäre Folgendes falsch:

```
HashSet<Object> set = new HashSet<String>(); // ☹ Compilerfehler!
```

Generics sind nicht invariant, sie sind *kovariant*. Diese Eigenschaft ist auf den ersten Blick gegen die Intuition, doch ein Beispiel rückt diesen Eindruck schnell gerade. Bleiben wir bei unserem Pocket und den Wrapper-Klassen. Auch wenn Number die Oberklasse von Integer ist, so gilt dennoch nicht, dass Pocket<Number> ein Obertyp von Pocket<Integer> ist. Wäre es das, wäre Folgendes möglich und zur Laufzeit ein Problem:

```
Pocket<Number> p;
p = new Pocket<Integer>(); // Ist das OK?
p.set( 2.2 );
```

Das Argument 2.2 ist über Autoboxing ein Double, und daher scheint es auf Number zu passen. Allerdings sollte Double aber gar nicht erlaubt sein, da wir mit Pocket<Integer> ja eine Tasche für Integer aufgebaut haben, und ein Double darf nicht in die Integer-Tasche. Daher folgt: Die Ableitungsbeziehung zwischen Typen überträgt sich nicht auf generische Klassen. Ein Pocket<Number> ist also keine Oberklasse, die alle erdenklichen numerischen Typen in der Tasche erlaubt. Der Compiler meckert bei diesem Versuch sofort:

```
Pocket<Number> p;
p = new Pocket<Integer>(); // ☹ Type mismatch: cannot convert from Pocket<Integer>
                           // to Pocket<Number>
```

Auch durch eine alternative Schreibweise lässt sich der Compiler nicht in die Irre führen:

```
Pocket<Integer> p = new Pocket<Integer>();
Pocket<Number> p2 = p; // ☹ Type mismatch: cannot convert
                        // from Pocket<Integer> to Pocket<Number>
```



Hinweis

Im Fall von immutable Objekten mit Nur-lese-Zugriff bestünde eigentlich kein Grund für Kovarianz. Nehmen wir an, die folgende Deklaration wäre korrekt:

```
Pocket<Number> p = new Pocket<Integer>( 1 );
Number n = p.get();
```

Dann haben wir gezeigt, dass `p.set(2.2)` zum Beispiel nicht in Ordnung ist, da `Double` nicht mit `Integer` kompatibel ist. Aber wenn das Objekt etwa über den Konstruktor initialisiert würde, spräche nichts dagegen, mit einem kleineren Typ, also hier `Number`, daraus zu lesen. Jedoch kann Java nicht erkennen, ob ein Typ immutable ist, und kann daher auch solche Ausnahmen bei den Generics nicht machen. Der Compiler nimmt immer an, Zugriffe wären lesend und schreibend.

9.5.3 Wildcards mit ?

Wir wollen eine Methode `isOnePocketEmpty()` schreiben, die eine variable Anzahl von beliebigen Tascheninhalten bekommt und testet, ob eine davon leer ist. Ein Aufruf könnte so aussehen:

```
Pocket<String> p1 = new Pocket<String>( "Bad-Bank" );
Pocket<Integer> p2 = new Pocket<Integer>( 1500000 );
System.out.println( isOnePocketEmpty( p1, p2 ) );                                // false
```

Die erste Idee für den Methodenkopf sieht so aus:

```
public static boolean isOnePocketEmpty( Pocket<Object>... pockets )
```

Doch halt! Da `Pocket<Object>` nicht Taschen mit allen Typen umfasst, sondern nur exakt eine Tasche trifft, die ein `Object`-Objekt enthält, ist das keine sinnvolle Parametrisierung für `isOnePocketEmpty()`. Das hatten wir im oberen Abschnitt schon festgestellt. Denn wäre das möglich, würde es die Typsicherheit gefährden. Denn wenn diese Methode tatsächlich Taschen mit allen Inhalten akzeptieren würde, so könnte einer Tasche leicht

ein Wert mit falschem Typ untergeschoben werden. Denn wird wie in unserem Beispiel die Methode `isOnePocketEmpty()` mit einem `Pocket<String>` aufgerufen, so würde wegen `isOnePocketEmpty(Pocket<Object>... pockets)` dann auch der Aufruf von `set(12)` auf dem Pocket gültig sein, und dann stände plötzlich statt des gewünschten Inhalts vom Taschentyp `String` nun ein `Integer` in der Tasche. Das darf nicht gültig sein!

Ist der Typ egal, könnten wir an den Original-Typ (Raw-Type) denken. Doch die Raw-Types haben den Nachteil, dass bei ihnen der Compiler überhaupt nichts prüft, wir aber eine gewisse Prüfung möchten. So soll die Methode `isOnePocketEmpty()` beliebige Taschen entgegennehmen, aber gleichzeitig soll es der Methode auch verboten sein, falsche Dinge in die Taschen zu setzen. Ein `isOnePocketEmpty(Pocket... pockets)` ist also keine gute Idee und führt außerdem zu diversen Warnungen.

Die Lösung besteht im Einsatz des *Wildcard-Typs* `?`. Er repräsentiert dann eine Familie von Typen. Wenn schon `Pocket<Object>` nicht der Basistyp aller Tascheninhalte ist, dann ist es `Pocket<?>`. Es ist wichtig zu verstehen, dass `?` nicht für `Object` steht, sondern für einen unbekannten Typ! Damit lässt sich `isOnePocketEmpty()` realisieren:

Listing 9.15: com/tutego/insel/generic/PocketsEmpty.java

```
public static boolean isOnePocketEmpty( Pocket<?>... pockets )
{
    for ( Pocket<?> pocket : pockets )
        if ( pocket.isEmpty() )
            return true;

    return false;
}

public static void main( String[] args )
{
    Pocket<String> p1 = new Pocket<String>( "Bad-Bank" );
    Pocket<Integer> p2 = new Pocket<Integer>( 1500000 );
    System.out.println( isOnePocketEmpty( p1, p2 ) ); // false
    System.out.println( isOnePocketEmpty( p1, p2, new Pocket<Byte>() ) ); // true
}
```

Dass der Aufruf von `isOnePocketEmpty()`, also bei keiner übergebenen Tasche, zu false führt, soll an dieser Stelle als gegeben gelten.

Wir müssen Wildcards von Typvariablen gedanklich streng trennen. Instanziierungen mit Wildcards sind nicht erlaubt, da ein Wildcard ja eben nicht für einen konkreten Typ, sondern für eine ganze Reihe von möglichen Typen steht. Wildcards können auch nicht wie Typvariablen in Methoden genutzt werden, auch wenn der Typ beliebig ist.

Korrekt mit Typvariable	Falsch mit Wildcard (Compilerfehler!)
Pocket<?> pocket = new Pocket<Byte>();	Pocket<?> pocket = new Pocket<?>();
static <T> T random(T m, T n) { ... }	static ? random(? m, ? n) { ... }

Tabelle 9.8: Möglicher und unmöglicher Einsatz von Wildcard

Auswirkungen auf Lese-/Schreiboperationen

Ist der Wildcard-Typ bei `Pocket<?>` im Einsatz, wissen wir nichts über den Typ, und dem Compiler gehen alle Informationen verloren. Deklarieren wir etwa

`Pocket<?> p1 = new Pocket<Integer>();`

oder

`Pocket<Integer> p2 = new Pocket<Integer>();`
`Pocket<?> p3 = p2;`

dann ist über die wirklichen Typparameter bei `p1` und `p3` nichts bekannt. Das hat wichtige Auswirkungen auf die Methoden, die wir auf `Pocket` aufrufen können.

- Ein Aufruf von `p1.get()` ist legal, denn alles, was die Methode liefern wird, ist immer ein `Object`, auch wenn es `null` ist. Die Anweisung `Object v = p1.get();` ist dementsprechend korrekt.
- Ein `p1.set(value)` kann nicht erlaubt sein, da über den Typ von `value` nichts bekannt ist. In `p1` dürfen wir kein `Double` einsetzen, da `Pocket` nur `Integer` speichern soll. Die einzige Ausnahme ist `null`, da `null` jeden Typ hat. `p1.set(null)` ist also eine zulässige Anweisung. Das heißt ebenso, dass mit `<?>` aufgebaute Objekte nicht automatisch `immutable` sind.

9.5.4 Bounded Wildcards

Die Angabe des konkreten Typparameters wie `Pocket<Integer>` und die Wildcard-Form `Pocket<?>` bilden Extreme. Die Tasche `Pocket<Integer>` nimmt nur Ganzzahlen auf, `Pocket<?>` auf der anderen Seiten alles. Es muss aber auch etwas dazwischen geben, um

zum Beispiel auszudrücken, dass die Tasche nur eine Zahl oder eine Zeichenkette enthalten soll.

Daher sind Typ-Einschränkungen mit `extends` und `super` möglich. Damit ergeben sich drei Arten von Wildcards:

Wildcard	Bezeichnung	Typparameter
<code>?</code>	<i>Wildcard-Typ</i>	ist beliebig
<code>? extends Typ</code>	<i>Upper-bound Wildcard-Typ</i>	muss Typ erweitern
<code>? super Typ</code>	<i>Lower-bound Wildcard-Typ</i>	muss über Typ stehen

Tabelle 9.9: Die drei Wildcard-Typen

Die Wildcard beschreibt also die Eigenschaft eines Typparameters. Wenn es

`Pocket<? extends Number> p;`

heißt, dann können in der Tasche `p` alle möglichen `Number`-Objekte sein.

Machen wir `extends` und `super` noch an einem anderem Beispiel deutlich, das zeigt, welche Familie von Typen die Syntax beschreibt:

<code>? extends CharSequence</code>	<code>? super String</code>
<code>CharSequence</code>	<code>String</code>
<code>String</code>	<code>CharSequence</code>
<code>StringBuffer</code>	<code>Object</code>
<code>StringBuilder</code>	
...	

Tabelle 9.10: Einige eingeschlossene Typen bei `extends` und `super`

Die erste Tabellenzeile (nach dem Tabellenkopf) macht deutlich, dass `extends` und `super` den angegebenen Typ selbst mit einschließen. In `<? extends CharSequence>` ist `CharSequence` genau der Upper-Bound der Wildcard, und in `<? super String>` ist `String` der Lower-Bound der Wildcard. Während die Anzahl der Typen beim Lower-Bound beschränkt ist (die Anzahl der Oberklassen kann sich nicht erweitern), ist die Anzahl der Typen mit Upper-Bound im Prinzip unbekannt, da es immer wieder neue Unterklassen geben kann.

Einsatzgebiete

Jeder der drei Wildcard-Typen hat seine Einsatzgebiete. Weitere Anwendungen der *Upper-bound Wildcard* und der *Lower-bound Wildcard* zeigen die Sortiermethoden der Datenstrukturen und Algorithmen.

Beispiel	Bedeutung
Pocket<?> p;	Taschen mit beliebigem Inhalt
Pocket<? extends Number> p;	Taschen nur mit Zahlen
Comparator<? super String> comp = String.CASE_INSENSITIVE_ORDER;	Entweder String-, Object- oder CharSequence-Comparator. Idee: Ein Comparator, der allgemeine Object-Objekte vergleichen kann, kann (irgendwie) auch Strings vergleichen, denn durch die Vererbung ist ein String eine Art von Object.

Tabelle 9.11: Beispiel für alle drei Wildcard-Typen

Beispiel mit Upper-bound-Wildcard-Typ

Die Upper-bound Wildcard ist häufiger zu finden als die Lower-bound-Variante. Daher wollen wir ein Beispiel aufführen, an dem gut der übliche Einsatz für den Upper-bound abzulesen ist. Unser Player hatte eine rechte und eine linke Tasche. Die Taschen sollen aber nun nicht alles Mögliche speichern können, sondern nur besondere Spielobjekte vom Typ Portable (engl. für *tragbar*). Portable ist eine Schnittstelle, die ein Gewicht für die tragbaren Objekte vorschreibt. Zwei Typen sollen tragbar sein: Pen und Cup. Die Implementierung sieht so aus:

Listing 9.16: com/tutego/insel/generic/PortableDemo.java, Ausschnitt

```
interface Portable
{
    double getWeight();
    void    setWeight( double weight );
}

abstract class AbstractPortable implements Portable
{
    double weight;
```

```

@Override public double getWeight() { return weight; }

@Override public void setWeight( double weight ) { this.weight = weight; }

@Override public String toString() { return getClass().getName() +
    "[weight=" + weight + "]"; }

}

class Pen extends AbstractPortable { }

class Cup extends AbstractPortable { }

```

Um zu testen, ob der Spieler nicht zu viele Sachen trägt, soll eine Methode `areLighterThan()` testen, ob das Gewicht einer Liste von tragbaren Dingen unter einer gegebenen Grenze bleibt. Der erste Versuch könnte so aussehen:

```
boolean areLighterThan( List<Portable> collection, double maxWeight )
```

Moment! Das würde wieder ausschließlich Portable-Objekte akzeptieren, denn Kovarianz gilt ja nicht. Selbst wenn es gehen würde, könnte das bedeuten, dass in einer Methode dann vielleicht über `collection.add()` ein Pen hinzugefügt werden kann, auch wenn die übergebene Liste mit Cup deklariert wurde. Dann stände in der Liste plötzlich etwas Falsches. Außerdem ist Portable eine Schnittstelle, sodass die Methode wirklich überhaupt keinen Sinn ergibt. So ist die korrekte Schreibweise nur mit einem Upper-bound-Wildcard-Typ möglich:

```
boolean areLighterThan( List<? extends Portable> list, double maxWeight )
```

Somit nimmt die Methode nur Listen mit Portable-Objekte an, und das ist auch nötig, denn Portable-Objekte haben ein Gewicht, und diese Eigenschaft brauchen wir.

Listing 9.17: com/tutego/insel/generic/PortableDemo.java, Ausschnitt

```

class PortableUtils
{
    public static boolean areLighterThan( List<? extends Portable> list, double maxWeight )
    {
        double accumulatedWeight = 0.0;

        for ( Portable portable : list )
            accumulatedWeight += portable.getWeight();
    }
}

```

```

        return accumulatedWeight < maxWeight;
    }
}
public class PortableDemo
{
    public static void main( String[] args )
    {
        Pen pen = new Pen();
        pen.setWeight( 10 );
        Cup cup = new Cup();
        cup.setWeight( 100 );
        System.out.println( PortableUtils.areLighterThan( Arrays.asList( pen, cup ),
                                                       10 ) ); //false
        System.out.println( PortableUtils.areLighterThan( Arrays.asList( pen, cup ),
                                                       120 ) ); //true
    }
}

```

Wie schon besprochen wurde, kann aus der mit den Upper-bound-Wildcards deklarierten Datenstruktur `List<? extends Portable>` nur gelesen, aber nicht verändert werden.

9.5.5 Bounded-Wildcard-Typen und Bounded-Typvariablen

Zwischen Bounded-Wildcard-Typen und Bounded-Typvariablen gibt es natürlich einen Zusammenhang, und bei der Deklaration sind zwei Varianten wählbar. Warum das so ist, kann unsere Methode `areLighterThan()` demonstrieren. Statt

```
boolean areLighterThan( List<? extends Portable> list, double maxWeight )
```

hätten wir auch einen formalen Typparameter lokal für die Methode deklarieren können:

```
<T extends Portable> boolean areLighterThan( List<T> list, double maxWeight )
```

Beide Varianten erfüllen den gleichen Zweck. Doch ist die erste Variante der zweiten vorzuziehen.

Best Practice

Immer dann, wenn der formale Typparameter (etwa T) nur in der Signatur auftaucht und es in der Methode selbst keinen Rückgriff auf den Typ T gibt, wähle die Variante mit der Wildcard.

Mit Typparametern lassen sich gut Abhängigkeiten zwischen den einzelnen Argumenten oder dem Rückgabetyp herstellen. Das zeigt das folgende Beispiel (mit einigen Methoden, die bisher noch nicht vorgestellt wurden), das das leichteste Objekt in einer Sammlung von Taschen zurückgeben soll:

Listing 9.18: com/tutego/insel/generic/PortableDemo.java, PortableUtils

```
public static <T extends Portable> T lightest( Collection<T> collection )
{
    Iterator<T> iterator = collection.iterator();
    T lightest = iterator.next();

    while ( iterator.hasNext() )
    {
        T next = iterator.next();

        if ( next.getWeight() < lightest.getWeight() )
            lightest = next;
    }

    return lightest;
}
```

Der Compiler achtet darauf, dass der Typ der Rückgabe mit dem Typ der Sammlung übereinstimmt.

Auf Bounded-Wildcard-Typen in Rückgaben verzichten

Wenn es möglich ist, Bounded-Wildcard-Typen oder Bounded-Typvariablen zu nutzen, sind Bounded-Typvariablen immer vorzuziehen. Wildcard-Typen liefern keine Typinformation, und es ist immer besser, sich vom Compiler über die Typ-Inferenz einen genaueren Typ geben zu lassen.

Nehmen wir eine statische Methode `leftSublist()` an, die von einer Liste eine Unterliste zurückgibt. Die Unterliste geht von der ersten Position bis zur Hälfte.

Versuch 1:

```
public static List<?> leftSublist( List<? extends Portable> list )
{
    return list.subList( 0, list.size() / 2 );
}
```

Der Rückgabetyp `List<?>` ist so ziemlich der schlechteste, den wir wählen können, denn der Aufrüfer der Methode kann mit der Rückgabe überhaupt nichts anfangen: Er weiß nichts über den Inhalt der Liste.

Versuch 2:

```
public static List<? extends Portable> leftSublist( List<? extends Portable> list )
```

Das ist schon ein wenig besser, denn hier bekommt der Empfänger wenigstens die Information zurück, dass die Liste irgendwelche tragbaren Dinge enthält.

Noch besser ist natürlich, auf die Typ-Inferenz des Compilers zu setzen und dem Aufrüfer genau den Typ wieder zurückzugeben, mit dem er den Parametertyp spickte. Dazu müssen wir aber eine Typvariable einsetzen. Der Grund ist: Deklariert eine Methode Parameter oder eine Rückgabe mit mehreren Wildcard-Typen, so sind die wirklichen Typargumente völlig frei wählbar und ohne Zusammenhang.

Listing 9.19: com/tutego/insel/generic/PortableDemo.java, PortableUtils

```
public static <T extends Portable> List<T> rightSublist( List<T> list )
{
    return list.subList( 0, list.size() / 2 );
}
```

Nun ist der Typ der Liste, die reinkommt, gleich dem Typ der Liste, die rauskommt. Mit `extends` ist die Liste zwar nur lesbar, aber das liegt in der Natur der Sache.



Hinweis

Insbesondere in der Klasse `Collections` aus der Java-Standard-API könnten viele Methoden auch anders geschrieben werden. Ein Beispiel: Statt `<T extends E> boolean addAll(Collection<T> c)` wählten die Autoren `boolean addAll(Collection<? extends E> c)`.

9.5.6 Das LESS-Prinzip

Während die mit `extends` eingeschränkten Familien Leseoperationen zulassen, gilt für `super` das Gegenteil. Hier ist Lesen nicht erlaubt, aber Schreiben. Als Merkhilfe lässt sich das als LESS-Prinzip¹⁰ festhalten:

$$\text{Lesen} = \text{Extends}, \text{Schreiben} = \text{Super} \text{ (LESS)}$$

Ein Beispiel ist auch hier hilfreich. Eine statische Methode `copyLighterThan()` soll nur die Elemente aus einer Liste in eine andere kopieren, die leichter als eine bestimmte Obergrenze sind. Der erste Versuch:

```
public static void copyLighterThan( List<? extends Portable> src,
                                    List<? extends Portable> dest, double maxWeight )
{
    for ( Portable portable : src )
        if ( portable.getWeight() < maxWeight )
            dest.add( portable );           // ☹ Compilerfehler !!
}
```

Auf den ersten Blick sieht es gut aus, aber das Programm lässt sich nicht übersetzen. Das Problem ist die Zeile `dest.add(portable);`. Wir erinnern uns: Mit einer Upper-bound Wildcard lässt sich nicht schreiben. Das ergibt Sinn, denn die Liste `src` kann ja zum Beispiel eine Liste von `Cup`-Objekten sein und `dest` eine Liste von `Pen`-Objekten. Beide sind `Portable`, aber dennoch inkompatibel, da `Cups` nicht in `Pens` kopiert werden können. Die Frage ist also, wie der Typ der Ergebnisliste aussehen soll. Beginnen wir bei der Quelliste. Hier ist `List<? extends Portable>` schon korrekt, denn die Liste kann ja alles enthalten, was tragbar ist. Doch welche Anforderungen gibt es an die Zielliste? Wie muss der Typ sein, sodass sich alles vom Typ `Portable`, wie `Cup` oder `Pen`, oder sogar noch Unterklassen speichern lassen? Die Antwort ist einfach: Jeder Typ, der über `Portable` liegt! Das sind `Portable` selbst und `Object`, also alle Obertypen. Dies ist aber der Lower-bound-Wildcard-Typ, den wir mit `super` schreiben. Damit folgt:

Listing 9.20: com/tutego/insel/generic/PortableDemo.java, PortableUtils

```
public static void copyLighterThan( List<? extends Portable> src,
                                    List<? super Portable> dest, double maxWeight )
{
    for ( Portable portable : src )
```

¹⁰ Im Englischen ist auch der Ausdruck PECS (*producer-extends, consumer-super*) im Umlauf.

```

    if ( portable.getWeight() < maxWeight )
        dest.add( portable );
}

```

Ein Beispiel für den Aufruf:

Listing 9.21: com/tutego/insel/generic/PortableDemo.java, Ausschnitt main()

```

List<? extends Portable> src = Arrays.asList( pen, cup );
List<? super Portable> dest = new ArrayList<Object>();
PortableUtils.copyLighterThan( src, dest, 20 );
System.out.println( dest.size() ); // 1
Object result = dest.get( 0 );
System.out.println( result ); // com.tutego.insel.generic.Pen[weight=10.0]

```

Die Liste `dest` ist schreibbar, aber der lesbare Typ ist lediglich `Object` – der Compiler weiß nicht, was hier tatsächlich in der Liste steckt, er weiß nur, dass es beliebige Ober-
typen von `Portable` sein können. Und da bleibt als allgemeinster Typ eben nur `Object`.

Wildcard-Capture

Das LESS-Prinzip hat eine wichtige Konsequenz, die insbesondere bei Listen-Operatio-
nen auffällt. Eine mit einer Wildcard parametrisierte Liste kann nicht verändert werden.
Doch wie lässt sich zum Beispiel eine Methode schreiben, die eine Liste umdreht? Vom
API-Design her könnte eine Methode `reverse()` wie folgt aussehen:

```
public static void reverse( List<?> list );
```

oder so:

```
public static void <T> reverse( List<T> list );
```

Nach unserem Verständnis, dass wir bei völlig freien Typen die Wildcard-Schreibweise
bevorzugen wollen, stehen wir vor einem Dilemma.

```

public static <T> void reverse( List<?> list )
{
    for ( int i = 0; i < list.size() / 2; i++ )
    {
        int j = list.size() - i - 1;
        ? tmp = list.get( i );           // ☹ Compilerfehler
        list.set( i, list.get( j ) );
    }
}

```

```

        list.set( j, tmp );
    }
}

```

Es bleibt uns nichts anderes, als doch die Variante mit der Typvariablen zu wählen, so dass wir Zugriff auf den Typ T haben.

Da nun vom API-Design `reverse(List<?> list)` bevorzugt wird, aber `reverse(List<T> list)` in der Implementierung nötig ist, stellt sich die Frage, ob beides miteinander vereinbar ist. Die gute Nachricht: Ja, denn `reverse(List<?> list)` kann auf eine Umdrehmethode `reverse_(List<T>)` weiterleiten. Zwar müssen die Methoden anders benannt werden, aber wegen des sogenannten *Wildcard-Capture* funktioniert die Abbildung von einer Wildcard auf eine Typvariable.

Listing 9.22: com/tutego/insel/generic/WildcardCapture, WildcardCapture

```

public class WildcardCapture
{
    private static <T> void reverse_( List<T> list )
    {
        for ( int i = 0; i < list.size() / 2; i++ )
        {
            int j = list.size() - i - 1;
            T tmp = list.get( i );
            list.set( i, list.get( j ) );
            list.set( j, tmp );
        }
    }

    public static void reverse( List<?> list )
    {
        reverse_( list );
    }
}

```

Der Compiler »fängt« bei `reverse(list)` den unbekannten Typ der Liste ein und »füllt« die Typvariable bei `reverse_(list)`.

9.5.7 `Enum<E extends Enum<E>>` *

Die generische Deklaration der Klasse `Enum` besitzt eine Besonderheit, die wir uns kurz vornehmen wollen:

```
public abstract class Enum<E extends Enum<E>>
    implements Comparable<E>, Serializable
```

Ein konkreterer parametrisierter Typ muss also die Typvariable `E` so wählen, dass sie einen Untertyp von `Enum` beschreibt.

Das Ganze lässt sich am besten an einem Beispiel erklären. Die Klasse `Enum` ist eine besondere Klasse, die der Compiler immer dann verwendet, wenn er eine enum-Aufzählung umsetzen soll. Angenommen, `Page` deklariert zwei Seitengrößen:

```
public enum Page
{
    A4, A3
}
```

Ohne zu genau auf die Methodenrümpfe zu schauen, generiert der Compiler folgenden Programmcode:

```
public final class Page extends java.lang.Enum<Page>
{
    public static final Page A4 = ...
    public static final Page A3 = ...
    public static Page[] values() { ... }
    public static Page valueOf(String s) { ... }
    ...
}
```

Aus einer enum-Aufzählung entsteht also eine Klasse, die `Enum` erweitert und als parametrisierter Typ genau diese Klasse nennt: `Page` extends `Enum<Page>`. Vergleichen wir das mit der generischen Typdeklaration `Enum<E extends Enum<E>>`, so ist der Typparameter `Page` eine Instanziierung der Typvariable `E`. Und `Page` ist eine Unterklasse von `Enum` (`Page` extends `Enum`), genauso wie die Typvariable `E` das mit dem Typ-Parameter-Bound vorschreibt: `E extends Enum`.

Was wir bisher gesehen haben, zeigt, dass die Deklaration »passt«. Aber warum ist sie so gewählt? Die Typvariable `E` ist so deklariert, dass sie für `Enum`-Unterklassen steht, also für

die konkrete Aufzählung selbst, wie es Page zeigt. Das ist wichtig für Vergleiche. Dazu schauen wir uns einen Ausschnitt aus der Deklaration der abstrakten Klasse `Enum` noch einmal an, und zwar genau die Teile, die etwas mit dem Typ `E` einfordern; das sind zwei Methoden:

Listing 9.23: `java/lang/Enum.java`, Ausschnitt

```
public abstract class Enum<E extends Enum<E>>
    implements Comparable<E>, Serializable
{
    public final int compareTo(E o) {
    public final Class<E> getDeclaringClass() {
        ...
    }
```

Bleiben wir bei der Vergleichsmethode: `compareTo()` ermöglicht es, dass wir zwei Aufzählungen vergleichen und zum Beispiel `A4.compareTo(A3)` schreiben können. Java erlaubt dabei nur, dass zwei Aufzählungen vom gleichen Typ verglichen werden können: Vergleiche der Art `A4.compareTo(Thread.State.NEW)` führen zu einem Compilerfehler. Damit sind wir der Lösung schon nah. Die Deklaration der `compareTo()`-Methode befindet sich in `Enum` und wird den Unterklassen vererbt – die Methode wird nicht vom Compiler magisch in die Unterklassen gesetzt, wie etwa `values()` oder `valueOf()`. Damit bei `compareTo(E o)` jetzt nur eine Unterklasse von `Enum`, nämlich die konkrete Aufzählung, erlaubt ist, fordert `Enum` eben `E extends Enum<E>`.

Die abschließende Frage ist, ob auch eine andere Deklaration für `Enum` möglich gewesen wäre, ohne dass es zu einem Nachteil kommen würde. Die Antwort ist: Ja, im Prinzip ist auch `class Enum<E>` möglich. Auf den ersten Blick scheint das aber falsch zu sein. Spielen wir diese Deklaration statt `Enum<E extends Enum<E>>` kurz durch. Dann könnte ein Entwickler schreiben: `class Page extends Enum<Bunny>` – die geerbte Vergleichsmethode von `Page` hieße dann `compareTo(Bunny o)`, was falsch wäre. Mit der korrekten Deklaration `Enum<E extends Enum<E>>` ist nur ein `class Page extends Enum<Page>` möglich.

Jetzt kommt aber die große Einschränkung: Wir dürfen keine Unterklassen von `Enum` aufbauen, sondern nur der Compiler darf das tun. Ein eigenmächtiger Versuch wird vom Compiler abgestraft. Der unfehlbare Compiler könnte mit einer Deklaration `class Enum<E>` arbeiten, denn er würde für `E` den Aufzählungstyp einsetzen, also Programmcode für `class Page extends Enum<Page>` generieren. So stände in `compareTo()` der richtige Typ, denn `E` wäre mit `Page` instanziert, was zu dem gewollten `compareTo(Page o)` führt. Und auch die in `Enum` deklarierte Methode `getDeclaringClass()` liefert `Page`. Einschränkungen der möglichen Typparameter helfen Entwicklern, Typfehler zu minimieren,

aber der Compiler macht keine Fehler, für ihn ist die Präzisierung nicht nötig. Aber es gibt für die Java-API-Designer keinen Grund, `Enum` schwächer zu deklarieren als nötig. Außerdem gibt es noch einen Unterschied im Bytecode, der sich durch die Typ-Lösung ergibt. Bei `Enum<E>` ist die Umsetzung von `E getDeclaringClass()` im Bytecode nur `Object getDeclaringClass()`, doch mit `Enum<E extends Enum<E>>` ist sie immerhin `Enum getDeclaringClass()`, was besser ist.

9.6 Konsequenzen der Typlösung: Typ-Token, Arrays und Brücken *

Die Typlösung ist im Allgemeinen kein so großes Problem, doch in speziellen Situationen ist es lästig, dass der Typ nicht zur Laufzeit vorliegt.

9.6.1 Typ-Token

Wir haben zum Beispiel gesehen, dass, wenn eine Tasche mit der Typvariablen `T` deklariert wurde, dieses `T` nicht wirklich wie ein Makro durch den Typparameter ersetzt wird, sondern dass in der Regel nur einfach `Object` eingesetzt wird.

```
class Pocket<T>
{
    T newPocketContent() { return new T(); } // ☹ Compilerfehler
}
```

Aus `new T()` macht die Typlösung also `new Object()`, und das ist nichts wert. Doch wie kann dennoch ein Typ erzeugt werden und der Typ `T` zur Laufzeit vorliegen?

Hier lässt es sich ein Trick nutzen, nämlich ein `Class`-Objekt für den Typ einsetzen.

Typparameter	Class-Objekt repräsentiert Typparameter
<code>String</code>	<code>Class.String</code>
<code>Integer</code>	<code>Class.Integer</code>

Tabelle 9.12: Transfer der Typparameter durch Class-Objekte

Dieses `Class`-Objekt, das nun den Typ repräsentiert, heißt *Typ-Token* (engl. *type token*). Es kommt natürlich entgegen, dass `Class` selbst als generischer Typ deklariert ist und zwei interessante Methoden ebenfalls generifiziert wurden:

```
Class<String> clazz1 = String.class;
String newInstance = clazz1.newInstance();
Class< ? extends String> clazz2 = newInstance.getClass();
System.out.println( clazz1.equals( clazz2 ) ); // true
```

Zunächst ist da die Methode `newInstance()`. Sie erzeugt ein neues Exemplar mit Typ, den das `Class`-Objekt repräsentiert.

```
final class java.lang.Class<T>
    implements Serializable, GenericDeclaration, Type, AnnotatedElement
{
    public T newInstance()
        throws InstantiationException, IllegalAccessException
```

Mit einem gegebenen Objekt lässt sich mit `getClass()` das zugehörige `Class`-Objekt zur Klasse erfragen.

```
class java.lang.Object
{
    public final native Class<?> getClass()
        Liefert Class-Objekt.
```

Hinweis

Die Rückgabe `Class<?>` bei `getClass()` ist unschön, insbesondere die allgemeine Wildcard. Sie verhindert, dass sich Folgendes schreiben lässt:

```
Class<String> clazz = "ARTE".getClass(); // ☹ Type mismatch Compilerfehler
```

Stattdessen muss es so heißen:

```
Class<? extends String> clazz = "ARTE".getClass();
```

Da `Object` nicht generisch deklariert ist, ist es kein Wunder, dass `getClass()` keine genaueren Angaben machen kann.



Lösungen mit dem Typ-Token

Um das Typ-Token einzusetzen, muss das `Class`-Object mit als Argument in einem Konstruktor oder einer Methode übergeben werden. So lässt sich etwa eine `newInstance()`-Methode nachbauen, die die geprüften Exceptions fängt und im Fehlerfall als `Runtime-Exception` meldet. Gut zu sehen ist, wie sich der Typ des `Class`-Objekts auf die Rückgabe überträgt.

```

public static <T> T newInstance( Class<T> type )
{
    try
    {
        return type.newInstance();
    }
    catch ( /*InstantiationException, IllegalAccessException*/ Exception e )
    {
        throw new RuntimeException( e );
    }
}

```

PS: Seit Java 7 ist `ReflectiveOperationException` die Basisklasse für Reflection-Fehler.

9.6.2 Super-Type-Token

Mit einem `Class`-Objekt lässt sich gut ein Typ repräsentieren, allerdings gibt es ein Problem. Das `Class`-Objekt kann selbst keine generischen Typen darstellen.

Typparameter	Class-Objekt repräsentiert Typparameter
<code>String</code>	<code>Class.String</code>
<code>Integer</code>	<code>Class.Integer</code>
<code>Pocket<String></code>	<code>Class.Pocket<String></code> Geht nicht!

Tabelle 9.13: Ein `Class`-Objekt kann keinen generischen Typ beschreiben.

Der wirkliche Typ lässt sich nur mit viel Getrickse bestimmen und festhalten. Hier kommt die Reflection-API zum Einsatz, sodass nur kurz die Klasse, ein Beispiel und Verweise auf weitere Literatur vorgestellt werden sollen. Hier die Klasse:

Listing 9.24: `com/tutego/insel/generic/TypeRef, TypeRef`

```

public abstract class TypeRef<T>
{
    public final Type type;

    protected TypeRef()
    {

```

```

ParameterizedType superclass = (ParameterizedType) getClass().getGenericSuperclass();
type = superclass.getActualTypeArguments()[0];
}
}

```

Und ein Beispiel, das eine anonyme Unterklasse erzeugt und so den Typ zugänglich macht:

Listing 9.25: com/tutego/insel/generic/TypeRefDemo, main()

```

TypeRef<Pocket<String>> ref1 = new TypeRef<Pocket<String>>() {};
System.out.println( ref1.type ); // com.tutego.insel.generic.Pocket<java.lang.String>
TypeRef<Pocket<Byte>> ref2 = new TypeRef<Pocket<Byte>>() {};
System.out.println( ref2.type ); // com.tutego.insel.generic.Pocket<java.lang.Byte>

```

Damit konnten wir den Typparameter über `java.lang.reflect.Type` festhalten, und `ref1` unterscheidet sich eindeutig von `ref2`. Der Typ liegt jedoch nicht als `Class`-Objekt vor, und Operationen wie `newInstance()` sind auf `Type` nicht möglich – die Schnittstelle deklariert überhaupt keine Methoden, sondern repräsentiert nur Typen.

Auf den Webseiten <http://gaffer.blogspot.com/2006/12/super-type-tokens.html> und <http://www.artima.com/weblogs/viewpost.jsp?thread=206350> sind weitere Informationen und Einsatzgebiete zu finden. Super-Typ-Token kommen in der Java-Welt nicht besonders oft vor.

9.6.3 Generics und Arrays

Die Typlösung ist der Grund dafür, dass Arrays nicht so umgesetzt werden können, wie es sich der Entwickler denkt.¹¹ Folgendes ergibt einen Compilerfehler:

```

class TwoBox<T>
{
    private T[] array = new T[ 2 ]; // ☹ Cannot create a generic array of T
    T[] getArray() { return array; }
}

```

¹¹ Bei Sun (http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4888066) ist dafür ein Bug gelistet. Sungs Antwort auf die Bitte, ihn zu beheben, lautet lapidar: »Some day, perhaps, but not now.«

Der Grund für diesen Fehler ist dann gut zu erkennen, wenn wir überlegen, zu welchem Programmcode die Typlösung führen würde:

```
class TwoBox
{
    Object[] array = new Object[ 2 ];           // (1)
    Object[] getArray() { return array; }
}
```

Der Aufrufer würde nun die `TwoBox` parametrisiert verwenden wollen:

```
TwoBox<String> twoStrings = new TwoBox<String>();
String[] stringArray = twoStrings.getArray();
```

Denken wir an dieser Stelle wieder an die Typlösung und an das, was der Compiler generiert:

```
TwoBox twoStrings = new TwoBox();
String[] stringArray = (String[]) twoStrings.getArray(); // (2)
```

Jetzt ist es auffällig: Während (1) ein `Object`-Array der Länge 2 aufbaut und auch `getArray()` dies als `Object`-Array nach außen gibt, castet (2) dieses `Object`-Array auf ein `String`-Array. Das geht aber nicht, denn diese beiden Typen sind nicht typkompatibel. Zwar kann natürlich ein `Object`-Array `String`s referenzieren, aber das Feld selbst als Objekt ist eben ein `Object[]` und kein `String[]`.

Reflection hilft

Die Java-API bietet über Reflection wieder eine Möglichkeit, Arrays eines Typs zu erzeugen:

```
T[] array = (T[]) Array.newInstance( clazz, 2 );
```

Allerdings muss der Class-Typ `clazz` bekannt sein und als zusätzlicher Parameter übergeben werden. Die Syntax `T.class` gibt einen Compilerfehler, denn über die Typlösung wäre das ja sowieso immer `Object.class`, was den gleichen Fehler wie vorher ergibt und kein Fortschritt wäre.

9.6.4 Brückenmethoden

Aus der Tatsache, dass mit Generics übersetzte Klassen auf einer JVM ablauffähig sein müssen, die kein generisches Typsystem besitzt, folgen diverse Hacks, die der Compiler zur Erhaltung der heiligen Kompatibilität vornimmt. Er fügt neue Methoden ein, sogenannte *Brückenmethoden*, damit der Bytecode nach der Typlösung auch von älteren Programmen genutzt werden kann.

Brückenmethode wegen Typvariablen in Parametern

Starten wir mit der Schnittstelle Comparable, die generisch deklariert wurde.

```
public interface Comparable<T> { public int compareTo( T o ); }
```

Die bekannte Klasse Integer implementiert zum Beispiel diese Schnittstelle und kann somit sagen, wie die Ordnung zu einem anderen Integer-Objekt ist:

Listing 9.26: java/lang/Integer.java, Ausschnitt

```
public final class Integer extends Number implements Comparable<Integer>
{
    private final int value;

    public Integer( int value ) { this.value = value; }

    public int compareTo( Integer anotherInteger )
    {
        int thisVal = this.value;
        int anotherVal = anotherInteger.value;
        return ( thisVal < anotherVal ? -1 : (thisVal == anotherVal ? 0 : 1) );
    }
    ...
}
```

Integer implementiert die Methode compareTo() mit dem Parametertyp Integer. Der Compiler wird also eine Methode mit der Signatur compareTo(Integer) erstellen. Doch damit beginnt ein Problem! Wir haben eine unbekannte Anzahl an Zeilen Quellcode, die sich auf eine Methode compareTo(Object) beziehen, denn vor Java 5 war die Signatur ja anders.

Damit es nicht zu Inkompatibilitäten kommt, setzt der Compiler einfach noch die Methode `compareTo(Object)` bei `Integer` dazu. Die Implementierung sieht so aus, dass sie einfach delegiert:

```
public int compareTo( Object anotherInteger )
{
    return compareTo( (Integer) anotherInteger );
}
```

Brückenmethode wegen kovarianten Rückgabetypen

Wenn eine Methode überschrieben wird, so muss die Unterklasse die gleiche Signatur (also den gleichen Methodennamen und die gleiche Parameterliste) besitzen. Nehmen wir eine Klasse `CloneableFont`, die `Font` erweitert und die `clone()`-Methode aus `Object` über-schreibt. Eine Klasse, die sich auch unter Java 1.4 übersetzen lässt, würde so aussehen:

Listing 9.27: com/tutego/insel/nongeneric/CloneableFont.java, CloneableFont

```
public class CloneableFont extends Font implements Cloneable
{
    public CloneableFont( String name, int style, int size )
    {
        super( name, style, size );
    }

    @Override public Object clone()
    {
        return new Font( getAttributes() );
    }
}
```

Im Bytecode der Klasse `CloneableFont` sind somit ein Konstruktor und eine Methode vermerkt.

Dazu kurz ein Blick auf die Ausgabe des Dienstprogramms `javap`, das die Signaturen anzeigt:

```
$ javap com.tutego.insel.nongeneric.CloneableFont
Compiled from "CloneableFont.java"
```

```
public class com.tutego.insel.nongeneric.CloneableFont extends java.awt.Font{
    public com.tutego.insel.nongeneric.CloneableFont(java.lang.String, int, int);
    public java.lang.Object clone();
}
```

Nehmen wir nun an, eine zweite Klasse ist Nutzer von `CloneableFont`:

Listing 9.28: com/tutego/insel/nongeneric/CloneableFontDemo.java, main()

```
CloneableFont font = new CloneableFont( "Arial", Font.BOLD, 12 );
Object font2 = font.clone();
```

Beim Aufruf `font.clone()` prüft der Compiler, ob die Methode `clone()` in `CloneableFont` aufrufbar ist, und trägt dann die exakte Signatur mit Rückgabe – das ist der entscheidende Punkt – im Bytecode ein. Die Anweisung `font.clone()` sieht im Bytecode von `CloneableFontDemo.class` etwa so aus (disassembliert mit *javap*):

```
invokevirtual #23;
```

Die #23 ist ein Verweis auf die `clone()`-Methode von `CloneableFont`, und `invokevirtual` ist der Bytecodebefehl zum Aufruf der Methode. Hinter der 23 steckt eine JVM-Methodenkennung, die von *javap* so ausgegeben wird:

```
com/tutego/insel/nongeneric/CloneableFont.clone:()Ljava/lang/Object;
```

Im Bytecode steht exakt ein Verweis auf die Methode `clone()` mir dem Rückgabetyp `Object`.

Seit Java 5 ist eine kleine Änderung beim Überschreiben hinzugekommen. Wenn eine Unterklasse eine Methode überschreibt, kann sie den Rückgabetyp auf einen Untertyp präzisieren – das nennt sich *kovariantes Überschreiben*. Also kann `clone()` statt `Object` jetzt `Font` zurückgeben.

Gleicher Rückgabetyp wie die überschriebene Methode	Konvarianter Rückgabetyp, seit Java 5
Listing 9.29: com/tutego/insel/nongeneric/CloneableFont.java, <code>clone()</code> <pre>@Override public Object clone() { return new Font(getAttributes()); }</pre>	Listing 9.30: com/tutego/insel/generic/CloneableFont.java, <code>clone()</code> <pre>@Override public Font clone() { return new Font(getAttributes()); }</pre>

Tabelle 9.14: Beispiel kovarianter Rückgabetypen

Da der Rückgabetyp der überschriebenen Methode nun nicht mehr `Object`, sondern `Font` ist, ändert sich auch der Bytecode von `CloneableFont`. Die Datei `CloneableFont.class` ist ohne konvariante Rückgabe 593 Byte groß und mit konvarianter Rückgabe 739 Byte. (Warum dieser satte Unterschied? Dazu später mehr.)

Stellen wir uns nun Folgendes vor: Die erste Version von `CloneableFont` wurde lange vor Java 5 implementiert und konnte so noch kein kovariantes Überschreiben nutzen. Die Klasse `CloneableFont` ist unglaublich populär, und die Methode `clone()` – die `Object` liefert – wird von vielen Stellen aufgerufen. Im Bytecode der nutzenden Klassen gibt es also immer einen Bezug auf die Methode mit der JVM-Signatur:

```
com/tutego/insel/nongeneric/CloneableFont.clone:()Ljava/lang/Object;
```

Bei einem Refactoring geht der Autor der Klasse `CloneableFont` über seine Klasse und sieht, dass er bei `clone()` die konvariante Rückgabe nutzen kann. Er korrigiert die Methode und setzt statt `Object` den Typ `Font` ein. Er kompiliert die Klasse und setzt sie wieder in die Öffentlichkeit.

Nun stellt sich die Frage, was mit den Nutzern ist, also Klassen wie `CloneableFontDemo`, die *nicht* neu übersetzt werden. Denn sie suchen eine Methode mit dieser JVM-Signatur:

```
com/tutego/insel/nongeneric/CloneableFont.clone:()Ljava/lang/Object;
```

Da `clone()` in `CloneableFont` in der aktuell Version nun `Font` zurückgibt, müsste die JVM einen Fehler auslösen, denn der Bytecode ist ja anders, und die gefragte Methode mit der Rückgabe `Object` ist nicht mehr da. Das wäre ein riesiges Problem, denn so würden durch die Änderung des Autors alle nutzenden Klassen illegal, und die Projekte mit diesen Klassen ließen sich alle nicht mehr übersetzen.

Doch es gibt kein Anlass zur Panik. Es kommt nicht zu einem Compilerfehler, da der Compiler eine Hilfsmethode einfügt, die in der JVM-Signatur mit der Java 1.4-Variante von `clone()`, also mit der Rückgabe `Object`, übereinstimmt. Der Disassembler `javap` zeigt das gut:

```
$ javap com.tutego.insel.generic.CloneableFont
Compiled from "CloneableFont.java"
public class com.tutego.insel.generic.CloneableFont extends java.awt.Font{
    public com.tutego.insel.generic.CloneableFont(java.lang.String, int, int);
    public java.awt.Font clone();
    public java.lang.Object clone() throws java.lang.CloneNotSupportedException;
}
```

Die `clone()`-Methode gibt es also zweimal! Interessant ist die Besonderheit, dass Dinge im Bytecode erlaubt sind, die im Java-Programmcode nicht möglich sind. In Java gehört der Rückgabetyp nicht zur Signatur, und der Java-Compiler erlaubt nicht zwei Methoden mit der gleichen Signatur. Im Bytecode allerdings gehört der Rückgabetyp schon dazu, und daher sind die Methoden dort erlaubt, da sie klar unterscheidbar sind.

Übersetzt ein Java 5-Compiler die Klasse `CloneableFont` mit dem kovarianten Rückgabetyp bei `clone()`, so setzt er automatisch die Brückenmethode ein. Das erklärt auch, warum der Bytecode der Klassen mit konvarianten Rückgabetypen größer ist. So finden auch die alten Klassen, die auf die ursprüngliche `clone()`-Methode mit der Rückgabe `Object` kompiliert sind, diese Methode, und es gibt keinen Laufzeitfehler.

Als Letztes muss noch geklärt werden, was der Compiler eigentlich genau für eine Brückenmethode generiert. Das ist einfach, denn in die Brückenmethode setzt der Compiler eine Weiterleitung.

```
@Override public Font clone()
{
    return new Font( getAttributes() );
}
@Override public Object clone()
{
    return (Font) clone();
}
```

Bleibt festzuhalten, dass auf Ebene der JVM kovariante Rückgabetypen nicht möglich sind und im Bytecode immer die Methode inklusive Rückgabetyp referenziert wird.

Hinweis



In Java sind alle Methoden einer Klasse, die sich auch im Bytecode befinden, über Reflection zugänglich. Lästig wäre es nun, wenn Tools Methoden sehen, die vom Compiler eingeführt werden, weil dieser herumtricksen und Beschränkungen umschiffen muss. Die Brückenmethoden werden daher mit einem speziellen Flag markiert und als *synthetische Methoden* (engl. *synthetic method*) gekennzeichnet. Das Flag lässt sich über Reflection mit `isSynthetic()` an den Field-Objekten erfragen.

Brückenmethode wegen einer Typvariable in der Rückgabe

Werfen wir einen Blick auf ein ähnliches Szenario, bei dem der Rückgabetyp durch eine Typvariable einer generisch deklarierten Klasse bestimmt wird, und sehen wir uns an, welche Konsequenzen sich im Bytecode daraus ergeben.

Die Schnittstelle `Iterator` dient im Wesentlichen dazu, Datensammlungen nach ihren Daten zu fragen. Die beiden Spalten zeigen die Deklaration der `Iterator`-Schnittstelle unter Java 1.4 und Java 5.

Java 1.4	Java 5
<pre>public interface Iterator { boolean hasNext(); Object next(); void remove(); }</pre>	<pre>public interface Iterator<E> { boolean hasNext(); E next(); void remove(); }</pre>

Tabelle 9.15: Veränderung der Implementierung der Schnittstelle `Iterator`

So soll `hasNext()` immer `true` ergeben, wenn der `Iterator` weitere Daten liefern kann, und `next()` liefert schlussendlich das Datum selbst. In der Variante unter Java 5 ist der Rückgabetyp von `next()` durch die Typvariable bestimmt.

Warum Brückenmethoden benötigt werden, zeigt wieder ein Beispiel, in dem Entwickler mit Java 1.4 begannen und später ihr Programm mit Generics verfeinern. Trotz der Änderung muss eine alte, mit Java 1.4 compilierte Version noch funktionieren.

Üblicherweise liefern Iteratoren alle Elemente einer Datenstruktur. Doch anstatt durch eine Datenstruktur zu laufen, soll unser `Iterator`, ein `EndlessRandomIterator`, unendlich viele Zufallszahlen liefern. Wir interessieren uns besonders für die Implementierung der Schnittstelle `Iterator`. Ohne Generics unter Java 1.4 sieht das so aus:

Listing 9.31: com/tutego/insel/nongeneric/EndlessRandomIterator.java, `EndlessRandomIterator`

```
public class EndlessRandomIterator implements Iterator
{
    public boolean hasNext() { return true; }
    public Object next() { return Double.valueOf( Math.random() ); }
    public void remove() { throw new UnsupportedOperationException(); }
}
```

Ein Programm, das den `EndlessRandomIterator` nutzt, empfängt bei `next()` nun ein `Double`. Aber durch den Ausschluss der Konvarianz in Java 1.4 kann durch die Deklaration von `Object next()` in `Iterator` in unserer Klasse `EndlessRandomIterator` auch nur `Object next()` stehen. Ein Nutzer von `next()` wird also auf jeden Fall die Methode `next()` mit dem Rückgabetyp `Object` erwarten und so im Bytecode vermerken – die Begründung hatten wir schon bei der Konvarianz im vorangehenden Unterkapitel aufgeführt.

Wird unter Java 5 der `EndlessRandomIterator` überarbeitet, ändern sich drei Dinge. Erstens wird `implements Iterator` zu `implements Iterator<Double>`. Dann wird `Object next()` zu `Double next()`, und drittens kann es im Rumpf von `next()` durch das Autoboxing etwas kürzer werden, nämlich `return Math.random()`. Der wichtige Punkt ist aber, dass sich der Rückgabetyp von `next()` ändert, also von `Object` auf `Double` geht. An der Stelle muss aber der Compiler mit einer Brückenmethode eingreifen, sodass im Bytecode wieder zwei `next()`-Methoden stehen: Einmal `Object next()` und dann `Double next()`. Denn ohne `Double next()` würde der alte Programmcode, der `Object next()` erwartet, plötzlich nicht mehr laufen, und das wäre ein Bruch zu der Abwärtskompatibilität.



Kapitel 10

Architektur, Design und angewandte Objektorientierung

»Eines der traurigsten Dinge im Leben ist, dass ein Mensch viele gute Taten tun muss, um zu beweisen, dass er tüchtig ist, aber nur einen Fehler zu begehen braucht, um zu beweisen, dass er nichts taugt.«
– George Bernard Shaw (1856–1950)

10

Während die Beispiele aus den Kapiteln 2 bis 9 im Wesentlichen in einer kleinen `main()`-Methode Platz fanden, sind die Beispiele aus diesem Kapitel etwas umfangreicher und zeigen das objektorientierte Zusammenspiel von mehreren Klassen.

10.1 Architektur, Design und Implementierung

Von dem Wunsch des Auftraggebers hin zur fertigen Software ist es ein weiter Weg. Dazwischen liegen Anforderungsdokumente, Testfälle, die Auswahl der Infrastruktur, die Wahl von Datenbanken, Lizenzfragen, menschliche Eitelkeiten und vieles, vieles mehr. Da die Insel die Softwareentwicklung in den Mittelpunkt rückt, wollen wir uns bevorzugt in den Bereichen Architektur, Design und Implementierung aufhalten.

Der Begriff *Software-Architektur* ist wenig klar umrissen, doch meint er das große Ganze, also die fundamentalen Entscheidungen beim Aufbau der Software. Am besten lässt sich Architektur vielleicht bissig als das charakterisieren, was aufwändig und teuer zu ändern ist, wenn es einmal steht. Ein bekanntes Architekturmuster ist das Schichtenmodell, das Software in mehrere Ebenen gliedert. Die obere Schicht kann dabei nur auf Dienste der direkt unter ihr liegenden Schicht zurückgreifen, aber keine Schicht überspringen, und die untere Schicht hat keine Idee von höher liegenden Schichten.

Im Design kümmern sich die Entwickler um die Abbildung der Ideen auf Pakete, Klassen und Schnittstellen. Diese Abbildung ist nicht eindeutig, und so gibt es viele Möglichkei-

ten; einige sind schlecht, weil sie die Lesbarkeit, Erweiterbarkeit oder Performance beeinträchtigen, andere sind besser. Es kommt aber immer auf den Kontext an. Wir wollen uns daher mit einigen Abbildungen beschäftigen und diese diskutieren.

Die Implementierung setzt das Design (das sich auf der Ebene von UML-Modellen befindet) in den Quellcode einer Programmiersprache um. Die Implementierung vereint das statische Modell, also welche Klasse von welcher erbt, und bringt sie mit der nötigen Dynamik zusammen. Fragen der Implementierung werden immer wieder diskutiert, wenn es zum Beispiel um die Wahl der richtigen Datenstruktur oder Realisierung der Nebenläufigkeit geht.

10.2 Design-Pattern (Entwurfsmuster)

Aus dem objektorientierten Design haben wir gelernt, dass Klassen nicht fest miteinander verzahnt, sondern lose gekoppelt sein sollen. Das bedeutet, Klassen sollten nicht zu viel über andere Klassen wissen, und die Interaktion soll über wohldefinierte Schnittstellen erfolgen, sodass die Klassen später noch verändert werden können. Die lose Kopplung hat viele Vorteile, da so die Wiederverwendung erhöht und das Programm änderungsfreundlicher wird. Wir wollen dies an einem Beispiel prüfen.

In einer Datenstruktur sollen Kundendaten gespeichert werden. Zu dieser Datenquelle gibt es eine grafische Oberfläche, die diese Daten anzeigt und verwaltet, etwa eine Eingabemaske. Wenn Daten eingegeben, gelöscht und verändert werden, sollen sie in die Datenstruktur übernommen werden. Den anderen Weg von der Datenstruktur in die Visualisierung werden wir gleich beleuchten. Bereits jetzt haben wir eine Verbindung zwischen Eingabemaske und Datenstruktur, und wir müssen aufpassen, dass wir uns im Design nicht verzetteln, denn vermutlich läuft die Programmierung darauf hinaus, dass beide fest miteinander verbunden sind. Wahrscheinlich wird die grafische Oberfläche irgendwie über die Datenstruktur Bescheid wissen, und bei jeder Änderung in der Eingabemaske werden direkt Methoden der konkreten Datenstruktur aufgerufen. Das wollen wir vermeiden. Genauso haben wir nicht bedacht, was passiert, wenn nun infolge weiterer Programmversionen eine grafische Repräsentation der Daten etwa in Form eines Balkendiagramms gezeichnet wird. Und was geschieht, wenn der Inhalt der Datenstruktur über eine andere Programmstelle geändert wird und dann einen Neuaufbau der Bildschirmsdarstellung erzwingt? Hier verfangen wir uns in einem Knäuel von Methodenaufrufen, und änderungsfreundlich ist dies dann auch nicht mehr. Was ist, wenn wir nun unsere selbst gestrickte Datenstruktur durch eine SQL-Datenbank ersetzen wollen?

10.2.1 Motivation für Design-Pattern

Wir sind nicht die Ersten, die sich über grundlegende Design-Kriterien Gedanken machen. Vor dem objektorientierten Programmieren (OOP) gab es das strukturierte Programmieren, und die Entwickler waren froh, mit Werkzeugen schneller und einfacher Software bauen zu können. Auch die Assembler-Programmierer waren erfreut, strukturiertes Programmieren zur Effizienzsteigerung einsetzen zu können – sie haben ja auch Unterprogramme nur deswegen eingesetzt, weil sich mit ihnen wieder ein paar Bytes sparen ließen. Doch nach Assembler und dem strukturierten Programmieren sind wir nun bei der Objektorientierung angelangt, und dahinter zeichnet sich bisher kein revolutionäres Programmierparadigma ab. Die Softwarekrise hat zu neuen Konzepten geführt, doch merkt fast jedes Entwicklungsteam, dass OO nicht alles ist, sondern nur ein verwunderter Ausspruch nach drei Jahren Entwicklungsarbeit an einem schönen Finanzprogramm: »Oh, oh, alles Mist.« So schön OO auch ist, wenn sich 10.000 Klassen im Klassendiagramm tummeln, ist das genauso unübersichtlich wie ein FORTRAN-Programm mit 10.000 Zeilen. Da in der Vergangenheit oft gutes Design für ein paar Millisekunden Laufzeit geopfert wurde, ist es nicht verwunderlich, dass Programme nicht mehr lesbar sind. Doch wie am Beispiel des Satzprogramms TeX (etwa 1985) zu sehen ist: Code lebt länger als Hardware, und die nächste Generation von Mehrkernprozessoren wird sich bald in unseren Desktop-PCs nach Arbeit sehnen.

Es fehlt demnach eine Ebene über den einzelnen Klassen und Objekten, denn die Objekte selbst sind nicht das Problem, vielmehr ist es die Kopplung. Hier helfen Regeln weiter, die unter dem Stichwort *Entwurfsmuster* (engl. *design patterns*) bekannt geworden sind. Dies sind Tipps von Softwaredesignern, denen aufgefallen war, dass viele Probleme auf ähnliche Weise gelöst werden können. Sie haben daher Regelwerke mit Lösungsmustern aufgestellt, die eine optimale Wiederverwendung von Bausteinen und Änderungsfreundlichkeit aufweisen. Design-Patterns ziehen sich durch die ganze Java-Klassenbibliothek, und die bekanntesten sind das Beobachter-(Observer-)Pattern, Singleton, Fabrik (Factory) und Composite; die Fabrik und das Singleton haben wir bereits kennengelernt.

10.2.2 Das Beobachter-Pattern (Observer/Observable)

Wir wollen uns nun mit dem Observer-Pattern beschäftigen, das seine Ursprünge in Smalltalk-80 hat. Dort ist es etwas erweitert unter dem Namen MVC (Model-View-Controller) bekannt, ein Kürzel, mit dem auch wir uns noch näher beschäftigen müssen, da dies ein ganz wesentliches Konzept bei der Programmierung grafischer Bedieneroberflächen mit Swing ist.

Stellen wir uns eine Party mit einer netten Gesellschaft vor. Hier finden sich zurückhaltende, passive Gäste und aktive Erzähler. Die Zuhörer sind interessiert an den Gesprächen der Unterhalter. Da die Erzähler nun von den Zuhörern beobachtet werden, bekommen sie den Namen *Beobachtete*, auf Englisch auch *observables* (Beobachtbare) genannt. Die Erzähler interessieren sich jedoch nicht dafür, wer ihnen zuhört. Für sie sind alle Zuhörer gleich. Sie schweigen aber, wenn ihnen überhaupt niemand zuhört. Die Zuhörer reagieren auf Witze der Unterhalter und werden dadurch zu *Beobachtern* (engl. *observers*).

Die Klasse Observable und die Schnittstelle Observer

Unser Beispiel mit den Erzählern und Zuhörern können wir auf Datenstrukturen übertragen. Die Datenstruktur lässt sich beobachten und wird zum Beobachteten. Sie wird in Java als Exemplar der Bibliotheksklasse `Observable` repräsentiert. Der Beobachter wird durch die Schnittstelle `Observer` abgedeckt und ist der, der informiert werden will, wenn sich die Datenstruktur ändert. Jedes Exemplar der `Observable`-Klasse informiert nun alle seine Hörer, sobald sich sein Zustand ändert. Denken wir wieder an unser ursprüngliches Beispiel mit der Visualisierung. Wenn wir nun zwei Ansichten der Datenstruktur haben, etwa die Eingabemaske und ein Balkendiagramm, dann ist es der Datenstruktur egal, wer an den Änderungen interessiert ist. Ein anderes Beispiel: Die Datenstruktur enthält einen Wert, der durch einen Schieberegler und ein Textfeld angezeigt wird. Beide Bedienelemente wollen informiert werden, wenn sich dieser Wert ändert. Es gibt viele Beispiele für diese Konstellation, sodass die Java-Entwickler die Klasse `Observable` und die Schnittstelle `Observer` mit in die Standardbibliothek aufgenommen haben. Noch besser wäre die Entscheidung gewesen, die Funktionalität in die oberste Klasse `Object` aufzunehmen, so wie es Smalltalk macht.

Die Klasse Observable

Eine Klasse, deren Exemplare sich beobachten lassen, muss jede Änderung des Objektzustands nach außen hin mitteilen. Dazu bietet die Klasse `Observable` die Methoden `setChanged()` und `notifyObservers()` an. Mit `setChanged()` wird die Änderung angekündigt, und mit `notifyObservers()` wird sie tatsächlich übermittelt. Gibt es keine Änderung, so wird `notifyObservers()` auch niemanden benachrichtigen.

Wir wollen nun das Party-Szenario in Java implementieren. Dazu schreiben wir eine Klasse `JokeTeller`, deren Objekte einen Witz erzählen können. Sie machen mit `setChanged()` auf eine Änderung ihres Zustands aufmerksam und versorgen dann mit `notifyObservers()` die Zuhörer mit dem Witz in Form einer Zeichenkette:

Listing 10.1: com/tutego/insel/pattern/observer/JokeTeller.java

```
package com.tutego.insel.ds.observer;

import java.util.*;

class JokeTeller extends Observable
{
    private static final List<String> jokes = Arrays.asList(
        "Sorry, aber du siehst so aus, wie ich mich fühle.",
        "Eine Null kann ein bestehendes Problem verzehnfachen.",
        "Wer zuletzt lacht, hat es nicht eher begriffen.",
        "Wer zuletzt lacht, stirbt wenigstens fröhlich.",
        "Unsere Luft hat einen Vorteil: Man sieht, was man einatmet."
    );

    public void tellJoke()
    {
        setChanged();
        Collections.shuffle( jokes );
        notifyObservers( jokes.get(0) );
    }
}
```

`setChanged()` setzt intern ein Flag, das von `notifyObservers()` abgefragt wird. Nach dem Aufruf von `notifyObservers()` wird dieses Flag wieder gelöscht. Dies kann auch manuell mit `clearChanged()` geschehen. `notifyObservers()` sendet nur dann eine Benachrichtigung an die Zuhörer, wenn auch das Flag gesetzt ist. So kommen folgende Programmzeilen häufig zusammen vor, da sie das Flag setzen und alle Zuhörer informieren:

```
setChanged();           // Eine Änderung ist aufgetreten
notifyObservers( Object ); // Informiere Observer über Änderung
```

Die `notifyObservers()`-Methode existiert auch ohne extra Parameter. Sie entspricht einem `notifyObservers(null)`. Mit der Methode `hasChanged()` können wir herausfinden, ob das Flag der Änderung gesetzt ist.

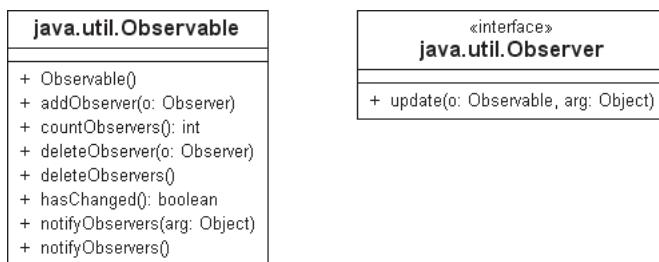


Abbildung 10.1: Klassendiagramm für Observable und Observer

Interessierte Beobachter müssen sich am `Observable`-Objekt mit der Methode `addObserver(Observer)` anmelden. Dabei sind aber keine beliebigen Objekte als Beobachter erlaubt, sondern nur solche, die die Schnittstelle `Observer` implementieren. Sie können sich mit `deleteObserver(Observer)` wieder abmelden. Die Anzahl der angemeldeten `Observer` teilt uns `countObservers()` mit. Leider ist die Namensgebung etwas unglücklich, da Klassen mit der Endung »able« eigentlich immer Schnittstellen sein sollten. Genau das ist hier aber nicht der Fall. Der Name `Observer` bezeichnet überraschenderweise eine Schnittstelle, und hinter dem Namen `Observable` verbirgt sich eine echte Klasse.

- ```

class java.util.Observable

```
- `void addObserver(Observer o)`  
Fügt einen `Observer` hinzu. Das Argument darf nicht `null` sein.
  - `int countObservers()`  
Liefert die Anzahl angemeldeter `Observer`.
  - `void deleteObserver(Observer o)`  
Entfernt den `Observer` `o` wieder.
  - `void deleteObservers()`  
Löscht alle angemeldeten `Observer`.
  - `void setChanged()`  
Markiert das Objekt als sendebereit, sodass bei `notifyObservers()` Meldungen gegeben werden.
  - `void clearChanged()`  
Setzt den Zustand zurück, sodass bei `notifyObservers()` keine Meldungen gegeben werden.
  - `boolean hasChanged()`  
Liefert `true`, wenn das Objekt im Meldemodus ist.

- void notifyObservers(Object arg)  
Liefert hasChanged() wahr, dann informiert das Observable alle Observer und übergibt arg der update()-Methode.
- void notifyObservers()  
Entspricht notifyObservers(null).

## Die Schnittstelle Observer

Das aktive Objekt, der Sender der Nachrichten, ist ein Exemplar der Klasse Observable, das Benachrichtigungen an angemeldete Objekte schickt. Das aktive Objekt informiert alle zuhörenden Objekte, die die Schnittstelle Observer implementieren müssen.

Jetzt können wir für die Party auch die Zuhörer implementieren:

**Listing 10.2:** com/tutego/insel/pattern/observer/JokeListener.java

```
package com.tutego.insel.pattern.observer;
```

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

```
class JokeListener implements Observer
```

```
{
```

```
 final private String name;
```

```
JokeListener(String name)
```

```
{
```

```
 this.name = name;
```

```
}
```

```
@Override public void update(Observable o, Object arg)
```

```
{
```

```
 System.out.println(name + " lacht über: " + arg + " ");
```

```
}
```

```
}
```

```
interface java.util.Observer
```

- void update(Observable o, Object arg)

Wird bei Benachrichtigungen vom Observable o aufgerufen. Als zweites Argument

trifft die über `notifyObservers(Object)` verschickte Nachricht ein. Bei der parameterlosen Variante `notifyObservers()` ist der aktuelle Parameter `null`.

### Party-Beispiel mit Observer/Observable

Da auf einer echten Party die Zuhörer und Erzähler nicht fehlen dürfen, baut die dritte Klasse Party nun echte Stimmung auf:

**Listing 10.3:** com/tutego/insel/pattern/observer/Party.java

```
package com.tutego.insel.pattern.observer;

import java.util.*;

public class Party
{
 public static void main(String[] args)
 {
 Observer achim = new JokeListener("Achim");
 Observer michael = new JokeListener("Michael");
 JokeTeller chris = new JokeTeller();

 chris.addObserver(achim);

 chris.tellJoke();
 chris.tellJoke();

 chris.addObserver(michael);

 chris.tellJoke();

 chris.deleteObserver(achim);

 chris.tellJoke();
 }
}
```

Wir melden zwei Zuhörer nacheinander an und einen wieder ab. Dann geht das Lachen los.

## Schwierigkeiten von Observer/Observable

Die Typen `Observer`/`Observable` bieten eine grundlegende Möglichkeit, das Beobachter-Muster in Java zu realisieren. Allerdings gibt es ein paar Dinge, die Entwickler sich noch zusätzlich wünschen:

- Die Typen `Observer`/`Observable` sind nicht generisch deklariert, was dazu führt, dass bei `update()` immer nur alles als `Object` übergeben werden kann. In `update()` muss der Typ des Objekts oft auf den wirklichen Typ »hochgecastet« werden, um etwa Daten aus dem Ereignisobjekt zu entnehmen.
- Oder `Observer` deklariert nur genau eine `update()`-Methode. Wenn der Ereignisauslöser unterschiedliche Ereignisse melden möchte, gibt es nur eine Lösung: unterschiedliche Ergebnis-Objekte. Das wiederum führt zu Fallunterscheidungen in der `update()`-Methode, und die Codequalität verschlechtert sich. Besser wären mehrere `update()`-Methoden, die im Optimalfall auch nicht einfach nur `update()` hießen, sondern semantisch starke Namen tragen würden. Das Interessante ist, dass bei verschiedenen Methoden auch gar kein Event-Objekt mehr nötig wäre, sodass dieses Modell ohne Zustand aus käme, solange keine Ereignisdaten zu übermitteln sind.

Um das erste Problem zu lösen, können wir eine generische Deklaration der Schnittstelle/Klasse angeben:

**Listing 10.4:** com/tutego/insel/pattern/observer/generic/Observer.java, `Observer`

```
interface Observer<T>
{
 public void update(Observable<T> o, T arg);
}
```

**Listing 10.5:** com/tutego/insel/pattern/observer/generic/Observable.java, `Observable`

```
public class Observable<T>
{
 private final List<Observer<T>> observers = new ArrayList<Observer<T>>();

 public void addObserver(Observer<T> observer)
 {
 if (! observers.contains(observer))
 observers.add(observer);
 }
}
```

```

public void deleteObserver(Observer<?> observer)
{
 observers.remove(observer);
}

public void notifyObservers(T arg)
{
 for (Observer<T> observer : observers)
 observer.update(this, arg);
}
}

```

Um es einfach zu halten, sind nur die zentralen Methoden in Observable realisiert, und die Frage der Nebenläufigkeit wurde beiseitegeschoben.

Kommen wir zu unserem Party-Beispiel zurück: Dann ist das Ereignis vom Typ String, sodass der JokeListener daraufhin Observer<String> implementiert und update(Observable<String> o, String arg) realisiert und der JokeTeller die Klasse Observable<String> erweitert.

### 10.2.3 Ereignisse über Listener

Eine zweite Variante zur Implementierung des Beobachter-Musters sind *Listener*. Sie lösen die beiden genannten Probleme von eben. Es gibt Ereignisauslöser, die spezielle Ereignis-Objekte aussenden, und Interessenten, die sich bei den Auslösern an- und abmelden. Die beteiligten Klassen und Schnittstellen folgen einer bestimmten Namenskonvention; XXX steht im Folgenden stellvertretend für einen Ereignisnamen:

- Eine Klasse für die Ereignisobjekte heißt XXXEvent. Die Ereignisobjekte können Informationen wie Auslöser, Zeitstempel und weitere Daten speichern.
- Die Interessenten implementieren als Listener eine Java-Schnittstelle, die XXXListener heißt. Die Operation der Schnittstelle kann beliebig lauten, doch wird ihr üblicherweise das XXXEvent übergeben. Anders als beim Observer/Observable kann diese Schnittstelle auch mehrere Operationen vorschreiben.
- Der Ereignisauslöser bietet Methoden addXXXListener(XXXListener) und removeXXXListener(XXXListener) an, um Interessenten an- und abzumelden. Immer dann, wenn ein Ereignis stattfindet, erzeugt der Auslöser das Ereignisobjekt XXXEvent und informiert

jeden Listener, der in der Liste eingetragen ist, über einen Aufruf der Methode aus dem Listener.

Ein Beispiel soll die beteiligten Typen verdeutlichen:

### Radios spielen Werbung

Ein Radio soll für Werbungen AdEvent-Objekte aussenden. Die Ereignis-Objekte sollen den Werbespruch (Slogan) speichern:

**Listing 10.6:** com/tutego/insel/pattern/listener/AdEvent.java

```
package com.tutego.insel.pattern.listener;
```

```
import java.util.EventObject;
```

```
public class AdEvent extends EventObject
{
 private String slogan;

 public AdEvent(Object source, String slogan)
 {
 super(source);
 this.slogan = slogan;
 }

 public String getSlogan()
 {
 return slogan;
 }
}
```

Die Klasse AdEvent erweitert die Java-Basisklasse EventObject, eine Klasse, die alle Ereignis-Klassen erweitern. Der parametrisierte Konstruktor von AdEvent nimmt im ersten Parameter den Ereignisauslöser an und gibt ihn mit `super(source)` an den Konstruktor der Oberklasse weiter, der ihn speichert und mit `getSource()` wieder verfügbar macht. Der zweite Parameter vom AdEvent-Konstruktor ist unsere Werbung.

Der AdListener ist die Schnittstelle, die Interessenten implementieren:

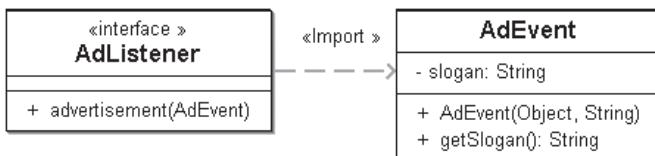
**Listing 10.7:** com/tutego/insel/pattern/listener/AdListener.java

```
package com.tutego.insel.pattern.listener;

import java.util.EventListener;

interface AdListener extends EventListener
{
 void advertisement(AdEvent e);
}
```

Unser AdListener implementiert die Schnittstelle EventListener (eine Markierungsschnittstelle), die alle Java-Listener implementieren sollen. Wir schreiben für konkrete Listener nur eine Operation advertisement() vor.



**Abbildung 10.2:** Klassendiagramm von AdListener, das AdEvent referenziert

Das Radio soll nun Interessenten an- und abmelden können. Es sendet über einen Timer Werbenachrichten. Das Spannende an der Implementierung ist die Tatsache, dass die Listener nicht in einer eigenen Datenstruktur verwaltet werden, sondern dass eine spezielle Listener-Klasse aus dem Swing-Paket verwendet wird:

**Listing 10.8:** com/tutego/insel/pattern/listener/Radio.java

```
package com.tutego.insel.pattern.listener;

import java.util.*;
import javax.swing.event.EventListenerList;

public class Radio
{
 private EventListenerList listeners = new EventListenerList();
```

```
private List<String> ads = Arrays.asList("Jetzt explodiert auch der Haarknoten",
 "Red Fish verleiht Flossen",
 "Bom Chia Wowo",
 "Wunder Whip. Iss milder.");

public Radio()
{
 new Timer().schedule(new TimerTask()
 {
 @Override public void run()
 {
 Collections.shuffle(ads);
 notifyAdvertisement(new AdEvent(this, ads.get(0)));
 }
 }, 0, 500);
}

public void addAdListener(AdListener listener)
{
 listeners.add(AdListener.class, listener);
}

public void removeAdListener(AdListener listener)
{
 listeners.remove(AdListener.class, listener);
}

protected synchronized void notifyAdvertisement(AdEvent event)
{
 for (AdListener l : listeners.getListeners(AdListener.class))
 l.advertisement(event);
}
```

Die Demo-Anwendung nutzt das Radio-Objekt und implementiert einen konkreten Listener:

```
package com.tutego.insel.pattern.listener;

public class RadioDemo
{
 public static void main(String args[])
 {
 Radio r = new Radio();

 class ComplainingAdListener implements AdListener {
 @Override public void advertisement(AdEvent e) {
 System.out.println("Oh nein, schon wieder Werbung: " + e.getSlogan());
 }
 }

 r.addAdListener(new ComplainingAdListener());
 }
}
```

Die Java API-Dokumentation enthält einige generische Typen:

- class javax.swing.event.EventListenerList**
- **EventListenerList()**  
Erzeugt eine Klasse zum Speichern von Listenern.
- **<T extends EventListener> void add(Class<T> t, T l)**  
Fügt einen Listener l vom Typ T hinzu.
- **Object[] getListenerList()**  
Liefert ein Feld aller Listener.
- **<T extends EventListener> T[] getListeners(Class<T> t)**  
Liefert ein Feld aller Listener vom Typ t.
- **int getListenerCount()**  
Nennt die Anzahl aller Listener.

- `int getListenerCount(Class<?> t)`  
Nennt die Anzahl der Listener vom Typ t.
- `<T extends EventListener> void remove(Class<T> t, T l)`  
Entfernt den Listener l aus der Liste.

## 10.3 JavaBean

Die Architektur von *JavaBeans* ist ein einfaches Komponenten-Modell. Ursprünglich waren JavaBeans eng mit grafischen Oberflächen verbunden, und so liest sich in der JavaBeans-Spezifikation 1.01 von 1997 noch:

»A Java Bean is a reusable software component that can be manipulated visually in a builder tool.«

Heutzutage ist das Feld viel größer, und Beans kommen in allen Ecken der Java-Bibliothek vor: bei der Persistenz (also bei der Abbildung der Objekte in relationalen Datenbanken oder XML-Dokumenten), als Datenmodelle für Webanwendungen, bei grafischen Oberflächen und in vielen weiteren Einsatzgebieten.

Im Kern basieren JavaBeans auf:

- *Selbstbeobachtung (Introspection)*. Eine Klasse lässt sich von außen auslesen. So kann ein spezielles Programm, etwa ein GUI-Builder oder eine visuelle Entwicklungsumgebung, eine Bean analysieren und ihre Eigenschaften abfragen. Auch umgekehrt kann eine Bean herausfinden, ob sie etwa gerade von einem grafischen Entwicklungswerkzeug modelliert wird oder in einer Applikation ohne GUI Verwendung findet.
- *Eigenschaften (Properties)*. Attribute beschreiben den Zustand des Objekts. In einem Modellierungswerkzeug lassen sie sich ändern. Da eine Bean zum Beispiel eine grafische Komponente sein kann, hat sie etwa eine Hintergrundfarbe. Diese Informationen können von außen durch bestimmte Methoden abgefragt und verändert werden. Für alle Eigenschaften werden spezielle Zugriffsmethoden deklariert; sie werden *Property-Design-Patterns* genannt.
- *Ereignissen (Events)*. Komponenten können Ereignisse auslösen, die Zustandsänderungen oder Programmteile weiterleiten können.
- *Anpassung (Customization)*. Der Bean-Entwickler kann die Eigenschaften einer Bean visuell und interaktiv anpassen.

- *Speicherung (Persistenz)*. Jede Bean kann ihren internen Zustand, also die Eigenschaften, durch Serialisierung speichern und wiederherstellen. So kann ein Builder-Tool die Komponenten laden und benutzen. Ein spezieller Externalisierungsmechanismus erlaubt dem Entwickler die Definition eines eigenen Speicherformats, zum Beispiel als XML-Datei.

Zusätzlich zu diesen notwendigen Grundpfeilern lässt sich durch Internationalisierung die Entwicklung internationaler Komponenten vereinfachen. Verwendet eine Bean länderspezifische Ausdrücke, wie etwa Währungs- oder Datumsformate, kann der Bean-Entwickler mit länderunabhängigen Bezeichnern arbeiten, die dann in die jeweilige Landessprache übersetzt werden.

### 10.3.1 Properties (Eigenschaften)

Die Properties einer JavaBean steuern den Zustand des Objekts. Bisher hat Java keine spezielle Schreibweise für Properties – anders als C# und andere Sprachen –, und so nutzt es eine spezielle Namensgebung bei den Methoden, um Eigenschaften zu lesen und zu schreiben. Der JavaBeans-Standard unterscheidet vier Arten von Properties:

- *Einfache Eigenschaften*. Hat eine Person eine Property »Name«, so bietet die Java-Bean die Methoden `getName()` und `setName()` an.
- *Indizierte/Indexierte Eigenschaften* (engl. *indexed properties*). Sie werden eingesetzt, falls mehrere gleiche Eigenschaften aus einem Array verwaltet werden. So lassen sich Felder gleichen Datentyps verwalten.
- *Gebundene Eigenschaften* (engl. *bound properties*). Ändert eine JavaBean ihren Zustand, kann sie angemeldete Interessenten (Listener) informieren.
- *Eigenschaft mit Vetorecht* (engl. *veto properties*, auch *constraint properties* beziehungsweise *eingeschränkte Eigenschaften* genannt). Ihre Benutzung ist in jenen Fällen angebracht, in denen eine Bean Eigenschaften ändern möchte, andere Beans aber dagegen sind und ihr Veto einlegen.

Die Eigenschaften der Komponente können primitive Datentypen, aber auch komplexe Klassen sein. Der Text einer Schaltfläche ist ein einfacher String; eine Sortierstrategie in einem Sortierprogramm ist dagegen ein komplexes Objekt.

### 10.3.2 Einfache Eigenschaften

Für die einfachen Eigenschaften muss für die Setter und Getter nur ein Paar von setXXX()- und getXXX()-Methoden eingesetzt werden. Der Zugriff auf eine Objektvariable wird also über Methoden geregelt. Dies hat den Vorteil, dass ein Zugriffsschutz und weitere Überprüfungen eingerichtet werden können. Soll eine Eigenschaft nur gelesen werden (weil sie sich zum Beispiel regelmäßig automatisch aktualisiert), müssen wir die setXXX()-Methode nicht implementieren. Genauso gut können wir Werte, die außerhalb des erlaubten Wertebereichs unserer Applikation liegen, prüfen und ablehnen. Dazu kann eine Methode eine Exception auslösen.

Allgemein sieht dann die Signatur der Methoden für eine Eigenschaft XXX vom Typ T folgendermaßen aus:

- public T getXXX()
- public void setXXX( T value )

Ist der Property-Typ ein Wahrheitswert, ist neben der Methode getXXX() eine isXXX()-Methode erlaubt:

- public boolean isXXX()
- public void setXXX( boolean value )

### 10.3.3 Indizierte Eigenschaften

Falls eine Bean nur über eine einfache Eigenschaft wie eine primitive Variable verfügt, so weisen die getXXX()-Methoden keinen Parameter und genau einen Rückgabewert auf. Der Rückgabewert hat den gleichen Datentyp wie die interne Eigenschaft. Die setXXX()-Methode besitzt genau einen Parameter des Datentyps dieser Eigenschaft und hat keinen expliziten Rückgabewert, sondern void. Wenn nun kein atomarer Wert, sondern ein Feld von Werten intern gespeichert ist, müssen wir Zugriff auf bestimmte Werte bekommen. Daher erwarten die setXXX()- und getXXX()-Methoden im zusätzlichen Parameter einen Index:

- public T[] getXXX()
- public T getXXX( int index )
- public void setXXX( T[] values )
- public void setXXX( T value, int index )

### 10.3.4 Gebundene Eigenschaften und PropertyChangeListener

Die *gebundenen Eigenschaften* einer Bean erlauben es, andere Komponenten über eine Zustandsänderung der Properties zu informieren. Wenn sich zum Beispiel der Name eines Spielers durch den Aufruf einer Methode `setName()` ändert, führt die Namensänderung vielleicht an anderer Stelle zu einer Aktualisierung der Darstellung. Bei den gebundenen Eigenschaften (engl. *bound properties*) geht es ausschließlich um Änderungen der Properties und nicht um andere Ereignisse, die nichts mit den Bean-Eigenschaften zu tun haben.

Die Listener empfangen von der Bean ein `PropertyChangeEvent`, das sie auswerten können. Die Interessierten implementieren dafür `PropertyChangeListener`. Das Ereignis-Objekt speichert den alten und den neuen Wert sowie den Typ und den Namen der Eigenschaft.

Die Bean muss also nur die Interessenten aufnehmen und dann feuern, wenn es eine Änderung an den Properties gibt. Da die Verwaltung der Listener immer gleich ist, bietet Java hier schon eine Klasse an: `PropertyChangeSupport`, die die JavaBeans nutzen, um die Listener zu verwalten. Die Interessenten lassen sich mit `addPropertyChangeListener()` als Zuhörer einfügen und mit `removePropertyChangeListener()` abhängen. Bei einer Veränderung ruft die Bean auf dem `PropertyChangeSupport`-Objekt die Methode `firePropertyChange()` auf, und so werden alle registrierten Zuhörer durch ein `PropertyChangeEvent` informiert. Die Zuhörer werden erst nach der Änderung des internen Zustands informiert.

Ein Beispiel: Unsere Person-Komponente besitzt eine Property »Name«, die der Setter `setName()` ändert. Nach der Änderung werden alle Listener informiert. Sie bewirkt darüber hinaus nichts Großartiges:

**Listing 10.9:** com/tutego/insel/bean/bound/Person.java

```
package com.tutego.insel.bean.bound;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Person
{
 private String name = "";

 private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

```

public void setName(String name)
{
 String oldName = this.name;
 this.name = name;
 changes.firePropertyChange("name", oldName, name);
}

public String getName()
{
 return name;
}

public void addPropertyChangeListener(PropertyChangeListener l)
{
 changes.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(PropertyChangeListener l)
{
 changes.removePropertyChangeListener(l);
}
}

```

Der Implementierung `setName()` kommt zentrale Bedeutung zu. Der erste Parameter von `firePropertyChange()` ist der Name der Eigenschaft. Er ist für das Ereignis von Belang und muss nicht zwingend der Name der Bean-Eigenschaft sein. Es folgen der alte und der neue Stand des Werts. Die Methode informiert alle angemeldeten Zuhörer über die Änderung mit einem `PropertyChangeEvent`.

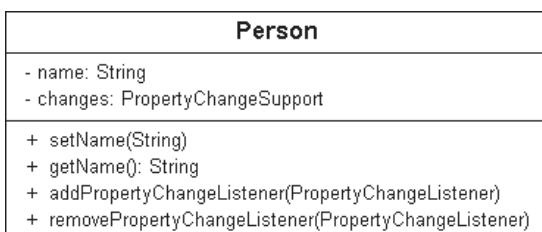


Abbildung 10.3: UML-Diagramm der Klasse Person zeigt die Listener-Unterstützung.

```
class java.beans.PropertyChangeSupport
 implements Serializable
```

- `PropertyChangeSupport(Object sourceBean)`  
Konstruiert ein PropertyChangeSupport-Objekt, das sourceBean als auslösende Bean betrachtet.
- `synchronized void addPropertyChangeListener(PropertyChangeListener listener)`  
Fügt einen Listener hinzu.
- `synchronized void removePropertyChangeListener(PropertyChangeListener listener)`  
Entfernt einen Listener.
- `synchronized void addPropertyChangeListener(String propertyName,
 PropertyChangeListener listener)`  
Fügt einen Listener hinzu, der nur auf Ereignisse mit dem Namen propertyName hört.
- `synchronized void removePropertyChangeListener(String propertyName,
 PropertyChangeListener listener)`  
Entfernt den Listener, der auf propertyName hört.
- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)`  
Informiert alle Listener über eine Werteänderung. Sind alte und neue Werte gleich, werden keine Events ausgelöst.
- `void firePropertyChange(String propertyName, int oldValue, int newValue)`
- `void firePropertyChange(String propertyName, boolean oldValue, boolean newValue)`  
Varianten von `firePropertyChange()` mit Integer- und Boolean-Werten
- `void firePropertyChange(PropertyChangeEvent evt)`  
Informiert alle Interessenten mit einem PropertyChangeEvent, indem es `propertyChange()` aufruft.
- `synchronized boolean hasListeners(String propertyName)`  
Liefert true, wenn es mindestens einen Listener für die Eigenschaft gibt.

Angemeldete PropertyChangeListener können auf das PropertyChangeEvent reagieren. Wir testen das an einer Person, die einen neuen Namen bekommt:

**Listing 10.10:** com/tutego/insel/bean/bound/PersonWatcher.java, main()

```
Person p = new Person();
p.addPropertyChangeListener(new PropertyChangeListener()
{
 @Override public void propertyChange(PropertyChangeEvent e)
```

```

{
 System.out.printf("Property '%s': '%s' -> '%s'%n",
 e.getPropertyName(), e.getOldValue(), e.getNewValue());
}
});
p.setName("Ulli"); // Property 'name': '' -> 'Ulli'
p.setName("Ulli");
p.setName("Chris"); // Property 'name': 'Ulli' -> 'Chris'

```

Beim zweiten `setName()` erfolgt kein Event, da es nur dann ausgelöst wird, wenn der Wert wirklich nach der `equals()`-Methode anders ist.

```

interface java.beans.PropertyChangeListener
extends java.util.EventListener

```

- `void propertyChange(PropertyChangeEvent evt)`

Wird aufgerufen, wenn sich die gebundene Eigenschaft ändert. Über das `PropertyChangeEvent` erfahren wir die Quelle und den Inhalt der Eigenschaft.

```

class java.beans.PropertyChangeEvent
extends java.util.EventObject

```

- `PropertyChangeEvent(Object source, String propertyName, Object oldValue, Object newValue)`

Erzeugt ein neues Objekt mit der Quelle, die das Ereignis auslöst, einem Namen, dem alten und dem gewünschten Wert. Die Werte werden intern in privaten Variablen gehalten und lassen sich später nicht mehr ändern.

- `String getPropertyName()`  
Liefert den Namen der Eigenschaft.
- `Object getNewValue()`  
Liefert den neuen Wert.
- `Object getOldValue()`  
Liefert den alten Wert.

10

### 10.3.5 Veto-Eigenschaften – dagegen!

Wenn sich der Zustand einer gebundenen Eigenschaft ändert, informieren JavaBeans ihre Zuhörer darüber. Möglicherweise haben diese Zuhörer jedoch etwas gegen diesen neuen Wert. In diesem Fall kann ein Zuhörer ein Veto mit einer `PropertyVetoException`

einlegen und so eine Werteänderung verhindern. Es geht nicht darum, dass die Komponente selbst den Wert ablehnt – es geht um die Interessenten, die das nicht wollen!

Bevor eine JavaBean eine Änderung an einer Property durchführt, holen wir zunächst die Zustimmung ein. Programmieren wir eine `setXXX()`-Methode mit Veto, gibt es im Rumpf vor dem meldenden `firePropertyChange()` ein fragendes `fireVetoableChange()`, das die Veto-Listener informiert. Der Veto-Listener kann durch eine ausgelöste `PropertyVetoException` anzeigen, dass er gegen die Änderung war. Das bricht den Setter ab, und es kommt nicht zum `firePropertyChange()`. Wegen der `PropertyVetoException` muss auch die Methode eine Signatur mit `throws PropertyVetoException` besitzen.

In unserem Beispiel darf die Person ein Bigamist sein. Aber natürlich nur dann, wenn es kein Veto gab!

**Listing 10.11:** com/tutego/insel/bean/veto/Person.java

```
package com.tutego.insel.bean.veto;

import java.beans.*;

public class Person
{
 private boolean bigamist;

 private PropertyChangeSupport changes = new PropertyChangeSupport(this);
 private VetoableChangeSupport vetos = new VetoableChangeSupport(this);

 public void setBigamist(boolean bigamist) throws PropertyVetoException
 {
 boolean oldValue = this.bigamist;
 vetos.fireVetoableChange("bigamist", oldValue, bigamist);
 this.bigamist = bigamist;
 changes.firePropertyChange("bigamist", oldValue, bigamist);
 }

 public boolean isBigamist()
 {
 return bigamist;
 }
}
```

```
public void addPropertyChangeListener(PropertyChangeListener l)
{
 changes.addPropertyChangeListener(l);
}

public void removePropertyChangeListener(PropertyChangeListener l)
{
 changes.removePropertyChangeListener(l);
}

public void addVetoableChangeListener(VetoableChangeListener l)
{
 vetos.addVetoableChangeListener(l);
}

public void removeVetoableChangeListener(VetoableChangeListener l)
{
 vetos.removeVetoableChangeListener(l);
}
```

Wie wir an dem Beispiel sehen, ist zusätzlich zum Veto eine gebundene Eigenschaft dabei. Das ist die Regel, damit Interessierte nicht nur gegen gewünschte Änderungen Einspruch erheben können, sondern die tatsächlich gemachten Belegungen ebenfalls erfahren. Der Kern einer Setter-Methode mit Veto ist es, erst eine Änderung mit `fireVetoableChange()` anzukündigen und dann, wenn es keine Einwände dagegen gibt, mit `firePropertyChange()` diese neue Belegung zu berichten.

Melden wir bei einer Person einen `PropertyChangeListener` an, um alle gültigen Zustandswechsel auszugeben:

**Listing 10.12:** com/tutego/insel/bean/veto/PersonWatcher.java, main() Teil 1

```
Person p = new Person();
p.addPropertyChangeListener(new PropertyChangeListener()
{
 @Override public void propertyChange(PropertyChangeEvent e)
 {
```

```

 System.out.printf("Property '%s': '%s' -> '%s'%n",
 e.getPropertyName(), e.getOldValue(), e.getNewValue());
 }
});

```

Ohne ein Veto gehen alle Zustandsänderungen durch:

**Listing 10.13:** com/tutego/insel/bean/veto/PersonWatcher.java, main() Teil 2

```

try
{
 p.setBigamist(true);
 p.setBigamist(false);
}
catch (PropertyVetoException e)
{
 e.printStackTrace();
}

```

Die Ausgabe wird sein:

```

Property 'bigamist': 'false' -> 'true'
Property 'bigamist': 'true' -> 'false'

```

Nach der Heirat darf unsere Person kein Bigamist mehr sein. Während am Anfang ein Wechsel der Zustände leicht möglich war, ist nach dem Hinzufügen eines veto-einlegenden VetoableChangeListener eine Änderung nicht mehr erlaubt:

**Listing 10.14:** com/tutego/insel/bean/veto/PersonWatcher.java, main() Teil 3

```

p.addVetoableChangeListener(new VetoableChangeListener()
{
 @Override
 public void vetoableChange(PropertyChangeEvent e)
 throws PropertyVetoException
 {
 if ("bigamist".equals(e.getPropertyName()))
 if ((Boolean) e.getNewValue())
 throw new PropertyVetoException("Nimm zwei ist nichts für mich!", e);
 }
});

```

Der Kern der Logik ist die Anweisung `throw new PropertyVetoException`. Jetzt sind keine unerwünschten Änderungen mehr möglich:

**Listing 10.15:** com/tutego/insel/bean/veto/PersonWatcher.java, main() Teil 4

```
try
{
 p.setBigamist(true);
}
catch (PropertyVetoException e)
{
 e.printStackTrace();
}
```

Das `setBigamist(true)` führt zu einer `PropertyVetoException`. Der Stack-Trace ist:

```
java.beans.PropertyVetoException: Nimm zwei ist nichts für mich!
at com.tutego.insel.bean.veto.PersonWatcher$2.vetoableChange(PersonWatcher.java:40)
at java.beans.VetoableChangeSupport.fireVetoableChange(VetoableChangeSupport.java:335)
at java.beans.VetoableChangeSupport.fireVetoableChange(VetoableChangeSupport.java:252)
at java.beans.VetoableChangeSupport.fireVetoableChange(VetoableChangeSupport.java:294)
at com.tutego.insel.bean.veto.Person.setBigamist(Person.java:19)
at com.tutego.insel.bean.veto.PersonWatcher.main(PersonWatcher.java:46)
```

Obwohl es mit `addPropertyChangeListener(PropertyChangeListener l)` sowie `addVetoableChangeListener(VetoableChangeListener l)` jeweils zwei Listener gibt, versenden beide Ereignis-Objekte vom Typ `PropertyChangeEvent`. Doch während bei Veto-Objekten vor der Zustandsänderung ein `PropertyChangeEvent` erzeugt und versendet wird, informieren die gebundenen Eigenschaften erst nach der Änderung ihre Zuhörer mit einem `PropertyChangeEvent`. Daher bedeutet das Aufkommen eines `PropertyChangeEvent` jeweils etwas Unterschiedliches.

```
class java.beans.VetoableChangeSupport
implements Serializable
```

- `void addVetoableChangeListener(VetoableChangeListener listener)`  
Fügt einen `VetoableListener` hinzu, der alle gewünschten Änderungen meldet.
- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener)`  
Fügt einen `VetoableListener` hinzu, der auf alle gewünschten Änderungen der Property `propertyName` hört.

- void fireVetoableChange(String propertyName, boolean oldValue, boolean newValue)
  - void fireVetoableChange(String propertyName, int oldValue, int newValue)
  - void fireVetoableChange(String propertyName, Object oldValue, Object newValue)
- Die `fireVetoableChange()`-Methoden melden eine gewünschte Änderung der Eigenschaft mit dem Namen `propertyName`.

```
interface java.beans.VetoableChangeListener
extends java.util.EventListener
```

- void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException
- Wird aufgerufen, wenn die gebundene Eigenschaft geändert werden soll. Über das `PropertyChangeEvent` erfahren wir die Quelle und den Inhalt der Eigenschaft. Die Methode löst eine `PropertyVetoException` aus, wenn die Eigenschaft nicht geändert werden soll.

## 10.4 Zum Weiterlesen

Viele Entwickler konzentrieren sich oft nur auf die Programmiersprache und APIs, aber weniger auf die Architektur oder das Design. Es ist empfohlen, Bücher und Literatur zu studieren, die sich auf den Entwurf von Anwendungen konzentrieren. Hervorzuheben ist in diesem Zusammenhang »Head First Design Patterns« von O'Reilly, 2004, (<http://headfirstlabs.com/books/hfdp/>), ISBN 978-0596007126.



# Kapitel 11

## Die Klassenbibliothek

»Was wir brauchen, sind ein paar verrückte Leute;  
seht euch an, wohin uns die Normalen gebracht haben.«  
– George Bernard Shaw (1856–1950)

### 11.1 Die Java-Klassenphilosophie

11

Eine Programmiersprache besteht nicht nur aus einer Grammatik, sondern, wie im Fall von Java, auch aus einer Programmierbibliothek. Eine plattformunabhängige Sprache – so wie sich viele C oder C++ vorstellen – ist nicht wirklich plattformunabhängig, wenn auf jedem Rechner andere Funktionen und Programmiermodelle eingesetzt werden. Genau dies ist der Schwachpunkt von C++. Die Algorithmen, die kaum vom Betriebssystem abhängig sind, lassen sich überall gleich anwenden, doch spätestens bei grafischen Oberflächen ist Schluss. Dieses Problem ergibt sich in Java seltener, weil sich die Entwickler große Mühe geben, alle wichtigen Methoden in wohlgeformten Klassen und Paketen unterzubringen. Diese decken insbesondere die zentralen Bereiche Datenstrukturen, Ein- und Ausgabe, Grafik- und Netzwerkprogrammierung ab.

#### 11.1.1 Übersicht über die Pakete der Standardbibliothek

Die Java 7-Klassenbibliothek bietet genau 208 Pakete.<sup>1</sup> Die wichtigsten davon fasst die folgende Tabelle zusammen:

---

<sup>1</sup> Unsere Kollegen aus der Microsoft-Welt müssen eine dicke Pille schlucken, denn .NET 4 umfasst 408 Pakete (*Assemblies* genannt). Dafür enthält .NET aber auch Dinge, die in der Java-Welt der Java EE zuzuordnen sind. Aber auch dann liegt .NET immer noch vorne, denn Java EE 6 deklariert gerade einmal 117 Pakete.

| Paket          | Beschreibung                                                                                                                                                    |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| java.awt       | Das Paket AWT ( <i>Abstract Windowing Toolkit</i> ) bietet Klassen zur Grafikausgabe und zur Nutzung von grafischen Bedienoberflächen.                          |
| java.awt.event | Schnittstellen für die verschiedenen Ereignisse unter grafischen Oberflächen                                                                                    |
| java.io        | Möglichkeiten zur Ein- und Ausgabe. Dateien werden als Objekte repräsentiert. Datenströme erlauben den sequenziellen Zugriff auf die Dateiinhalte.              |
| java.lang      | Ein Paket, das automatisch eingebunden ist und unverzichtbare Klassen wie String-, Thread- oder Wrapper-Klassen enthält                                         |
| java.net       | Kommunikation über Netzwerke. Bietet Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP beziehungsweise IP mit dem Internet verbinden lassen. |
| java.text      | Unterstützung für internationalisierte Programme. Bietet Klassen zur Behandlung von Text und zur Formatierung von Datumswerten und Zahlen.                      |
| java.util      | Bietet Typen für Datenstrukturen, Raum und Zeit sowie für Teile der Internationalisierung sowie für Zufallszahlen.                                              |
| javax.swing    | Swing-Komponenten für grafische Oberflächen. Das Paket besitzt diverse Unterpakete.                                                                             |

Tabelle 11.1: Wichtige Pakete in Java 7

Eine vollständige Übersicht aller Pakete gibt Anhang A, »Die Klassenbibliothek«. Als Entwickler ist es unumgänglich für die Details die JavaDoc unter <http://download.oracle.com/javase/7/docs/api/> zu studieren.

### Offizielle Schnittstelle (java und javax-Pakete)

Das, was die JavaDoc dokumentiert, bildet den erlaubten Zugang zum JDK. Die Typen sind für die Ewigkeit ausgelegt, sodass Entwickler darauf zählen können, auch noch in 100 Jahren ihre Java-Programme ausführen zu können. Doch wer definiert die API? Im Kern sind es vier Quellen:

- Oracle-Entwickler setzen neue Pakete und Typen in die API.
- Der *Java Community Process* (JCP) beschließt eine neue API. Dann ist es nicht nur Oracle allein, sondern eine Gruppe, die eine neue API erarbeitet und die Schnittstellen definiert.

- Die *Object Management Group* (OMG) definiert eine API für CORBA.
- Das *World Wide Web Consortium* (W3C) gibt eine API etwa für XML-DOM vor.

Die Merkhilfe ist, dass alles, was mit `java` oder `javax` beginnt, eine erlaubte API darstellt, und alles andere zu nicht portablen Java-Programmen führen kann. Es gibt weiterhin Klassen, die unterstützt werden, aber nicht Teil der offiziellen API sind. Dazu zählen etwa diverse Swing-Klassen für das Aussehen der Oberfläche.

#### Hinweis

Die Laufzeitumgebung von Oracle liefert noch über 3.000 Klassendateien in den Paketen `sun` und `sunw` aus. Diese internen Klassen sind nicht offiziell dokumentiert,<sup>2</sup> aber zum Teil sehr leistungsfähig und erlauben selbst direkten Speicherzugriff oder können Objekte ohne Standard-Konstruktor erzeugen:

**Listing 11.1:** com/tutego/insel/sun/UnsafeInstance.java, Ausschnitt

```
Field field = Unsafe.class.getDeclaredField("theUnsafe");
field.setAccessible(true);
sun.misc.Unsafe unsafe = (sun.misc.Unsafe) field.get(null);
File f = (File) unsafe.allocateInstance(File.class);
System.out.println(f.getPath()); // null
```

File hat keinen Standard-Konstruktor, nicht einmal einen privaten. Diese Art der Objekterzeugung kann bei der Deserialisierung hilfreich sein.

11

#### Standard Extension API (javax-Pakete)

Einige der Java-Pakete beginnen mit `javax`. Dies sind ursprünglich Erweiterungspakete (*Extensions*), die die Kern-Klassen ergänzen sollten. Im Laufe der Zeit sind jedoch viele der früher zusätzlich einzubindenden Pakete in die Standard-Distribution gewandert, sodass heute ein recht großer Anteil mit `javax` beginnt, aber keine Erweiterungen mehr darstellt, die zusätzlich installiert werden müssen. Sun wollte damals die Pakete nicht umbenennen, um so eine Migration nicht zu erschweren. Fällt heute im Quellcode ein Paketname mit `javax` auf, ist es daher nicht mehr so einfach zu entscheiden, ob eine externe Quelle mit eingebunden werden muss beziehungsweise ab welcher Java-Version das Paket Teil der Distribution ist. Echte externe Pakete sind unter anderem:

---

<sup>2</sup> Das Buch »Java Secrets« von Elliotte Rusty Harold, <http://ibiblio.org/java/books/secrets/>, IDG Books, ISBN 0764580078, geht einigen Klassen nach, ist aber schon älter.

- *Enterprise/Server API* mit den Enterprise *JavaBeans*, *Servlets* und *JavaServer Faces*
- *Java Persistence API* (JPA) zum dauerhaften Abbilden von Objekten auf (in der Regel) relationale Datenbanken
- *Java Communications API* für serielle und parallele Schnittstellen
- *Java Telephony API*
- Sprachein-/ausgabe mit der *Java Speech API*
- *JavaSpaces* für gemeinsamen Speicher unterschiedlicher Laufzeitumgebungen
- *JXTA* zum Aufbauen von P2P-Netzwerken

## 11.2 Sprachen der Länder

Programme der ersten Generation konnten nur mit fest verdrahteten Landessprachen und landesüblichen Bezeichnungen umgehen. Daraus ergaben sich natürlich vielfältige Probleme. Mehrsprachige Programme mussten aufwendig entwickelt werden, damit sie unter mehreren Sprachen lokalisierte Ausgaben lieferten. (Es ergaben sich bereits Probleme durch unterschiedliche Zeichenkodierungen. Dies umging aber der Unicode-Standard.) Es blieb das Problem, dass sprachabhängige Zeichenketten, wie alle anderen Zeichenketten auch, überall im Programmtext verteilt sind und eine nachträgliche Sprachanpassung nur aufwendig zu erreichen ist. Java bietet hier eine Lösung an: zum einen durch die Definition einer Sprache und damit durch automatische Formatierungen, und zum anderen durch die Möglichkeit, sprachenabhängige Teile in Ressourcen-Dateien auszulagern.

### 11.2.1 Sprachen und Regionen über Locale-Objekte

In Java repräsentieren `Locale`-Objekte geografische, politische oder kulturelle Regionen. Die Sprache und die Region müssen getrennt werden, denn nicht immer gibt eine Region oder ein Land die Sprache eindeutig vor. Für Kanada in der Umgebung von Quebec ist die französische Ausgabe relevant, und die unterscheidet sich von der englischen. Jede dieser sprachspezifischen Eigenschaften ist in einem speziellen Objekt gekapselt.

**zB**

#### Beispiel

Sprach-Objekte werden immer mit dem Namen der Sprache und optional mit dem Namen des Landes beziehungsweise einer Region erzeugt. Im Konstruktor der Klasse `Locale` werden dann Länderabkürzungen angegeben, etwa für ein Sprach-Objekt für Großbritannien oder Frankreich:

**Beispiel (Forts.) (Forts.)**

zB

```
Locale greatBritain = new Locale("en", "GB");
Locale french = new Locale("fr");
```

Im zweiten Beispiel ist uns das Land egal. Wir haben einfach nur die Sprache Französisch ausgewählt, egal in welchem Teil der Welt.

Die Sprachen sind durch Zwei-Buchstaben-Kürzel aus dem ISO-639-Code<sup>3</sup> (ISO Language Code) identifiziert, und die Ländernamen sind Zwei-Buchstaben-Kürzel, die in ISO 3166<sup>4</sup> (ISO Country Code) beschrieben sind.

```
final class java.util.Locale
 implements Cloneable, Serializable
```

- `Locale(String language)`  
Erzeugt ein neues Locale-Objekt für die Sprache (language), die nach dem ISO-693-Standard gegeben ist.
- `Locale(String language, String country)`  
Erzeugt ein Locale-Objekt für eine Sprache (language) nach ISO 693 und ein Land (country) nach dem ISO-3166-Standard.
- `public Locale(String language, String country, String variant)`  
Erzeugt ein Locale-Objekt für eine Sprache, ein Land und eine Variante. variant ist eine herstellerabhängige Angabe wie »WIN« oder »MAC«.

Die statische Methode `Locale.getDefault()` liefert die aktuell eingestellte Sprache. Für die laufende JVM kann `Locale.setLocale(Locale)` diese ändern.

11

### Konstanten für einige Länder und Sprachen

Die Locale-Klasse besitzt Konstanten für häufig auftretende Länder und Sprachen. Statt für Großbritannien explizit `new Locale("en", "GB")` zu schreiben, bietet die Klasse mit `Locale.UK` eine Abkürzung. Unter den Konstanten für Länder und Sprachen sind: CANADA, CANADA\_FRENCH, CHINA ist gleich CHINESE (und auch PRC bzw. SIMPLIFIED\_CHINESE), ENGLISH, FRANCE, FRENCH, GERMAN, GERMANY, ITALIAN, ITALY, JAPAN, JAPANESE, KOREA, KOREAN, TAIWAN (ist gleich TRADITIONAL\_CHINESE), UK und US.

3 [http://www.loc.gov/standards/iso639-2/php/code\\_list.php](http://www.loc.gov/standards/iso639-2/php/code_list.php)

4 <http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/index.html>

## Methoden von Locale

Locale-Objekte bieten eine Reihe von Methoden an, um etwa den ISO-639-Code des Landes preiszugeben.

### zB Beispiel

Gib für Deutschland zugängliche Informationen aus. Das Objekt out aus System und GERMANY aus Locale sind statisch importiert:

**Listing 11.2:** com/tutego/insel/locale/GermanyLocal.java, main()

```
out.println(GERMANY.getCountry()); // DE
out.println(GERMANY.getLanguage()); // de
out.println(GERMANY.getVariant()); //
out.println(GERMANY.getDisplayCountry()); // Deutschland
out.println(GERMANY.getDisplayLanguage()); // Deutsch
out.println(GERMANY.getDisplayName()); // Deutsch (Deutschland)
out.println(GERMANY.getDisplayVariant()); //
out.println(GERMANY.getISO3Country()); // DEU
out.println(GERMANY.getISO3Language()); // deu
```

```
final class java.util.Locale
implements Cloneable, Serializable
```

- `String getCountry()`  
Liefert das Länderkürzel nach dem ISO-3166-zwei-Buchstaben-Code.
- `String getLanguage()`  
Liefert das Kürzel der Sprache im ISO-639-Code.
- `String getVariant()`  
Liefert das Kürzel der Variante.
- `final String getDisplayCountry()`  
Liefert ein Kürzel des Landes für Bildschirmausgaben.
- `final String getDisplayLanguage()`  
Liefert ein Kürzel der Sprache für Bildschirmausgaben.
- `final String getDisplayName()`  
Liefert den Namen der Einstellungen.
- `final String getDisplayVariant()`  
Liefert den Namen der Variante.

- `String getISO3Country()`  
Liefert die ISO-Abkürzung des Landes dieser Einstellungen und löst eine MissingResourceException aus, wenn die ISO-Abkürzung nicht verfügbar ist.
- `String getISO3Language()`  
Liefert die ISO-Abkürzung der Sprache dieser Einstellungen und löst eine MissingResourceException aus, wenn die ISO-Abkürzung nicht verfügbar ist.
- `static Locale[] getAvailableLocales()`  
Liefert eine Aufzählung aller installierten Locale-Objekte. Das Feld enthält mindestens Locale.US und unter Java 7 fast 160 Einträge.



Abbildung 11.1: UML-Diagramm der Locale-Klasse

## 11.3 Die Klasse Date

Die ältere Klasse `java.util.Date` ist durch die Aufgabenverteilung auf die Klassen `DateFormat` und `Calendar` sehr schlank. Ein Exemplar der Klasse `Date` verwaltet ein besonderes Datum oder eine bestimmte Zeit; die Zeitgenauigkeit beträgt eine Millisekunde. `Date`-Objekte sind `mutable`, also veränderbar. Sie lassen sich daher nur mit Vorsicht an Methoden übergeben oder zurückgeben.

Im SQL-Paket gibt es eine Unterklasse von `java.util.Date`, die Klasse `java.sql.Date`. Bis auf eine statische Methode `java.sql.Date.valueOf(String)`, die Zeichenfolgen mit dem Aufbau »yyyy-mm-dd« erkennt, gibt es keine Unterschiede.

### 11.3.1 Objekte erzeugen und Methoden nutzen

Viele Methoden von `Date` sind veraltet, und zwei Konstruktoren der Klasse bleiben uns:

```
class java.util.Date
implements Serializable, Cloneable, Comparable<Date>
```

- `Date()`

Erzeugt ein Datum-Objekt und initialisiert es mit der Zeit, die bei der Erzeugung gelesen wurde. Die gegenwärtige Zeit erfragt dieser Konstruktor mit `System.currentTimeMillis()`.

- `Date(long date)`

Erzeugt ein Datum-Objekt und initialisiert es mit der übergebenen Anzahl von Millisekunden seit dem 1. Januar 1970, 00:00:00 GMT.

**zB**

### Beispiel

Mit der `toString()`-Methode können wir ein minimales Zeitanzeige-Programm schreiben. Wir rufen den Standard-Konstruktor auf und geben dann die Zeit aus. Die `println()`-Methode ruft wie üblich automatisch `toString()` auf:

**Listing 11.3:** com/tutego/insel/date/MiniClock.java

```
package com.tutego.insel.date;
```

```
class MiniClock
{
```

zB

**Beispiel (Forts.)**

```
public static void main(String[] args)
{
 System.out.println(new java.util.Date()); // Fri Jul 07 09:05:16 CEST 2006
}
```

Die anderen Methoden erlauben Zeitvergleiche und operieren auf den Millisekunden.

- class **java.util.Date**  
implements Serializable, Cloneable, Comparable<Date>
- long **getTime()**  
Liefert die Anzahl der Millisekunden nach dem 1. Januar 1970, 00:00:00 GMT zurück.  
Der Wert ist negativ, wenn der Zeitpunkt vor dem 1.1.1970 liegt.
- void **setTime(long time)**  
Setzt wie der Konstruktor die Anzahl der Millisekunden des Datum-Objekts neu.
- boolean **before(Date when)**  
Testet, ob das eigene Datum vor oder nach dem übergebenen Datum liegt: Gibt true zurück, wenn when vor oder nach dem eigenen Datum liegt, sonst false. Falls die Millisekunden in long bekannt sind, kommt ein Vergleich mit den primitiven Werten zum gleichen Ergebnis.
- boolean **equals(Object obj)**  
Testet die Datumsobjekte auf Gleichheit. Gibt true zurück, wenn getTime() für den eigenen Zeitwert und das Datumsobjekt hinter obj den gleichen Wert ergibt und der aktuelle Parameter nicht null ist.
- int **compareTo(Date anotherDate)**  
Vergleicht zwei Datum-Objekte und gibt 0 zurück, falls beide die gleiche Zeit repräsentieren. Der Rückgabewert ist kleiner 0, falls das Datum des aufrufenden Exemplars vor dem Datum von anotherDate ist, sonst größer 0.
- int **compareTo(Object o)**  
Ist das übergebene Objekt vom Typ Date, dann verhält sich die Methode wie compareTo(). Andernfalls löst die Methode eine ClassCastException aus. Die Methode ist eine Vorgabe aus der Schnittstelle Comparable. Mit der Methode lassen sich Date-

Objekte in einem Feld über `Arrays.sort(Object[])` oder `Collections.sort()` einfach sortieren.

- `String toString()`

Gibt eine Repräsentation des Datums aus. Das Format ist nicht landesspezifisch.

### 11.3.2 Date-Objekte sind nicht immutable

Dass Date-Objekte nicht immutable sind, ist sicherlich aus heutiger Sicht eine große Designschwäche. Immer dann, wenn Date-Objekte übergeben und zurückgegeben werden sollen, ist eine Kopie des Zustands das Beste, damit nicht später plötzlich ein verteiltes Date-Objekt ungewünschte Änderungen an den verschiedensten Stellen provoziert. Am besten sieht es also so aus:

**Listing 11.4:** com.tutego.insel.date.Person.java, Person

```
class Person
{
 private Date birthday;

 public void setBirthday(Date birthday)
 {
 this.birthday = new Date(birthday.getTime());
 }

 public Date getBirthday()
 {
 return new Date(birthday.getTime());
 }
}
```

#### → Hinweis

Eigentlich hat Sun die verändernden Methoden wie `setHours()` oder `setMinutes()` für *deprecated* erklärt. Allerdings blieb eine Methode außen vor: `setTime(long)`, die die Anzahl der Millisekunden seit dem 1.1.1970 neu setzt. In Programmen sollte diese zustandsverändernde Methode vorsichtig eingesetzt und stattdessen die Millisekunden im Konstruktor für ein neues Date-Objekt übergeben werden.

## 11.4 Calendar und GregorianCalendar

Ein Kalender unterteilt die Zeit in Einheiten wie Jahr, Monat, Tag. Der bekannteste Kalender ist der *gregorianische Kalender*, den Papst Gregor XIII. im Jahre 1582 einführte. Vor seiner Einführung war der *julianische Kalender* populär, der auf Julius Cäsar zurückging – daher auch der Name. Er stammt aus dem Jahr 45 vor unserer Zeitrechnung. Der gregorianische und der julianische Kalender sind Sonnenkalender, die den Lauf der Erde um die Sonne als Basis für die Zeiteinteilung nutzen; der Mond spielt keine Rolle. Daneben gibt es Mondkalender wie den islamischen Kalender und die Lunisolarkalender, die Sonne und Mond miteinander verbinden. Zu diesem Typus gehören der chinesische, der griechische und der jüdische Kalender.

Mit Exemplaren vom Typ `Calendar` ist es möglich, Datum und Uhrzeit in den einzelnen Komponenten wie Jahr, Monat, Tag, Stunde, Minute, Sekunde zu setzen und zu erfragen. Da es unterschiedliche Kalendertypen gibt, ist `Calendar` eine abstrakte Basisklasse, und Unterklassen bestimmen, wie konkret eine Abfrage oder Veränderung für ein bestimmtes Kalendersystem aussehen muss. Bisher bringt die Java-Bibliothek mit der Unterklasse `GregorianCalendar` nur eine öffentliche konkrete Implementierung mit, deren Exemplare Daten und Zeitpunkte gemäß dem gregorianischen Kalender verkörpern. In Java 6 ist eine weitere interne Klasse für einen japanischen Kalender hinzugekommen. IBM hat mit *International Components for Unicode for Java (ICU4J)* unter <http://icu.sourceforge.net/> weitere Klassen wie `ChineseCalendar`, `BuddhistCalendar`, `JapaneseCalendar`, `HebrewCalendar` und `IslamicCalendar` freigegeben. Hier findet sich auch einiges zum Thema Ostertage.

### 11.4.1 Die abstrakte Klasse Calendar

Die Klasse `Calendar` besitzt zum einen Anfrage- und Modifikationsmethoden für konkrete Exemplare und zum anderen statische Fabrikmethoden. Eine einfache statische Methode ist `getInstance()`, um ein benutzbares Objekt zu bekommen.

```
abstract class java.util.Calendar
 implements Serializable, Cloneable, Comparable<Calendar>
```

- `static Calendar getInstance()`  
Liefert einen Standard-Calendar mit der Standard-Zeitzone und Standard-Lokalisierung zurück.

Neben der parameterlosen Variante von `getInstance()` gibt es drei weitere Varianten, denen ein `TimeZone`-Objekt und `Locale`-Objekt mit übergeben werden kann. Damit kann dann der Kalender auf eine spezielle Zeitzone und einen Landstrich zugeschnitten werden.



Abbildung 11.2: UML-Diagramm der Klasse Calendar



### Hinweis

`Calendar` (bzw. `GregorianCalendar`) hat keine menschenfreundliche `toString()`-Methode. Der String enthält alle Zustände des Objekts:

**Hinweis (Forts.)**

```
java.util.GregorianCalendar[time=1187732409256,areFieldsSet=true,
areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/
Berlin",offset=3600000,dstSavings=3600000,useDaylight=true,transitions=143,
lastRule=java.util.SimpleTimeZone[id=Europe/Berlin,offset=3600000,
dstSavings=3600000,useDaylight=true,startYear=0,startMode=2,startMonth=2,
startDay=-1,startDayOfWeek=1,startTime=3600000,startTimeMode=2,endMode=2,
endMonth=9,endDay=-1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]],
firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,YEAR=2007,MONTH=7,
WEEK_OF_YEAR=34,WEEK_OF_MONTH=4,DAY_OF_MONTH=21,DAY_OF_YEAR=233,DAY_OF_WEEK=3,
DAY_OF_WEEK_IN_MONTH=3,AM_PM=1,HOUR=11,HOUR_OF_DAY=23,MINUTE=40,SECOND=9,
MILLISECOND=256,ZONE_OFFSET=3600000,DST_OFFSET=3600000]
```

### 11.4.2 Der gregorianische Kalender

Die Klasse GregorianCalendar erweitert die abstrakte Klasse Calendar. Sieben Konstruktoren stehen zur Verfügung; vier davon sehen wir uns an:

```
class java.util.GregorianCalendar
extends Calendar
```

- **GregorianCalendar()**  
Erzeugt ein standardmäßiges GregorianCalendar-Objekt mit der aktuellen Zeit in der voreingestellten Zeitzone und Lokalisierung.
- **GregorianCalendar(int year, int month, int date)**  
Erzeugt ein GregorianCalendar-Objekt in der voreingestellten Zeitzone und Lokalisierung. Jahr, Monat (der zwischen 0 und 11 und nicht zwischen 1 und 12 liegt) und Tag legen das Datum fest.
- **GregorianCalendar(int year, int month, int date, int hour, int minute)**  
Erzeugt ein GregorianCalendar-Objekt in der voreingestellten Zeitzone und Lokalisierung. Das Datum legen Jahr, Monat (0 <= month <= 11), Tag, Stunde und Minute fest.
- **GregorianCalendar(int year, int month, int date, int hour, int minute, int second)**  
Erzeugt ein GregorianCalendar-Objekt in der voreingestellten Zeitzone und Lokalisierung. Das Datum legen Jahr, Monat (0 <= month <= 11), Tag, Stunde, Minute und Sekunde fest.

**Hinweis**

Die Monate beginnen bei 0, sodass new GregorianCalendar(1973, 3, 12) nicht den 12. März, sondern den 12. April ergibt! Damit Anfrageprobleme vermieden werden, sollten die Calendar-Konstanten JANUARY (0), FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER (11) verwendet werden. Die spezielle Variable UNDECIMBER (12) steht für den dreizehnten Monat, der etwa bei einem Mondkalender anzutreffen ist. Die Konstanten sind keine typsicheren Enums, bieten aber den Vorteil, als int einfach mit ihnen zählen zu können.

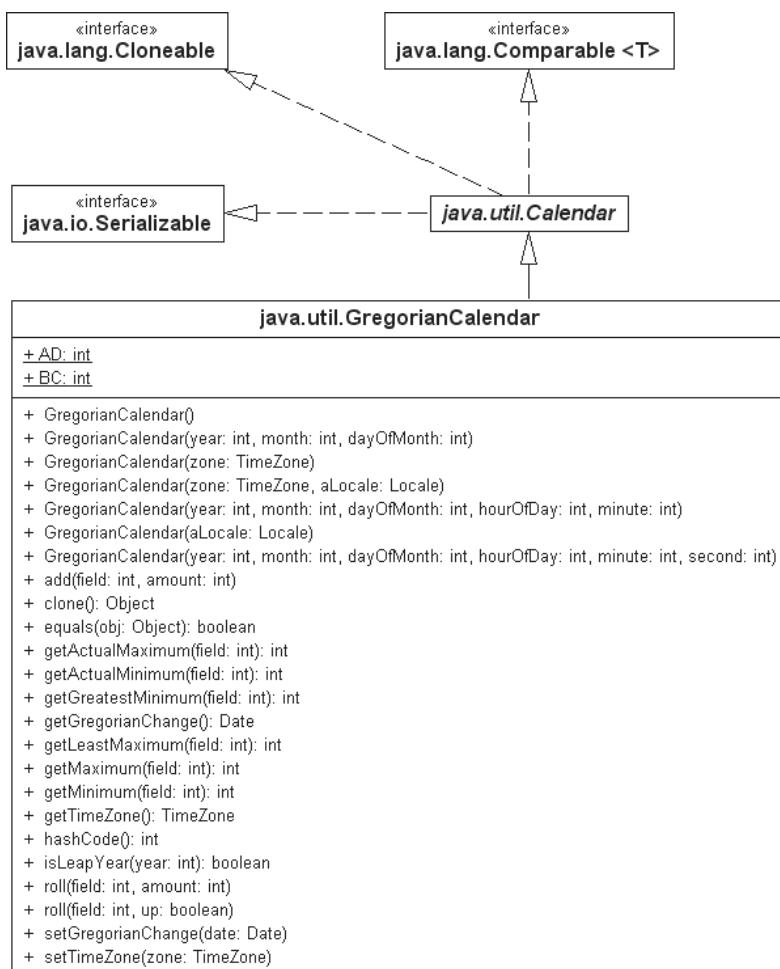


Abbildung 11.3: UML-Diagramm für GregorianCalendar

Neben den hier aufgeführten Konstruktoren gibt es noch weitere, die es erlauben, die Zeitzone und Lokalisierung zu ändern. Standardmäßig eingestellt sind die lokale Zeitzone und die aktuelle Lokalisierung. Ist eines der Argumente im falschen Bereich, löst der Konstruktor eine `IllegalArgumentException` aus.

#### Hinweis

Zum Aufbau von `Calendar`-Objekten gibt es nun zwei Möglichkeiten:

```
Calendar c = Calendar.getInstance();
```

und

```
Calendar c = new GregorianCalendar();
```

Die erste Variante ist besonders in internationalisierter Software zu bevorzugen, da es einige Länder gibt, die nicht nach dem gregorianischen Kalender arbeiten.

```
Calendar c = Calendar.getInstance(new Locale("ja", "JP", "JP"));
```

11

#### 11.4.3 Calendar nach Date und Millisekunden fragen

Da `java.util.Date`-Objekte zwar auf den ersten Blick Konstruktoren anbieten, die Jahr, Monat, Tag entgegennehmen, diese Konstruktoren aber veraltet sind, sollten wir den Blick auf `GregorianCalendar` lenken, wie wir das im vorangehenden Abschnitt gemacht haben.

Um von einem `Calendar` die Anzahl der vergangenen Millisekunden seit dem 1.1.1970 abzufragen, dient `getTimeInMillis()` (eine ähnliche Methode hat auch `Date`, nur heißt sie dort `getTime()`).

#### Beispiel

zB

Bestimme die Anzahl der Tage, die seit einem bestimmten Tag, Monat und Jahr vergangen sind:

```
int date = 1;
int month = Calendar.JANUARY;
int year = 1900;
long ms = new GregorianCalendar(year, month, date).getTimeInMillis();
long days = TimeUnit.MILLISECONDS.toDays(System.currentTimeMillis() - ms);
System.out.println(days); // 40303
```



### Hinweis

Calendar und Date haben beide eine `getTime()`-Methode. Nur liefert die Calendar-Methode `getTime()` ein `java.util.Date`-Objekt und die Date-Methode `getTime()` ein `long`. Gutes API-Design sieht anders aus. Damit Entwickler aber keine unschönen `cal.getTime().getTime()`-Ausdrücke schreiben müssen, um vom Calendar die Anzahl der Millisekunden zu beziehen, ist `getTimeInMillis()` im Angebot.

```
abstract class java.util.Calendar
 implements Serializable, Cloneable, Comparable<Calendar>
```

- `final long getTimeInMillis()`  
Liefert die seit der Epoche (January 1, 1970 00:00:00.000 GMT, Gregorian) vergangene Zeit in Millisekunden.
- `final Date getTime()`  
Liefert ein Date-Objekt zu diesem Calendar.

#### 11.4.4 Abfragen und Setzen von Datumselementen über Feldbezeichner

Das Abfragen und Setzen von Datumselementen des gregorianischen Kalenders erfolgt mit den überladenen Methoden `get()` und `set()`. Beide erwarten als erstes Argument einen Feldbezeichner – eine Konstante aus der Klasse `Calendar` –, der angibt, auf welches Datum-/Zeitfeld zugegriffen werden soll. Die `get()`-Methode liefert den Inhalt des angegebenen Felds, und `set()` schreibt den als zweites Argument übergebenen Wert in das Feld.



### Beispiel

Führe Anweisungen aus, wenn es 19 Uhr ist:

```
if (Calendar.getInstance().get(Calendar.HOUR_OF_DAY) == 19)
 ...
```

Die folgende Tabelle gibt eine Übersicht der Feldbezeichner und ihrer Wertebereiche im Fall des konkreten `GregorianCalendar`.

```
abstract class java.util.Calendar
implements Serializable, Comparable<Calendar>
```

| Feldbezeichner<br>Calendar.*    | Minimalwert     | Maximalwert              | Erklärung                                   |
|---------------------------------|-----------------|--------------------------|---------------------------------------------|
| ERA                             | 0 (BC)          | 1 (AD)                   | Datum vor oder nach Christus                |
| YEAR                            | 1               | theoretisch unbeschränkt | Jahr                                        |
| MONTH                           | 0               | 11                       | Monat (nicht von 1 bis 12!)                 |
| DAY_OF_MONTH<br>alternativ DATE | 1               | 31                       | Tag                                         |
| WEEK_OF_YEAR                    | 1               | 54                       | Woche                                       |
| WEEK_OF_MONTH                   | 1               | 6                        | Woche des Monats                            |
| DAY_OF_YEAR                     | 1               | 366                      | Tag des Jahres                              |
| DAY_OF_WEEK                     | 1               | 7                        | Tag der Woche<br>(1 = Sonntag, 7 = Samstag) |
| DAY_OF_WEEK_IN_MONTH            | 1               | 6                        | Tag der Woche im Monat                      |
| HOUR                            | 0               | 11                       | Stunde von 12                               |
| HOUR_OF_DAY                     | 0               | 23                       | Stunde von 24                               |
| MINUTE                          | 0               | 59                       | Minute                                      |
| SECOND                          | 0               | 59                       | Sekunden                                    |
| MILLISECOND                     | 0               | 999                      | Millisekunden                               |
| AM_PM                           | 0               | 1                        | vor 12, nach 12                             |
| ZONE_OFFSET                     | $13*60*60*1000$ | $+14*60*60*1000$         | Zeitzonenabweichung in Millisekunden        |
| DST_OFFSET                      | 0               | $2*60*60*1000$           | Sommerzeitabweichung in Millisekunden       |

Tabelle 11.2: Konstanten aus der Klasse Calendar

Nun können wir mit den Varianten von `set()` die Felder setzen und mit `get()` wieder herholen. Beachtenswert sind der Anfang der Monate mit 0 und der Anfang der Woche mit 1 (SUNDAY), 2 (MONDAY), ..., 7 (SATURDAY) – Konstanten der Klasse `Calendar` stehen in Klammern. Die Woche beginnt in der Java-Welt also bei 1 und Sonntag, statt – wie vielleicht anzunehmen – bei 0 und Montag.

**zB Beispiel**

Ist ein Date-Objekt gegeben, so speichert es Datum und Zeit. Soll der Zeitanteil gelöscht werden, so bietet Java dafür keine eigene Methode. Die Lösung ist, Stunden, Minuten, Sekunden und Millisekunden von Hand auf 0 zu setzen. Löschen wir vom Hier und Jetzt die Zeit:

```
Date date = new Date();
Calendar cal = Calendar.getInstance();
cal.setTime(date);
cal.set(Calendar.HOUR_OF_DAY, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
date = cal.getTime();
```

Eine Alternative wäre, den Konstruktor GregorianCalendar(int year, int month, int dayOfMonth) mit den Werten vom Datum zu nutzen.

```
abstract class java.util.Calendar
implements Serializable, Cloneable, Comparable<Calendar>
```

- `int get(int field)`  
Liefert den Wert für `field`.
- `void set(int field, int value)`  
Setzt das Feld `field` mit dem Wert `value`.
- `final void set(int year, int month, int date)`  
Setzt die Werte für Jahr, Monat und Tag.
- `final void set(int year, int month, int date, int hourOfDay, int minute)`  
Setzt die Werte für Jahr, Monat, Tag, Stunde und Minute.
- `final void set(int year, int month, int date, int hourOfDay, int minute, int second)`  
Setzt die Werte für Jahr, Monat, Tag, Stunde, Minute und Sekunde.

**Hinweis**

Wo die Date-Klasse etwa spezielle (veralte) Methoden wie `getYear()`, `getDay()`, `getHours()` anbietet, so müssen Nutzer der Calendar-Klasse immer die `get(field)`-Methode nutzen. Es gibt keinen Getter für den Zugriff auf ein bestimmtes Feld.

## Werte relativ setzen

Neben der Möglichkeit, die Werte entweder über den Konstruktor oder über `set()` absolut zu setzen, sind auch relative Veränderungen möglich. Dazu wird die `add()`-Methode eingesetzt, die wie `set()` als erstes Argument einen Feldbezeichner bekommt und als zweites die Verschiebung.

### Beispiel

zB

Was ist der erste und letzte Tag einer Kalenderwoche?

```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.WEEK_OF_YEAR, 15);
cal.set(Calendar.DAY_OF_WEEK, Calendar.MONDAY);
System.out.printf("%tD ", cal); // 04/09/07
cal.add(Calendar.DAY_OF_WEEK, 6);
System.out.printf("%tD", cal); // 04/15/07
```

Die Methode `add()` setzt das Datum um sechs Tage hoch.

11

Da es keine `sub()`-Methode gibt, können die Werte bei `add()` auch negativ sein.

### Beispiel

zB

Wo waren wir heute vor einem Jahr?

```
Calendar cal = Calendar.getInstance();
System.out.printf("%tF%n", cal); // 2006-06-09
cal.add(Calendar.YEAR, -1);
System.out.printf("%tF%n", cal); // 2005-06-09
```

Eine weitere Methode `roll()` ändert keine folgenden Felder, was `add()` macht, wenn etwa zum Dreißigsten eines Monats zehn Tage addiert werden.

```
abstract class java.util.Calendar
implements Serializable, Cloneable, Comparable<Calendar>
```

- `abstract void add(int field, int amount)`

Addiert (bzw. subtrahiert, wenn `amount` negativ ist) den angegeben Wert auf dem (bzw. vom) Feld.

- abstract void roll(int field, boolean up)  
Setzt eine Einheit auf dem gegebenen Feld hoch oder runter, ohne die nachfolgenden Felder zu beeinflussen. Ist der aktuelle Feldwert das Maximum (bzw. Minimum) und wird um eine Einheit addiert (bzw. subtrahiert), ist der nächste Feldwert das Minimum (bzw. Maximum).
- void roll(int field, int amount)  
Ist amount positiv, führt diese Methode die Operation `roll(field, true)` genau amount-mal aus, ist amount negativ, dann wird amount-mal `roll(field, false)` aufgerufen.

In `GregorianCalendar` ist die Implementierung in Wirklichkeit etwas anders. Da ist `roll(int, int)` implementiert, und `roll(int, boolean)` ist ein Aufruf von `roll(field, up ? +1 : -1)`.

## 11.5 Klassenlader (Class Loader)

Ein Klassenlader ist dafür verantwortlich, eine Klasse zu laden. Aus der Datenquelle (im Allgemeinen einer Datei) liefert der Klassenlader ein Byte-Feld mit den Informationen, die im zweiten Schritt dazu verwendet werden, die Klasse im Laufzeitsystem einzubringen; das ist *Linking*. Es gibt eine Reihe von vordefinierten Klassenladern und die Möglichkeit, eigene Klassenlader zu schreiben, um etwa verschlüsselte und komprimierte `.class`-Dateien zu laden.

### 11.5.1 Woher die kleinen Klassen kommen

Nehmen wir zu Beginn ein einfaches Programm mit zwei Klassen:

```
class A
{
 static String s = new java.util.Date().toString();

 public static void main(String[] args)
 {
 B b = new B();
 }
}

class B
```

```
{
 A a;
}
```

Wenn die Laufzeitumgebung das Programm A startet, muss sie eine Reihe von Klassen laden. Sofort wird klar, dass es zumindest A sein muss. Wenn aber die statische `main()`-Methode aufgerufen wird, muss auch B geladen sein. Und da beim Laden einer Klasse auch die statischen Variablen initialisiert werden, wird auch die Klasse `java.util.Date` geladen. Zwei weitere Dinge werden nach einiger Überlegung deutlich:

- Wenn B geladen wird, bezieht es sich auf A. Da A aber schon geladen ist, muss es nicht noch einmal geladen werden.
- Unsichtbar stecken noch andere referenzierte Klassen dahinter, die nicht direkt sichtbar sind. So wird zum Beispiel `Object` geladen werden, da implizit in der Klassendeklaration von A steht: `class A extends Object`.

Im Beispiel mit den Klassen A und B lädt die Laufzeitumgebung selbstständig die Klassen (*implizites Klassenladen*). Klassen lassen sich auch mit `Class.forName()` über ihren Namen laden (*explizites Klassenladen*).

#### Hinweis

Um zu sehen, welche Klassen überhaupt geladen werden, lässt sich der virtuellen Maschine beim Start der Laufzeitumgebung ein Schalter mitgeben – `verbose:class`. Dann gibt die Maschine beim Lauf alle Klassen aus, die sie lädt.

11

## Die Suchorte

Ein festes, dreistufiges Schema bestimmt die Suche nach den Klassen:

1. Klassen wie `String`, `Object` oder `Point` stehen in einem ganz speziellen Archiv. Wenn ein eigenes Java-Programm gestartet wird, so sucht die virtuelle Maschine die angeforderten Klassen zuerst in diesem Archiv. Da es elementare Klassen sind, die zum Hochfahren eines Systems gehören, werden sie *Bootstrap-Klassen* genannt. Das Archiv mit diesen Klassen heißt oft `rt.jar` (für Runtime). Andere Archive können hinzukommen – wie `i18n.jar`, das Internationalisierungsdaten beinhaltet. Die Implementierung dieses Bootstrap-Laders ist nicht öffentlich und wird von System zu System unterschiedlich sein.
2. Findet die Laufzeitumgebung die Klassen nicht bei den Bootstrap-Klassen, so werden alle Archive eines speziellen Verzeichnisses untersucht, das sich *Extension-Verzeich-*

*nis* nennt. Das Verzeichnis gibt es bei jeder Java-Version. Es liegt unter *lib/ext*. Werden hier Klassen eingelagert, so findet die Laufzeitumgebung diese Klassen ohne weitere Anpassung und Setzen von Pfaden. In sonstige Verzeichnisse einer Java-Installation sollten keine Klassen kopiert werden.

3. Ist eine Klasse auch im Erweiterungsverzeichnis nicht zu finden, beginnt die Suche im *Klassenpfad (Classpath)*. Diese Pfadangabe besteht aus einer Aufzählung einzelner Verzeichnisse, Klassen oder Jar-Archive, in denen die Laufzeitumgebung nach den Klassendateien sucht. Standardmäßig ist dieser Klassenpfad auf das aktuelle Verzeichnis gesetzt (»..«).



#### Hinweis

Es gibt spezielle Bootstrap-Klassen, die sich überschreiben lassen. Sie werden in das spezielle Verzeichnis *endorsed* gesetzt. Mehr Informationen dazu folgen in Abschnitt 11.5.6.

### 11.5.2 Setzen des Klassenpfades

Die Suchorte lassen sich angeben, wobei die Bestimmung des Klassenpfades für die eigenen Klassen die wichtigste ist. Sollen in einem Java-Projekt Dateien aus einem Verzeichnis oder einem externen Java-Archiv geholt werden, so ist der übliche Weg, dieses Verzeichnis oder Archiv im Klassenpfad anzugeben. Dafür gibt es zwei Varianten. Die erste ist, über den Schalter *-classpath* (kurz *-cp*) beim Start der virtuellen Maschine die Quellen aufzuführen:

```
$ java -classpath classpath1;classpath2 MainClass
```

Eine Alternative ist das Setzen der Umgebungsvariablen *CLASSPATH* mit einer Zeichenfolge, die die Klassen spezifiziert:

```
$ SET CLASSPATH=classpath1;classpath2
$ java MainClass
```

Ob der Klassenpfad überhaupt gesetzt ist, ermittelt ein einfaches `echo $CLASSPATH` (Unix) beziehungsweise `echo %CLASSPATH%` (Windows).



#### Hinweis

Früher – das heißt vor Java 1.2 – umfasste der *CLASSPATH* auch die Bootstrap-Klassen. Das ist seit 1.2 überflüssig und bedeutet: Die typischen Klassen aus den Paketen `java.*`, `com.sun.*` usw. wie `String` stehen nicht im *CLASSPATH*.

Zur Laufzeit steht dieser Klassenpfad in der Umgebungsvariablen `java.class.path`. Auch die Bootstrap-Klassen können angegeben werden. Dazu dient der Schalter `-Xbootclasspath` oder die Variable `sun.boot.class.path`. Zusätzliche Erweiterungsverzeichnisse lassen sich über die Systemeigenschaft `java.ext.dirs` zuweisen.

**Listing 11.5:** Ausgaben von com/tutego/insel/lang/ClasspathDir.java

| Eigenschaft                      | Beispielbelegung                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>java.class.path</code>     | <code>C:\Insel\programme\1_11_Java_Library\bin</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>java.ext.dirs</code>       | <code>C:\Program Files\Java\jdk1.7.0\jre\lib\ext;</code><br><code>C:\Windows\Sun\Java\lib\ext</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>sun.boot.class.path</code> | <code>C:\Program Files\Java\jdk1.7.0\jre\lib\resources.jar;</code><br><code>C:\Program Files\Java\jdk1.7.0\jre\lib\rt.jar;</code><br><code>C:\Program Files\Java\jdk1.7.0\jre\lib\sunrsasign.jar;</code><br><code>C:\Program Files\Java\jdk1.7.0\jre\lib\jsse.jar;</code><br><code>C:\Program Files\Java\jdk1.7.0\jre\lib\jce.jar;</code><br><code>C:\Program Files\Java\jdk1.7.0\jre\lib\charsets.jar;</code><br><code>C:\Program Files\Java\jdk1.7.0\jre\lib\modules\jdk.boot.jar;</code><br><code>C:\Program Files\Java\jdk1.7.0\jre\classes</code> |

**Tabelle 11.3:** Beispielbelegungen der Variablen

#### Hinweis

Wird die JVM über `java -jar` aufgerufen, beachtet sie nur Klassen in dem genannten Jar und ignoriert den Klassenpfad.

### 11.5.3 Die wichtigsten drei Typen von Klassenladern

Eine Klassendatei kann von der Java-Laufzeitumgebung über verschiedene Klassenlader bezogen werden. Die wichtigsten sind: Bootstrap-, Erweiterungs- und Applikations-Klassenlader. Sie arbeiten insofern zusammen, als dass sie sich gegenseitig Aufgaben zuschieben, wenn eine Klasse nicht gefunden wird:

- **Bootstrap-Klassenlader:** für die Bootstrap-Klassen
- **Erweiterungs-Klassenlader:** für die Klassen im `lib/ext`-Verzeichnis
- **Applikations-Klassenlader (auch System-Klassenlader):** Der letzte Klassenlader im Bunde berücksichtigt bei der Suche den `java.class.path`.

Aus Sicherheitsgründen beginnt der Klassenlader bei einer neuen Klasse immer mit dem System-Klassenlader und reicht dann die Anfrage weiter, wenn er selbst die Klasse nicht laden konnte. Dazu sind die Klassenlader miteinander verbunden. Jeder Klassenlader  $L$  hat dazu einen Vater-Klassenlader  $V$ . Erst darf der Vater versuchen, die Klassen zu laden. Kann er es nicht, gibt er die Arbeit an  $L$  ab.

Hinter dem letzten Klassenlader können wir einen eigenen *benutzerdefinierten Klassenlader* installieren. Auch dieser wird einen Vater haben, den üblicherweise der Applikations-Klassenlader verkörpert.

#### 11.5.4 Die Klasse `java.lang.ClassLoader` \*

Jeder Klassenlader in Java ist vom Typ `java.lang.ClassLoader`. Die Methode `loadClass()` erwartet einen sogenannten »binären Namen«, der an den vollqualifizierten Klassennamen erinnert.

```
abstract class java.lang.ClassLoader
```

- `protected Class<?> loadClass(String name, boolean resolve)`  
Lädt die Klasse und bindet sie mit `resolveClass()` ein, wenn `resolve` gleich `true` ist.
- `public Class<?> loadClass(String name)`  
Die öffentliche Methode ruft `loadClass(name, false)` auf, was bedeutet, dass die Klasse nicht standardmäßig angemeldet (gelinkt) wird. Beide Methoden können eine `ClassNotFoundException` auslösen.

Die geschützte Methode führt anschließend drei Schritte durch:

1. Wird `loadClass()` auf einer Klasse aufgerufen, die dieser Klassenlader schon eingelesen hat, so kehrt die Methode mit dieser gecachten Klasse zurück.
2. Ist die Klasse nicht gespeichert, darf zuerst der Vater (*parent class loader*) versuchen, die Klasse zu laden.
3. Findet der Vater die Klasse nicht, so darf jetzt der Klassenlader selbst mit `findClass()` versuchen, die Klasse zu beziehen.

Eigene Klassenlader überschreiben in der Regel die Methode `findClass()`, um nach einem bestimmten Schema zu suchen, etwa nach Klassen aus der Datenbank. In diesen Stufen ist es auch möglich, höher stehende Klassenlader zu umgehen, was beispielsweise bei Servlets Anwendung findet.

## Neue Klassenlader

Java nutzt an den verschiedensten Stellen spezielle Klassenlader, etwa für Applets den `sun.applet.AppletClassLoader`. Für uns ist der `java.net.URLClassLoader` interessant, da er Klassen von beliebigen URLs laden kann und die Klassen nicht im klassischen Klassenpfad benötigt. Wie ein eigener Klassenlader aussieht, zeigt das Beispiel unter <http://tutego.com/go/urlclassloader>, das den URL-Classloader vom Prinzip her nachimplementiert.

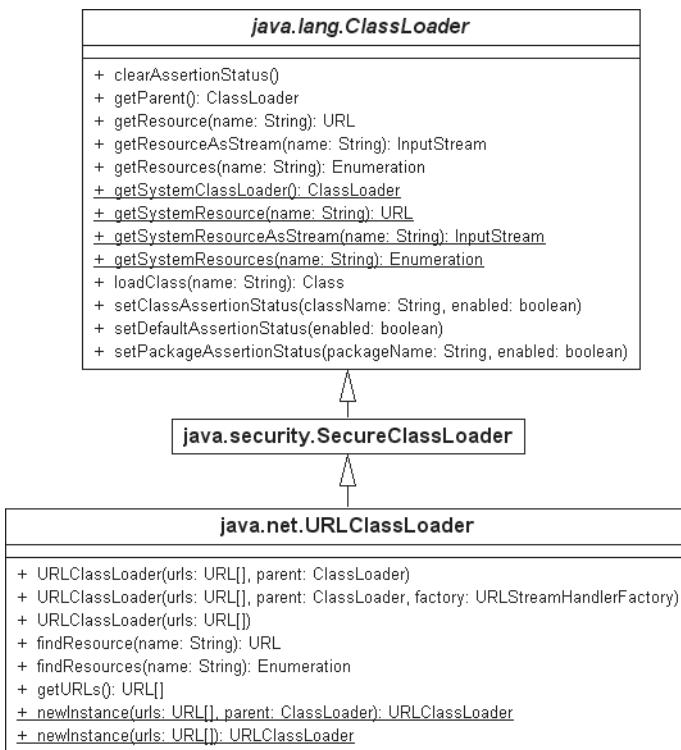


Abbildung 11.4: Klassenhierarchie von URLClassLoader

### 11.5.5 Hot Deployment mit dem URL-ClassLoader \*

Unter *Hot Deployment* ist die Möglichkeit zu verstehen, zur Laufzeit Klassen auszutauschen. Diese Möglichkeit ist für viele EJB- oder Servlet-Container wichtig, da sie im Dateisystem auf eine neue Klassendatei warten und im gegebenen Fall die alte Klasse durch eine neue ersetzen. Ein Servlet-Container überwacht geladene Klassen und lädt sie bei Änderungen neu. Eine Internetsuche mit dem Stichwort *AdaptiveClassLoader* listet Implementierungen auf.

Damit dieser heiße Wechsel funktioniert, muss die Klasse über einen neuen Klassenlader bezogen werden. Das liegt daran, dass der Standardklassenlader von Haus aus keine Klasse mehr loswird, wenn er sie einmal geladen hat. Mit anderen Worten: Wenn eine Klasse über `Class.forName(Klasse)` angefordert wird, ist sie immer im Cache und wird nicht mehr entladen. Ein neuer Klassenlader fängt immer von vorn an, wenn er die Klasse für sich zum ersten Mal sieht.

Mit immer neuen Klassenladern funktioniert das Neuladen, weil für eine neue Klasse dann jeweils ein eigener Klassenlader zuständig ist. Ändert sich die Klasse, wird ein neuer Klassenlader konstruiert, der die neue Klasse lädt. Doch damit ist die alte Klasse noch nicht aus dem Spiel. Nur wenn sich niemand mehr für die alte Klasse und für den Klassenlader interessiert, kann die Laufzeitumgebung diese nicht benutzten Objekte erkennen und aufräumen.

### Gleiche Klasse mehrfach laden

Wir wollen im Folgenden eine eigene statische Methode `newInstance()` vorstellen, die beim Aufruf die neueste Version des Dateisystems lädt und ein Exemplar bildet. Die neu zu ladende Klasse soll ohne Beschränkung der Allgemeinheit einen Standard-Konstruktor haben – andernfalls muss über Reflection ein parametrisierter Konstruktor aufgerufen werden; wir wollen das Beispiel aber kurz halten.

Beginnen wir mit der Klasse, die zweimal geladen werden soll. Sie besitzt einen statischen Initialisierungsblock, der etwas auf der Konsole ausgibt, wenn er beim Laden ausgeführt wird:

**Listing 11.6:** com/tutego/insel/lang/ClassToLoadMultipleTimes.java

```
package com.tutego.insel.lang;
```

```
public class ClassToLoadMultipleTimes
{
 static
 {
 System.out.println("ClassToLoadMultipleTimes");
 }
}
```

Die Testklasse legen wir unter *C:\* ab, und zwar so, dass die Verzeichnisstruktur durch das Paket erhalten bleibt – demnach unter *C:\com\tutego\insel\lang*.

Jetzt brauchen wir noch eine Testklasse, die `ClassToLoadMultipleTimes` unter dem Wurzelverzeichnis liest (also etwa unter `C:/`):

Listing 11.7: com/tutego/insel/lang/LoadClassMultipleTimes.java

```
package com.tutego.insel.lang;

import java.io.File;
import java.net.*;

public class LoadClassMultipleTimes
{
 static Object newInstance(String path, String classname) throws Exception
 {
 URL url = new File(path).toURI().toURL();

 URLClassLoader cl = new URLClassLoader(new URL[]{ url });

 Class<?> c = cl.loadClass(classname);

 return c.newInstance();
 }

 public static void main(String[] args) throws Exception
 {
 newInstance("/", "com.tutego.insel.lang.ClassToLoadMultipleTimes");
 newInstance("/", "com.tutego.insel.lang.ClassToLoadMultipleTimes");
 }
}
```

Nach dem direkten Start ohne Vorbereitung bekommen wir nur einmal die Ausgabe – anders als erwartet. Der Grund liegt in der Hierarchie der Klassenlader. Wichtig ist hier, dass der Standardklassenlader die Klasse `ClassToLoadMultipleTimes` nicht »sehen« darf. Wir müssen die Klasse also aus dem Zugriffspfad der Laufzeitumgebung löschen, da andernfalls aufgrund des niedrigen Rangs unser eigener URL-Klassenlader nicht zum Zuge kommt. (Und ist die Klassendatei nicht im Pfad, können wir das praktische `ClassToLoadMultipleTimes.class.getName()` nicht nutzen.) Erst nach dem Löschen werden wir Zeuge,

wie die virtuelle Maschine auf der Konsole die beiden Meldungen ausgibt, wenn der statische Initialisierungsblock ausgeführt wird.

Die zu ladende Klasse darf nicht den gleichen voll qualifizierten Namen wie eine Standardklasse (etwa `java.lang.String`) tragen. Das liegt daran, dass auch in dem Fall, in dem die Klasse mit dem eigenen `URLClassLoader` bezogen werden soll, die Anfrage trotzdem erst an den System-Klassenlader, dann an den Erweiterungs-Klassenlader und erst ganz zum Schluss an unseren eigenen Klassenlader geht. Es ist also nicht möglich, aus einem Java-Programm Klassen zu beziehen, die prinzipiell vom System-Klassenlader geladen werden. Wir können eine Klasse wie `javax.swing.JButton` nicht selbst beziehen. Wenn sie mit einem Klassenlader ungleich unserem eigenen geladen wird, hat dies wiederum zur Folge, dass wir die geladene Klasse nicht mehr loswerden – was allerdings im Fall der Systemklassen kein Problem sein sollte.

Implementiert die Klasse eine bestimmte Schnittstelle oder erbt sie von einer Basisklasse, lässt sich der Typ der Rückgabe unserer Methode `newInstance()` einschränken. Auf diese Weise ist ein Plugin-Prinzip realisierbar: Die geladene Klasse bietet mit dem Typ Methoden an. Während dieser Typ bekannt ist (der implizite Klassenlader besorgt sie), wird die Klasse selbst erst zur Laufzeit geladen (expliziter Klassenlader).



### Einzigartigkeit eines Singletons

Ein Singleton ist ein Erzeugermuster, das ein Exemplar nur einmal hervorbringt. Singletons finden sich in der JVM an einigen Stellen; so gibt es `java.lang.Runtime` nur einmal, genauso wie `java.awt.Toolkit`. Auch Enums sind Singletons, und so lassen sich die Aufzählungen problemlos mit `==` vergleichen. Und doch gibt es zwischen den Bibliotheks-Singletons und den von Hand gebauten Singleton-Realisierungen und Enums einen großen Unterschied: Sie basieren alle auf statischen Variablen, die dieses eine Exemplar referenzieren. Damit ist eine Schwierigkeit verbunden. Denn wie wir an den Beispielen mit dem `URLClassLoader` gesehen haben, ist dieses Exemplar immer nur pro Klassenlader einzigartig, aber nicht in der gesamten JVM an sich, die eine unbestimmte Anzahl von Klassenladern nutzen kann. Die Enums sind ein gutes Beispiel. In einem Server kann es zwei gleiche `Weekday`-Aufzählungen im gleichen Paket geben. Und doch sind sie völlig unterschiedlich und miteinander inkompatibel, wenn sie zwei unterschiedliche Klassenlader einlesen. Selbst die `Class`-Objekte dieser Enums, die ja auch Singletons innerhalb eines Klassenladers sind, sind bei zwei verschiedenen Klassenladern nicht identisch. Globale Singletons für die gesamte JVM gibt es nicht – zum Glück. Auf der anderen Seite verursachen diese Klassen-Phantome viele Probleme in Java EE-Umgebungen. Doch das ist eine andere Geschichte für ein Java EE-Buch.

```
class java.net.URLClassLoader
extends SecureClassLoader
```

- URLClassLoader(URL[] urls)  
Erzeugt einen neuen URLClassLoader für ein Feld von URLs mit dem Standard-Vater-Klassenlader.
- URLClassLoader(URL[] urls, ClassLoader parent)  
Erzeugt einen neuen URLClassLoader für ein Feld von URLs mit einem gegebenen Vater-Klassenlader.
- protected void addURL(URL url)  
Fügt eine URL hinzu.
- URL[] getURLs()  
Liefert die URLs.

11

### 11.5.6 Das Verzeichnis *jre/lib/endorsed* \*

Im Fall der XML-Parser und weiterer Bibliotheken kommt es häufiger vor, dass sich die Versionen einmal ändern. Es wäre nun müßig, aus diesem Grund die neuen Bibliotheken immer im bootclasspath aufzunehmen, da dann immer eine Einstellung über die Kommandozeile stattfände. Die Entwickler haben daher für spezielle Pakete ein Verzeichnis vorgesehen, in dem Updates eingelagert werden können: das Verzeichnis *jre/lib/endorsed* der Java-Installation. Alternativ können die Klassen und Archive auch durch die Kommandozeilenoption `java.endorsed.dirs` spezifiziert werden.

Wenn der Klassenlader im Verzeichnis *endorsed* eine neue Version – etwa vom XML-Parser – findet, lädt er die Klassen von dort und nicht aus dem Jar-Archiv, aus dem sonst die Klassen geladen würden. Standardmäßig bezieht er die Ressourcen aus der Datei *rt.jar*. Alle im Verzeichnis *endorsed* angegebenen Typen überdecken somit die Standardklassen aus der Java SE; neue Versionen lassen sich einfach einspielen.

Nicht alle Klassen lassen sich mit *endorsed* überdecken. Zum Beispiel lässt sich keine neue Version von `java.lang.String` einfügen. Die Dokumentation »Endorsed Standards Override Mechanism« unter <http://download.oracle.com/javase/7/docs/technotes/guides/standards/> zeigt die überschreibbaren Pakete an: `javax.rmi.CORBA`, `org.omg.*`, `org.w3c.dom` und `org.xml.*`. (Im Übrigen definiert auch Tomcat, die Servlet-Engine, ein solches Überschreibverzeichnis. Hier können Sie Klassen in das Verzeichnis *common/lib/endorsed* aufnehmen, die dann beim Start von Tomcat die Standardklassen überschreiben.)

## 11.6 Die Utility-Klasse System und Properties

In der Klasse `java.lang.System` finden sich Methoden zum Erfragen und Ändern von Systemvariablen, zum Umlenken der Standard-Datenströme, zum Ermitteln der aktuellen Zeit, zum Beenden der Applikation und noch für das ein oder andere. Alle Methoden sind ausschließlich statisch, und ein Exemplar von `System` lässt sich nicht anlegen. In der Klasse `java.lang.Runtime` – die Schnittstelle `RunTime` aus dem CORBA-Paket hat hiermit nichts zu tun – finden sich zusätzlich Hilfsmethoden, wie etwa das Starten von externen Programmen oder Methoden zum Erfragen des Speicherbedarfs. Anders als `System` ist hier nur eine Methode statisch, nämlich die Singleton-Methode `getRuntime()`, die das Exemplar von `Runtime` liefert.

| <code>java.lang.System</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | <code>java.lang.Runtime</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>+ err: PrintStream + in: InputStream + out: PrintStream  + arraycopy(src: Object, srcPos: int, dest: Object, destPos: int, length: int) + clearProperty(key: String): String + console(): Console + currentTimeMillis(): long + exit(status: int) + gc() + getProperties(): Properties + getProperty(key: String, def: String): String + getProperty(key: String): String + getSecurityManager(): SecurityManager + getenv(name: String): String + getenv(): Map + identityHashCode(x: Object): int + inheritedChannel(): Channel + load(filename: String) + loadLibrary(libname: String) + mapLibraryName(libname: String): String + nanoTime(): long + runFinalization() + runFinalizersOnExit(value: boolean) + setErr(err: PrintStream) + setIn(in: InputStream) + setOut(out: PrintStream) + setProperties(props: Properties) + setProperty(key: String, value: String): String + setSecurityManager(s: SecurityManager)</pre> | <pre>+ addShutdownHook(hook: Thread) + availableProcessors(): int + exec(cmdarray: String[], envp: String[], dir: File): Process + exec(command: String, envp: String[], dir: File): Process + exec(command: String): Process + exec(cmdarray: String[]): Process + exec(cmdarray: String[], envp: String[]): Process + exec(command: String, envp: String[]): Process + exit(status: int) + freeMemory(): long + gc() + getLocalizedInputStream(in: InputStream): InputStream + getLocalizedOutputStream(out: OutputStream): OutputStream + getRuntime(): Runtime + halt(status: int) + load(filename: String) + loadLibrary(libname: String) + maxMemory(): long + removeShutdownHook(hook: Thread): boolean + runFinalization() + runFinalizersOnExit(value: boolean) + totalMemory(): long + traceInstructions(on: boolean) + traceMethodCalls(on: boolean)</pre> |

Abbildung 11.5: Eigenschaften der Klassen System und Runtime



### Bemerkung

Insgesamt machen die Klassen `System` und `Runtime` keinen besonders aufgeräumten Eindruck; sie wirken irgendwie so, als sei hier alles zu finden, was an anderer Stelle nicht mehr hineingepasst hat. Auch wären Methoden einer Klasse genauso gut in der anderen Klasse aufgehoben.



### Bemerkung (Forts.)

Dass die statische Methode `System.arraycopy()` zum Kopieren von Feldern nicht in `java.util.Arrays` stationiert ist, lässt sich nur historisch erklären. Und `System.exit()` leitet an `Runtime.getRuntime().exit()` weiter. Einige Methoden sind veraltet beziehungsweise anders verteilt: Das `exec()` von `Runtime` zum Starten von externen Prozessen übernimmt eine neue Klasse `ProcessBuilder`, und die Frage nach dem Speicherzustand oder der Anzahl der Prozessoren beantworten MBeans, wie etwa `ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()`. Aber API-Design ist wie Sex: Eine unüberlegte Aktion, und es lebt mit uns für immer.

### 11.6.1 Systemeigenschaften der Java-Umgebung

Die Java-Umgebung verwaltet Systemeigenschaften wie Pfadtrenner oder die Version der virtuellen Maschine in einem `java.util.Properties`-Objekt. Die statische Methode `System.getProperties()` erfragt diese Systemeigenschaften und liefert das gefüllte `Properties`-Objekt zurück. Zum Erfragen einzelner Eigenschaften ist das `Properties`-Objekt aber nicht unbedingt nötig: `System.getProperty()` erfragt direkt eine Eigenschaft.

11

#### Beispiel

zB

Gib den Namen des Betriebssystems aus:

```
System.out.println(System.getProperty("os.name"));
```

Gib alle Systemeigenschaften auf dem Bildschirm aus:

```
System.getProperties().list(System.out);
```

Die Ausgabe beginnt mit:

```
-- listing properties --
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Program Files\Java\jdk1.7.0\jre\bin
java.vm.version=21.0-b17
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator=;
```

Eine Liste der wichtigen Standard-Systemeigenschaften:

| Schlüssel         | Bedeutung                                                |
|-------------------|----------------------------------------------------------|
| java.version      | Version der Java-Laufzeitumgebung                        |
| java.class.path   | Klassenpfad                                              |
| java.library.path | Pfad für native Bibliotheken                             |
| java.io.tmpdir    | Pfad für temporäre Dateien                               |
| os.name           | Name des Betriebssystems                                 |
| file.separator    | Trenner der Pfadsegmente, etwa / (Unix) oder \ (Windows) |
| path.separator    | Trenner bei Pfadangaben, etwa : (Unix) oder ; (Windows)  |
| line.separator    | Zeilenumbruchzeichen(-folge)                             |
| user.name         | Name des angemeldeten Benutzers                          |
| user.home         | Home-Verzeichnis des Benutzers                           |
| user.dir          | Aktuelles Verzeichnis des Benutzers                      |

Tabelle 11.4: Standardsystemeigenschaften

## API-Dokumentation

Ein paar weitere Schlüssel zählt die API-Dokumentation bei `System.getProperties()` auf. Einige der Variablen sind auch anders zugänglich, etwa über die Klasse `File`.

```
final class java.lang.System
```

- `static String getProperty(String key)`  
Gibt die Belegung einer Systemeigenschaft zurück. Ist der Schlüssel `null` oder leer, gibt es eine `NullPointerException` beziehungsweise eine `IllegalArgumentException`.
- `static String getProperty(String key, String def)`  
Gibt die Belegung einer Systemeigenschaft zurück. Ist sie nicht vorhanden, liefert die Methode die Zeichenkette `def`, den Default-Wert. Für die Ausnahmen gilt das Gleiche wie bei `getProperty(String)`.
- `static String setProperty(String key, String value)`  
Belegt eine Systemeigenschaft neu. Die Rückgabe ist die alte Belegung – oder `null`, falls es keine alte Belegung gab.

- static String clearProperty(String key)  
Löscht eine Systemeigenschaft aus der Liste. Die Rückgabe ist die alte Belegung – oder null, falls es keine alte Belegung gab.
- static Properties getProperties()  
Liefert ein mit den aktuellen Systembelegungen gefülltes Properties-Objekt.

### 11.6.2 line.separator

Um nach dem Ende einer Zeile an den Anfang der nächsten zu gelangen, wird ein *Zeilenumbruch* (engl. *new line*) eingefügt. Das Zeichen für den Zeilenumbruch muss kein einzelnes sein, es können auch mehrere Zeichen nötig sein. Zum Leidwesen der Programmierer unterscheidet sich die Anzahl der Zeichen für den Zeilenumbruch auf den bekannten Architekturen:

- Unix: Line Feed (Zeilenvorschub)
- Windows: beide Zeichen (Carriage Return und Line Feed)
- Macintosh: Carriage Return (Wagenrücklauf)

Der Steuercode für Carriage Return (kurz CR) ist 13 (0x0D), der für Line Feed (kurz LF) 10 (0x0A). Java vergibt obendrein eigene Escape-Sequenzen für diese Zeichen: \r für Carriage Return und \n für Line Feed (die Sequenz \f für einen Form Feed – Seitenvorschub – spielt bei den Zeilenumbrüchen keine Rolle).

Bei der Ausgabe mit einem `println()` oder der Nutzung des Formatspezifizierers %n in `format()` beziehungsweise `printf()` haben wir bei Zeilenumbrüchen keinerlei Probleme. So ist es oft gar nicht nötig, das Zeilenumbruchzeichen vom System über die Property `line.separator` zu erfragen.

### 11.6.3 Eigene Properties von der Konsole aus setzen \*

Eigenschaften lassen sich auch beim Programmstart von der Konsole aus setzen. Dies ist praktisch für eine Konfiguration, die beispielsweise das Verhalten des Programms steuert. In der Kommandozeile werden mit -D der Name der Eigenschaft und nach einem Gleichheitszeichen (ohne Leerzeichen) ihr Wert angegeben. Das sieht dann etwa so aus:

```
$ java -DLOG=-DUSER=Chris -DSIZE=100 com.tutego.insel.lang SetProperty
```

Die Property `LOG` ist einfach nur »da«, aber ohne zugewiesenen Wert. Die nächsten beiden Properties, `USER` und `SIZE`, sind mit Werten verbunden, die erst einmal vom Typ `String` sind und vom Programm weiterverarbeitet werden müssen.

Die Informationen tauchen nicht bei der Argumentliste in der statischen `main()`-Methode auf, da sie vor dem Namen der Klasse stehen und bereits von der Java-Laufzeitumgebung verarbeitet werden.

Um die Eigenschaften auszulesen, nutzen wir das bekannte `System.getProperty()`:

**Listing 11.8:** com/tutego/insel/lang SetProperty.java

```
package com.tutego.insel.lang;

class SetProperty
{
 static public void main(String[] args)
 {
 String logProperty = System.getProperty("LOG");
 String usernameProperty = System.getProperty("USER");
 String sizeProperty = System.getProperty("SIZE");

 System.out.println(logProperty != null); // true

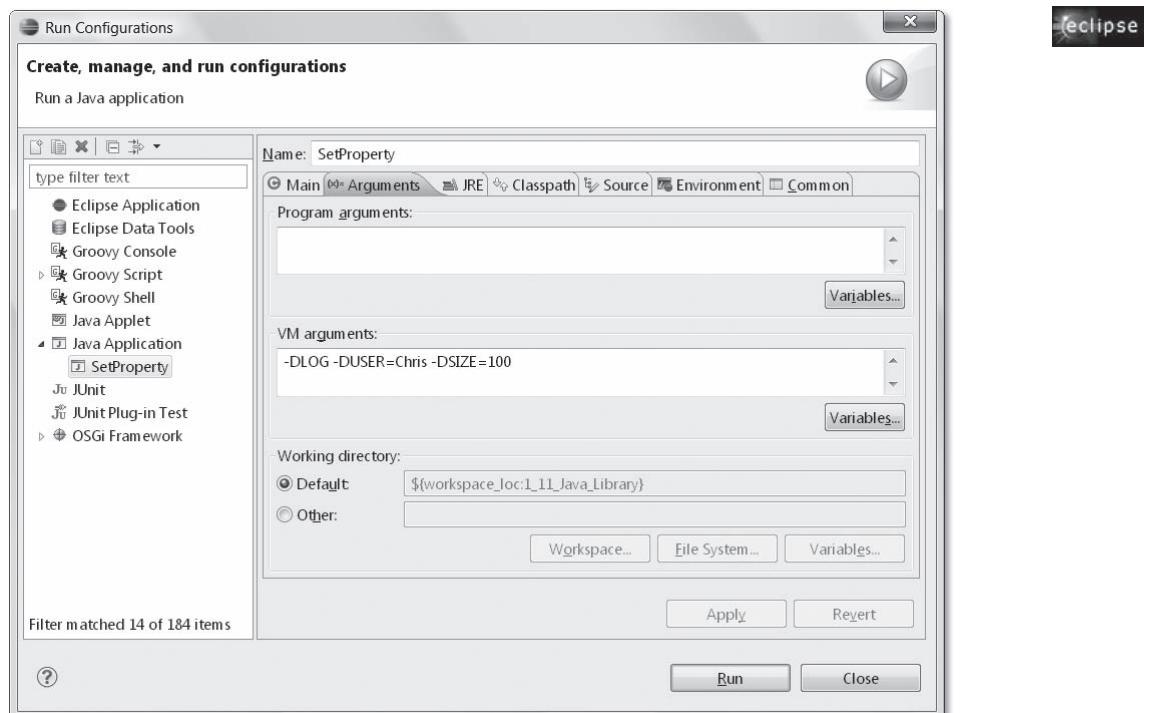
 System.out.println(usernameProperty); // Chris

 if (sizeProperty != null)
 System.out.println(Integer.parseInt(sizeProperty)); // 100

 System.out.println(System.getProperty("DEBUG", "false")); // false
 }
}
```

Wir bekommen über `getProperty()` einen `String` zurück, der den Wert anzeigt. Falls es überhaupt keine Eigenschaft dieses Namens gibt, erhalten wir stattdessen `null`. So wissen wir auch, ob dieser Wert überhaupt gesetzt wurde. Ein einfacher Test wie bei `logProperty != null` sagt also, ob `logProperty` vorhanden ist oder nicht. Statt `-DLOG` führt auch `-DLOG=` zum gleichen Ergebnis, denn der assozierte Wert ist der Leerstring. Da alle Properties erst einmal vom Typ `String` sind, lässt sich `usernameProperty` einfach ausgeben, und wir bekommen entweder `null` oder den hinter `=` angegebenen String. Sind die

Typen keine Strings, müssen sie weiterverarbeitet werden, also etwa mit `Integer.parseInt()`, `Double.parseDouble()` usw. Nützlich ist die Methode `System.getProperty()`, der zwei Argumente übergeben werden, denn das zweite steht für einen Default-Wert. So kann immer ein Standardwert angenommen werden.



**Abbildung 11.6:** Entwicklungsumgebungen erlauben es, die Kommandozeilenargumente in einem Fenster zu setzen. Unter Eclipse gehen wir dazu unter RUN • RUN CONFIGURATIONS, dann zu ARGUMENTS.

### Boolean.getBoolean()

Im Fall von Properties, die mit Wahrheitswerten belegt werden, kann Folgendes geschrieben werden:

```
boolean b = Boolean.parseBoolean(System.getProperty(property)); // (*)
```

Für die Wahrheitswerte gibt es eine andere Variante. Die statische Methode `Boolean.getBoolean(name)` sucht aus den System-Properties eine Eigenschaft mit dem angegebenen Namen heraus. Analog zur Zeile (\*) ist also:

```
boolean b = Boolean.getBoolean(property);
```

Es ist schon erstaunlich, diese statische Methode in der Wrapper-Klasse Boolean anzu treffen, weil Property-Zugriffe nichts mit den Wrapper-Objekten zu tun haben und die Klasse hier eigentlich über ihre Zuständigkeit hinausgeht.

Gegenüber einer eigenen, direkten System-Anfrage hat `getBoolean()` auch den Nachteil, dass wir bei der Rückgabe `false` nicht unterscheiden können, ob es die Eigenschaft schlichtweg nicht gibt oder ob die Eigenschaft mit dem Wert `false` belegt ist. Auch falsch gesetzte Werte wie `-DP=false` ergeben immer `false`.<sup>5</sup>

```
final class java.lang.Boolean
 implements Serializable, Comparable<Boolean>
```

- `static boolean getBoolean(String name)`

Liest eine Systemeigenschaft mit dem Namen `name` aus und liefert `true`, wenn der Wert der Property gleich dem String "true" ist. Die Rückgabe ist `false`, wenn entweder der Wert der Systemeigenschaft "false" ist oder er nicht existiert oder `null` ist.

#### 11.6.4 Umgebungsvariablen des Betriebssystems \*

Fast jedes Betriebssystem nutzt das Konzept der *Umgebungsvariablen* (engl. *environment variables*); bekannt ist etwa `PATH` für den Suchpfad für Applikationen unter Windows und unter Unix. Java macht es möglich, auf diese System-Umgebungsvariablen zuzugreifen. Dazu dienen zwei statische Methoden:

```
final class java.lang.System
```

- `static Map<String, String> getEnv()`

Liest eine Menge von `<String, String>`-Paaren mit allen Systemeigenschaften.

- `static String getEnv(String name)`

Liest eine Systemeigenschaft mit dem Namen `name`. Gibt es sie nicht, ist die Rückgabe `null`.

#### **zB Beispiel**

Was ist der Suchpfad? Den liefert `System.getenv("path")`;

---

<sup>5</sup> Das liegt an der Implementierung: `Boolean.valueOf("false")` liefert genauso `false` wie `Boolean.valueOf("faise")`, `Boolean.valueOf("")` oder `Boolean.valueOf(null)`.

| Name der Variablen | Beschreibung                                        | Beispiel                                           |
|--------------------|-----------------------------------------------------|----------------------------------------------------|
| COMPUTERNAME       | Name des Computers                                  | MOE                                                |
| HOMEDRIVE          | Laufwerksbuchstabe des Benutzerverzeichnisses       | C                                                  |
| HOMEPATH           | Pfad des Benutzerverzeichnisses                     | \Dokumente und Einstellungen\Christian Ullenboom   |
| OS                 | Name des Betriebssystems                            | Windows_NT                                         |
| PATH               | Suchpfad                                            | C:\WINDOWS\system32;<br>C:\WINDOWS                 |
| PATHEXT            | Dateiendungen, die für ausführbare Programme stehen | .COM;.EXE;.BAT;.CMD;.WSH                           |
| SYSTEMDRIVE        | Laufwerksbuchstabe des Betriebssystems              | C                                                  |
| TEMP und auch TMP  | Temporäres Verzeichnis                              | C:\DOKUME~1\CHRIST~1\LOKALE~1\Temp                 |
| USERDOMAIN         | Domäne des Benutzers                                | MOE                                                |
| USERNAME           | Name des Nutzers                                    | Christian Ullenboom                                |
| USERPROFILE        | Profilverzeichnis                                   | C:\Dokumente und Einstellungen\Christian Ullenboom |
| WINDIR             | Verzeichnis des Betriebssystems                     | C:\WINDOWS                                         |

Tabelle 11.5: Auswahl einiger unter Windows verfügbarer Umgebungsvariablen

Einige der Variablen sind auch über die System-Properties (System.getProperties(), System.getProperty()) erreichbar.

### Beispiel

zB

Gib die Umgebungsvariablen des Systems aus. Um die Ausgabe etwas übersichtlicher zu gestalten, ist bei der Aufzählung jedes Komma durch ein Zeilenvorschubzeichen ersetzt worden:

```
Map<String, String> map = System.getenv();
System.out.println(map.toString().replace(',', '\n'));
```

### 11.6.5 Einfache Zeitmessung und Profiling \*

Neben den komfortablen Klassen zum Verwalten von Datumswerten gibt es mit zwei statischen Methoden einfache Möglichkeiten, Zeiten für Programmabschnitte zu messen:

```
final class java.lang.System
{
 static long currentTimeMillis()
 Gibt die seit dem 1.1.1970 vergangenen Millisekunden zurück.

 static long nanoTime()
 Liefert die Zeit vom genauesten System-Zeitgeber. Sie hat keinen Bezugspunkt zu irgendeinem Datum; seit dem 1.1.1970 sind so viele Nanosekunden vergangen, dass sie gar nicht in den long passen würden.
```

Die Differenz zweier Zeitwerte kann zur groben Abschätzung von Ausführungszeiten für Programme dienen:

**Listing 11.9:** com/tutego/insel/lang/Profiling.java

```
package com.tutego.insel.lang;

import static java.util.concurrent.TimeUnit.NANOSECONDS;

class Profiling
{
 private static long[] measure()
 {
 final int MAX = 4000;

 final String string = "Aber Angie, Angie, ist es nicht an der Zeit, Goodbye
 zu sagen? " +
 "Ohne Liebe in unseren Seelen und ohne Geld in unseren
 Mänteln. " +
 "Du kannst nicht sagen, dass wir zufrieden sind.';

 final int number = 123;
 final double nullnummer = 0.0;
```

```
// StringBuffer(size) und append() zur Konkatenation

long time1 = System.nanoTime();

final StringBuilder sb1 = new StringBuilder(MAX * (string.length() + 6));
for (int i = MAX; i-- > 0;)
 sb1.append(string).append(number).append(nullnummer);
sb1.toString();

time1 = NANOSECONDS.toMillis(System.nanoTime() - time1);

// StringBuffer und append() zur Konkatenation

long time2 = System.nanoTime();

final StringBuilder sb2 = new StringBuilder();
for (int i = MAX; i-- > 0;)
 sb2.append(string).append(number).append(nullnummer);
sb2.toString();

time2 = NANOSECONDS.toMillis(System.nanoTime() - time2);

// + zur Konkatenation

long time3 = System.nanoTime();

String t = "";
for (int i = MAX; i-- > 0;)
 t += string + number + nullnummer;

time3 = NANOSECONDS.toMillis(System.nanoTime() - time3);

return new long[] { time1, time2, time3 };
}
```

```

public static void main(String[] args)
{
 measure(); System.gc(); measure(); System.gc();
 long[] durations = measure();

 System.out.printf("sb(size), append(): %d ms%n", durations[0]);
 // sb(size), append(): 2 ms
 System.out.printf("sb(), append() : %d ms%n", durations[1]);
 // sb(), append() : 21 ms
 System.out.printf("t+= : %d ms%n", durations[2]);
 // t+= : 10661 ms
}
}

```

Das Testprogramm hängt Zeichenfolgen mit

- einem `StringBuilder`, der nicht in der Endgröße initialisiert ist,
- einem `StringBuilder`, der eine vorinitialisierte Endgröße nutzt, und
- dem Plus-Operator von `String`s zusammen.

Vor der Messung gibt es zwei Testläufe und ein `System.gc()`, was den Garbage-Collector (GC) anweist, Speicher freizugeben. (Das würde in gewöhnlichen Programmen nicht stehen, da der GC schon selbst ganz gut weiß, wann Speicher freizugeben ist. Nur kostet das Freigeben auch Ausführungszeit, und es würde die Messzeiten beeinflussen, was wir hier nicht wollen.)

Auf meinem Rechner (Intels Core 2 Quad Q6600 (Quadcore), 2,4 GHz, JDK 6) liefert das Programm die Ausgabe:

```

sb(size), append(): 1 ms
sb(), append() : 3 ms
t+= : 39705 ms

```

Das Ergebnis: Bei großen Anhänge-Operationen ist es ein wenig besser, einen passend in der Größe initialisierten `StringBuilder` zu benutzen. Über das + entstehen viele temporäre Objekte, was wirklich teuer kommt. Aber auch, wenn der `StringBuilder` nicht die passende Größe enthält, sind die Differenzen nahezu unbedeutend.

Wo im Programm überhaupt Taktzyklen verbraucht werden, zeigt ein *Profiler*. An diesen Stellen kann dann mit der Optimierung begonnen werden. Eclipse sieht mit dem TPTP

(<http://www.eclipse.org/tptp/>) eine solche Messumgebung vor, und auch <http://code.google.com/a/eclipselabs.org/p/jvmonitor/> ist ein kleines Plugin für Eclipse. NetBeans integriert einen Profiler, Informationen liefert <http://profiler.netbeans.org/>.

## 11.7 Einfache Benutzereingaben

Ein Aufruf von `System.out.println()` gibt Zeichenketten auf der Konsole aus. Für den umgekehrten Weg der Benutzereingabe sind folgende Wege denkbar:

- Statt `System.out` für die Ausgabe lässt sich `System.in` als sogenannter Eingabestrom nutzen. Der allerdings liest nur Bytes und muss für String-Eingaben etwas komfortabler zugänglich gemacht werden. Dazu dient etwa `Scanner`, den Kapitel 4, »Der Umgang mit Zeichenketten«, schon für die Eingabe vorgestellt hat.
- Die Klasse `Console` erlaubt Ausgaben und Eingaben. Die Klasse ist nicht so nützlich, wie es auf den ersten Blick scheint, und eigentlich nur dann wirklich praktisch, wenn passwortgeschützte Eingaben nötig sind.
- Statt der Konsole kann der Benutzer natürlich auch einen grafischen Dialog präsentiert bekommen. Java bietet eine einfache statische Methode für Standardeingaben über einen Dialog an.

### 11.7.1 Grafischer Eingabedialog über JOptionPane

Der Weg über die Befehlszeile ist dabei steinig, da Java eine Eingabe nicht so einfach wie eine Ausgabe vorsieht. Wer dennoch auf Benutzereingaben reagieren möchte, der kann dies über einen grafischen Eingabedialog `JOptionPane` realisieren:

**Listing 11.10:** com/tutego/insel/input/InputWithDialog.java

```
class InputWithDialog
{
 public static void main(String[] args)
 {
 String s = javax.swing.JOptionPane.showInputDialog("Wo kommst du denn wech?");
 System.out.println("Aha, du kommst aus " + s);
 System.exit(0); // Exit program
 }
}
```

Soll die Zeichenkette in eine Zahl konvertiert werden, dann können wir die statische Methode `Integer.parseInt()` nutzen.

### **zB Beispiel**

Zeige einen Eingabedialog an, der zur Zahleneingabe auffordert. Quadriere die eingelesene Zahl, und gib sie auf dem Bildschirm aus:

```
String s = javax.swing.JOptionPane.showInputDialog("Bitte Zahl eingeben");
int i = Integer.parseInt(s);
System.out.println(i * i);
```

Sind Falscheingaben zu erwarten, dann sollte `parseInt()` in einen try-Block gesetzt werden. Bei einer unmöglichen Umwandlung, etwa wenn die Eingabe aus Buchstaben besteht, löst die Methode `parseInt()` eine `NumberFormatException` aus, die – nicht abgefangen – zum Ende des Programms führt.<sup>6</sup>

### **zB Beispiel**

Es soll ein einzelnes Zeichen eingelesen werden:

```
String s = javax.swing.JOptionPane.showInputDialog("Bitte Zeichen eingeben");
char c = 0;
if (s != null && s.length() > 0)
 c = s.charAt(0);
```

### **zB Beispiel**

Ein Wahrheitswert soll eingelesen werden. Dieser Wahrheitswert soll vom Benutzer als Zeichenkette `true` oder `false` beziehungsweise als `1` oder `0` eingegeben werden:

```
String s = javax.swing.JOptionPane.showInputDialog(
 "Bitte Wahrheitswert eingeben");
boolean buh;

if (s != null)
```

---

<sup>6</sup> Oder zumindest zum Ende des Threads.

zB

**Beispiel (Forts.)**

```
if (s.equals("0") || s.equals("false"))
 buh = false;
else if (s.equals("1") || s.equals("true"))
 buh = true;
```

```
class javax.swing.JOptionPane
extends JPanel
implements Accessible
```

- static String showInputDialog( Object message )

Zeigt einen Dialog mit Texteingabezeile. Die Rückgabe ist der eingegebene String oder null, wenn der Dialog abgebrochen wurde. Der Parameter message ist in der Regel ein String.

11

### 11.7.2 Geschützte Passwort-Eingaben mit der Klasse Console \*

Die Klasse `java.io.Console` erlaubt Konsolenausgaben und -eingaben. Ausgangspunkt ist `System.console()`, was ein aktuelles Exemplar liefert – oder null bei einem System ohne Konsolenmöglichkeit. Das `Console`-Objekt ermöglicht übliche Ausgaben und Eingaben und insbesondere mit `readPassword()` eine Möglichkeit zur Eingabe ohne Echo der eingegebenen Zeichen.

Ein Passwort einzulesen und es auf der Konsole auszugeben, sieht so aus:

**Listing 11.11:** com/tutego/insel/io/PasswordFromConsole.java, main()

```
if (System.console() != null)
{
 String passwd = new String(System.console().readPassword());
 System.out.println(passwd);
}
```

```
final class java.lang.System
implements Flushable
```

- static Console console()

Liefert das `Console`-Objekt oder null, wenn es keine Konsole gibt.

```
final class java.io.Console
 implements Flushable
```

- `char[] readPassword()`  
Liest ein Passwort ein, wobei die eingegebenen Zeichen nicht auf der Konsole wiederholt werden.
- `Console format(String fmt, Object... args)`
- `Console printf(String format, Object... args)`  
Ruft `String.format(fmt, args)` auf und gibt den formatierten String auf der Konsole aus.
- `char[] readPassword(String fmt, Object... args)`  
Gibt erst eine formatierte Meldung aus und wartet dann auf die geschützte Passworteingabe.
- `String readLine()`  
Liest eine Zeile von der Konsole und gibt sie zurück.

## 11.8 Ausführen externer Programme \*

Aus Java lassen sich leicht externe Programme aufrufen, etwa Programme des Betriebssystems<sup>7</sup> oder Skripte. Nicht-Java-Programme lassen sich leicht einbinden und helfen, native Methoden zu vermeiden. Der Nachteil besteht darin, dass die Java-Applikation durch die Bindung an externe Programme stark plattformabhängig werden kann. Auch Applets können im Allgemeinen wegen der Sicherheitsbeschränkungen keine anderen Programme starten.

Um die Ausführung anzustoßen, gibt es im Paket `java.lang` zwei Klassen:

- `ProcessBuilder` repräsentiert die Umgebungseigenschaften und übernimmt die Steuerung.
- `Runtime` erzeugt mit `exec()` einen neuen Prozess. Vor Java 5 war dies die einzige Lösung.

---

<sup>7</sup> Wie in C und Unix: `printf("Hello world!\n");system("/bin/rm -rf /&");printf("Bye world!");`

### 11.8.1 ProcessBuilder und Prozesskontrolle mit Process

Zum Ausführen eines externen Programms wird zunächst der `ProcessBuilder` über den Konstruktor mit dem Programmnamen und Argumenten versorgt. Ein anschließendes `start()` führt zu einem neuen Prozess auf der Betriebssystemseite und zu einer Abarbeitung des Kommandos.

```
new ProcessBuilder(kommando).start();
```

Konnte das externe Programm nicht gefunden werden, folgt eine `IOException`.

```
class java.lang.ProcessBuilder
```

- `ProcessBuilder(String... command)`
- `ProcessBuilder(List<String> command)`
- Baut einen neuen `ProcessBuilder` mit einem Programmnamen und einer Liste von Argumenten auf.
- `Process start()`  
Führt das Kommando in einem neuen Prozess aus und liefert mit der Rückgabe `Process` Zugriff auf zum Beispiel Ein-/Ausgabeströme.

#### Hinweis

Die Klasse `ProcessBuilder` gibt es erst seit Java 5. In den vorangehenden Java-Versionen wurden externe Programme mit der Objektmethode `exec()` der Klasse `Runtime` gestartet – ein Objekt vom Typ `Runtime` liefert die Singleton-Methode `getRuntime()`. Für ein Kommando `command` sieht das Starten dann so aus:

```
Runtime.getRuntime().exec(command);
```

#### Ein Objekt vom Typ `Process` übernimmt die Prozesskontrolle

Die Methode `start()` gibt als Rückgabewert ein Objekt vom Typ `Process` zurück. Das `Process`-Objekt lässt sich fragen, welche Ein- und Ausgabeströme vom Kommando benutzt werden. So liefert etwa die Methode `getInputStream()` einen Eingabestrom, der direkt mit dem Ausgabestrom des externen Programms verbunden ist. Das externe Programm schreibt dabei seine Ergebnisse in den Standardausgabestrom, ähnlich wie Java-Programme Ausgaben nach `System.out` senden. Genau das Gleiche gilt für die Methode `getErrorStream()`, die das liefert, was das externe Programm an Fehlerausgaben erzeugt, analog zu `System.err` in Java. Schreiben wir in den Ausgabestrom, den `getOutputStream()`

liefert, so können wir das externe Programm mit eigenen Daten füttern, die es auf seiner Standardeingabe lesen kann. Bei Java-Programmen wäre dies `System.in`. Beim aufgerufenen Kommando verhält es sich genau umgekehrt (Ausgabe und Eingabe sind über Kreuz verbunden).

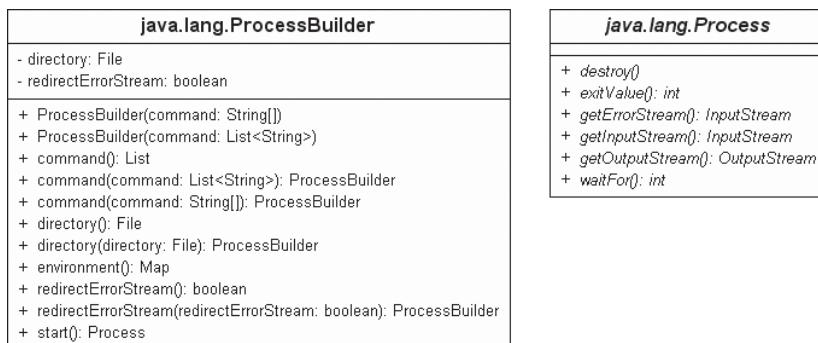


Abbildung 11.7: Klassendiagramm von `ProcessBuilder` und `Process`

### DOS-Programme aufrufen

Da es beim Aufruf von externen Programmen schon eine Bindung an das Betriebssystem gibt, ist auch die Notation für den Aufruf typischer Kommandozeilenprogramme nicht immer gleich. Unter Unix-Systemen ist Folgendes möglich:

```
new ProcessBuilder("rm -rf /bin/laden").start();
```

Das Verfahren, einfach ein bekanntes Konsolenprogramm im String anzugeben, lässt sich nicht ohne Weiteres auf Windows übertragen. Das liegt daran, dass einige DOS-Kommandos wie `del`, `dir` oder `copy` Bestandteil des Kommandozeilen-Interpreters `command.com` sind. Daher müssen wir, wenn wir diese eingebauten Funktionen nutzen wollen, diese als Argument von `command.com` angeben. Für eine Verzeichnisausgabe schreiben wir Folgendes:

```
new ProcessBuilder("cmd", "/c", "dir").start();
```

Einen E-Mail-Client bekommen wir mit:

```
new ProcessBuilder("cmd", "/c", "start", "/B", "mailto:god@163.com").start();
```

Vor der Windows NT-Ära hieß der Interpreter nicht `cmd.exe`, sondern `command.com`.<sup>8</sup>

---

<sup>8</sup> Ein schönes Beispiel für die Plattformabhängigkeit von `exec()`, auch wenn nur Windows 9X und NT gemeint sind.

## Ausgabe der externen Programme verarbeiten

Schreiben die externen Programme in einen Standardausgabekanal, so kann Java diese Ausgabe einlesen. Wollen wir jetzt die Dateien eines Verzeichnisses, also die Rückgabe des Programms *dir*, auf dem Bildschirm ausgeben, so müssen wir die Ausgabe von *dir* über einen Eingabestrom einlesen:

**Listing 11.12:** com/tutego/insel/lang/ExecDir.java, main()

```
ProcessBuilder builder = new ProcessBuilder("cmd", "/c", "dir");
builder.directory(new File("c:/"));
Process p = builder.start();

Scanner s = new Scanner(p.getInputStream()).useDelimiter("\\\z");
System.out.println(s.next());
s.close();
```

abstract class java.lang.Process

- abstract InputStream getInputStream()

Liefert einen Eingabestrom, mit dem sich Daten vom externen Prozess holen lassen, die er in die Standardausgabe schreibt.

## Umgebungsvariablen

Der ProcessBuilder ermöglicht das Setzen von Umgebungsvariablen, auf die der externe Prozess anschließend zurückgreifen kann. Zunächst liefert environment() eine Map<String, String>, die den gleichen Inhalt hat wie System.getenv(). Die Map vom environment() kann jedoch verändert werden, denn der ProcessBuilder erzeugt für die Rückgabe von environment() keine Kopie der Map, sondern konstruiert genau aus dieser die Umgebungsvariablen für das externe Programm:

**Listing 11.13:** com/tutego/insel/lang/ExecWithArguments.java, main()

```
ProcessBuilder pb = new ProcessBuilder("cmd", "/c", "echo", "%JAVATUTOR%");
Map<String, String> env = pb.environment();
env.put("JAVATUTOR", "Christian Ullenboom");
Process p = pb.start();
System.out.println(new Scanner(p.getInputStream()).nextLine());
```

Der Effekt ist gut sichtbar, wenn die Zeile mit env.put() auskommentiert wird.

## Startverzeichnis

Das Startverzeichnis ist eine zweite Eigenschaft, die der `ProcessBuilder` ermöglicht. Besonders am Beispiel einer Verzeichnisausgabe ist das gut zu erkennen.

```
ProcessBuilder builder = new ProcessBuilder("cmd", "/c", "dir");
builder.directory(new File("c:/"));
Process p = builder.start();
```

Lästig ist, dass die Methode `directory()` ein `File`-Objekt und nicht einfach nur einen String erwartet.

```
class java.lang.ProcessBuilder
```

- `File directory()`  
Liefert das aktuelle Verzeichnis des `ProcessBuilder`.
- `ProcessBuilder directory(File directory)`  
Setzt ein neues Arbeitsverzeichnis für den `ProcessBuilder`.
- `Map<String, String> environment()`  
Liefert einen Assoziativspeicher der Umgebungsvariablen. Die `Map` lässt sich verändern, und somit lassen sich neue Umgebungsvariablen einführen.

## Auf das Ende warten

Mit Methoden von `Process` lässt sich der Status des externen Programms erfragen und verändern. Die Methode `waitFor()` lässt den eigenen Thread so lange warten, bis das externe Programm zu Ende ist, oder löst eine `InterruptedException` aus, wenn das gestartete Programm unterbrochen wurde. Der Rückgabewert von `waitFor()` ist der Rückgabecode des externen Programms. Wurde das Programm schon beendet, liefert auch `exitValue()` den Rückgabewert. Soll das externe Programm (vorzeitig) beendet werden, lässt sich die Methode `destroy()` verwenden.

```
abstract class java.lang.Process
```

- `abstract void destroy()`  
Beendet das externe Programm.
- `abstract int exitValue()`  
Wenn das externe Programm beendet wurde, liefert `exitValue()` die Rückgabe des gestarteten Programms. Ist die Rückgabe 0, deutet das auf ein normales Ende hin.

- `abstract void waitFor()`

Wartet auf das Ende des externen Programms (ist es schon beendet, muss nicht gewartet werden) und liefert dann den `exitValue()`.

### Achtung



`waitFor()` wartet ewig, sofern noch Daten abgeholt werden müssen, wenn etwa das externe Programm in den Ausgabestrom schreibt. Ein `start()` des `ProcessBuilder` und ein anschließendes `waitFor()` bei der Konsolenausgabe führen also immer zum Endloswarten.

## Process-Ströme

Ist der Unterprozess über `start()` gestartet, lassen sich über das `Process`-Objekt die Ein-/Ausgabe-Datenströme erfragen. Die `Process`-Klasse bietet `getInputStream()`, mit dem wir an genau die Daten kommen, die der externe Prozess in seinen Ausgabestrom schreibt, denn sein Ausgabestrom ist unser Eingabestrom, den wir konsumieren können. Auch ist `getErrorStream()` ein `InputStream`, denn das, was die externe Anwendung in den Fehlerkanal schreibt, empfangen wir in einem Eingabestrom. Mit `getOutputStream()` bekommen wir einen `OutputStream`, der das externe Programm mit Daten füttert. Dies ist der Pipe-Modus, sodass wir einfach mit externen Programmen Daten austauschen können.

```
abstract class java.lang.Process
```

- `abstract OutputStream getOutputStream()`  
Liefert einen Ausgabestrom, mit dem sich Daten zum externen Prozess schicken lassen, die er über die Standardeingabe empfängt.
- `abstract InputStream getInputStream()`  
Liefert einen Eingabestrom, mit dem sich Daten vom externen Prozess holen lassen, die er in die Standardausgabe schreibt.
- `abstract InputStream getErrorStream()`  
Liefert einen Eingabestrom, mit dem sich Daten vom externen Prozess holen lassen, die er in die Standardfehlerausgabe schreibt.

## Process-Ströme in Dateien umlenken

Neben diesem Pipe-Modus gibt es seit Java 7 eine Alternative, die Ströme direkt auf Dateien umzulenken. Dazu deklariert die `ProcessBuilder`-Klasse diverse `redirectXXX()`-Methoden. (Sollte dann ein `getXXXStream()`-Aufruf gemacht werden, so kommen nicht-

aktive Ströme zurück, denn das externe Programm kommuniziert dann ja direkt mit einer Datei, und die Java-Pipe hängt nicht dazwischen.)

```
class java.lang.ProcessBuilder
```

- ProcessBuilder redirectInput(File file)
- ProcessBuilder redirectInput(ProcessBuilder.Redirect source)
 

Der Unterprozess wird die Eingaben aus der angegebenen Quelle beziehen.
- ProcessBuilder redirectOutput(File file)
- ProcessBuilder redirectOutput(ProcessBuilder.Redirect destination)
 

Der Unterprozess wird Standardausgaben an das angegebene Ziel senden.
- ProcessBuilder redirectError(File file)
- ProcessBuilder redirectError(ProcessBuilder.Redirect destination)
 

Der Unterprozess wird Fehlerausgaben an das angegebene Ziel senden.

Die redirectXXX(File file)-Methoden bekommen als Ziel ein einfaches File-Objekt. Die redirectXXX()-Methoden sind aber mit einem anderen Typ Redirect überladen, der als innere statische Klasse in ProcessBuilder angelegt ist. Mit Redirect.PIPE und Redirect.INHERIT gibt es zwei Konstanten und drei statische Methoden Redirect.from(File), Redirect.to(File), Redirect.appendTo(File), die Redirect-Objekte für die Umleitung zur Datei liefern. Die mit File parametrisierten Methoden greifen auf die Redirect-Klasse zurück, sodass es bei redirectOutput(File file) intern auf ein redirectOutput(Redirect.to(file)) hinausläuft.

### 11.8.2 Einen Browser, E-Mail-Client oder Editor aufrufen

Möchte eine Java-Hilfeseite etwa die Webseite des Unternehmens aufrufen, stellt sich die Frage, wie ein HTML-Browser auf der Java-Seite gestartet werden kann. Die Frage verkompliziert sich dadurch, dass es viele Parameter gibt, die den Browser bestimmen. Welche Plattform: Unix, Windows oder Mac? Soll ein Standardbrowser genutzt werden oder ein bestimmtes Produkt? In welchem Pfad befindet sich die ausführbare Datei des Browsers?

Seit Java 6 ist das über die Klasse `java.awt.Desktop` ganz einfach. Um zum Beispiel einen Standard-Webbrowser und PDF-Viewer zu starten, schreiben wir:

**Listing 11.14:** com/tutego/insel.awt/OpenBrowser.java, main()

```

try
{
 Desktop.getDesktop().browse(new URI("http://www.tutego.de/"));
 Desktop.getDesktop().open(new File("S:/Public.Comp.Lang.Java/3d/Java3D.pdf"));
}
catch (Exception /* IOException, URISyntaxException */ e)
{
 e.printStackTrace();
}

```

Zusammen ergeben sich folgende Objektmethoden:

#### class java.awt.Desktop

- void browse(URI uri)
- void edit(File file)
- void mail()
- void mail(URI mailtoURI)
- void open(File file)
- void print(File file)

Ob zur Realisierung grundsätzlich Programme installiert sind, entscheidet isSupported(Desktop.Action), etwa isSupported(Desktop.Action.OPEN). Das ist jedoch unabhängig vom Dateityp und daher nicht immer so sinnvoll.

#### Tipp

Um unter Windows ein Anzeigeprogramm vor Java 6 zu starten, hilft der Aufruf von rundll32 mit passendem Parameter:

**Listing 11.15:** com/tutego/insel/lang/LaunchBrowser.java, main()

```

String url = "http://www.tutego.de/";
new ProcessBuilder("rundll32", "url.dll,FileProtocolHandler", url).start();

```

Der BrowserLauncher unter <http://browserlaunch2.sourceforge.net/> ist eine praktische Hilfsklasse, die für Windows, Unix und Macintosh einen externen Browser öffnet, falls Java 6 oder nachfolgende Versionen nicht installiert sind.

## 11.9 Benutzereinstellungen \*

Einstellungen des Benutzers – wie die letzten vier geöffneten Dateien oder die Position eines Fensters – müssen abgespeichert und erfragt werden können. Dafür bietet Java eine Reihe von Möglichkeiten. Sie unterscheiden sich unter anderem in dem Punkt, ob die Daten lokal beim Benutzer oder zentral auf einem Server abgelegt sind.

Im lokalen Fall lassen sich die Einstellungen zum Beispiel in einer Datei speichern. Das Dateiformat kann in Textform oder binär sein. In Textform lassen sich die Informationen etwa in der Form *Schlüssel=Wert* oder im XML-Format ablegen. Welche Unterstützung Java in diesem Punkt gibt, zeigen die *Properties*-Klasse (siehe Kapitel 13, »Einführung in Datenstrukturen und Algorithmen«) und die XML-Fähigkeiten der Java-API (siehe Kapitel 16, »Die Einführung in die <XML>-Verarbeitung mit Java«). Werden Datenstrukturen mit den Benutzereinstellungen serialisiert, kommen in der Regel binäre Dateien heraus. Unter Windows gibt es eine andere Möglichkeit der Speicherung: die *Registry*. Auch sie ist eine lokale Datei, nur kann das Java-Programm keinen direkten Zugriff auf die Datei vornehmen, sondern muss über Betriebssystemaufrufe Werte einfügen und erfragen.

Sollen die Daten nicht auf dem Benutzerrechner abgelegt werden, sondern zentral auf einem Server, so gibt es auch verschiedene Standards. Die Daten können zum Beispiel über einen Verzeichnisdienst oder Namensdienst verwaltet werden. Bekanntere Dienste sind hier LDAP oder Active Directory. Zum Zugriff auf die Dienste lässt sich das *Java Naming and Directory Interface* (JNDI) einsetzen. Natürlich können die Daten auch in einer ganz normalen Datenbank stehen, auf die dann die eingebaute JDBC-API Zugriff gewährt. Bei den letzten beiden Formen können die Daten auch lokal vorliegen, denn eine Datenbank oder ein Server, der über JNDI zugänglich ist, kann auch lokal sein. Der Vorteil von nicht-lokalen Servern ist einfach der, dass sich der Benutzer flexibler bewegen kann und immer Zugriff auf seine Daten hat.

Zu guter Letzt lassen sich Einstellungen auch auf der Kommandozeile übergeben. Das lässt die Option `-D` auf der Kommandozeile zu, wenn das Dienstprogramm *java* die JVM startet. Nur lassen sich dann die Daten nicht einfach vom Programm ändern, aber zumindest lassen sich so sehr einfach Daten an das Java-Programm übertragen.

### 11.9.1 Benutzereinstellungen mit der Preferences-API

Mit der Klasse `java.util.prefs.Preferences` können Konfigurationsdateien gespeichert und abgefragt werden. Für die Benutzereinstellungen stehen zwei Gruppen zur Verfügung: die *Benutzerumgebung* und die *Systemumgebung*. Die Benutzerumgebung ist in-

dividuell für jeden Benutzer (jeder Benutzer hat andere Dateien zum letzten Mal geöffnet), aber die Systemumgebung ist global für alle Benutzer. Je nach Betriebssystem verwendet die Preferences-Implementierung unterschiedliche Speichervarianten und Orte:

- Unter Windows wird dazu ein Teilbaum der Registry reserviert. Java-Programme bekommen einen Zweig, *SOFTWARE/JavaSoft/Prefs* unter *HKEY\_LOCAL\_MACHINE* beziehungsweise *HKEY\_CURRENT\_USER* zugewiesen. Es lässt sich nicht auf die gesamte Registry zugreifen!
- Unix und Mac OS X speichern die Einstellungen in XML-Dateien. Die Systemeigenschaften landen bei Unix unter */etc/java/.systemPrefs* und die Benutzereigenschaften lokal unter *\$HOME/.java/.userPrefs*. Mac OS X speichert Benutzereinstellungen im Verzeichnis */Library/Preferences/*.

| <i>java.util.prefs.Preferences</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>+ MAX_KEY_LENGTH: int + MAX_NAME_LENGTH: int + MAX_VALUE_LENGTH: int  + absolutePath(): String + addNodeChangeListener(ncl: NodeChangeListener) + addPreferenceChangeListener(pcl: PreferenceChangeListener) + childrenNames(): String[] + clear() + exportNode(os: OutputStream) + exportSubtree(os: OutputStream) + flush() + get(key: String, def: String): String + getBoolean(key: String, def: boolean): boolean + getByteArray(key: String, def: byte[]): byte[] + getDouble(key: String, def: double): double + getFloat(key: String, def: float): float + getInt(key: String, def: int): int + getLong(key: String, def: long): long + importPreferences(is: InputStream) + isUserNode(): boolean + keys(): String[] + name(): String + node(pathName: String): Preferences + nodeExists(pathName: String): boolean + parent(): Preferences + put(key: String, value: String) + putBoolean(key: String, value: boolean) + putByteArray(key: String, value: byte[]) + putDouble(key: String, value: double) + putFloat(key: String, value: float) + putInt(key: String, value: int) + putLong(key: String, value: long) + remove(key: String) + removeNode() + removeNodeChangeListener(ncl: NodeChangeListener) + removePreferenceChangeListener(pcl: PreferenceChangeListener) + sync() + systemNodeForPackage(c: Class&lt;?&gt;): Preferences + systemRoot(): Preferences + toString(): String + userNodeForPackage(c: Class&lt;?&gt;): Preferences + userRoot(): Preferences</pre> |

Abbildung 11.8: UML-Diagramm Preferences

Preferences-Objekte lassen sich über statische Methoden auf zwei Arten erlangen:

- Die erste Möglichkeit nutzt einen absoluten Pfad zum Registry-Knoten. Die Methoden sind am Preferences-Objekt befestigt und heißen für die Benutzerumgebung `userRoot()` und für die Systemumgebung `systemRoot()`.
- Die zweite Möglichkeit nutzt die Eigenschaft, dass automatisch jede Klasse in eine Paketstruktur eingebunden ist. `userNodeForPackage(Class)` oder `systemNodeForPackage(Class)` liefern ein Preferences-Objekt für eine Verzeichnisstruktur, in der die Klasse selbst liegt.

**zB**

### Beispiel

Erfrage ein Benutzer-Preferences-Objekt über einen absoluten Pfad und über die Paketstruktur der eigenen Klasse:

```
Preferences userPrefs = Preferences.userRoot().node("/com/tutego/insel");
Preferences userPrefs = Preferences.userNodeForPackage(this.getClass());
```

Eine Unterteilung in eine Paketstruktur ist anzuraten, da andernfalls Java-Programme gegenseitig die Einstellung überschreiben könnten; die Registry-Informationen sind für alle sichtbar. Die Einordnung in das Paket der eigenen Klasse ist eine der Möglichkeiten.

```
abstract class java.util.prefs.Preferences
```

- static Preferences `userRoot()`  
Liefert ein Preferences-Objekt für Einstellungen, die lokal für den Benutzer gelten.
- static Preferences `systemRoot()`  
Liefert ein Preferences-Objekt für Einstellungen, die global für alle Benutzer gelten.

### 11.9.2 Einträge einfügen, auslesen und löschen

Die Klasse Preferences hat große Ähnlichkeit mit den Klassen Properties beziehungsweise `HashMap` (vergleiche Kapitel 13, »Einführung in Datenstrukturen und Algorithmen«). Schlüssel/Werte-Paare lassen sich einfügen, löschen und erfragen. Allerdings ist die Klasse Preferences kein Mitglied der Collection-API, und es existiert auch keine Implementierung von Collection-Schnittstellen.

```
abstract class java.util.prefs.Preferences
```

- abstract void `put(String key, String value)`
- abstract void `putBoolean(String key, boolean value)`

- abstract void putByteArray(String key, byte[] value)
- abstract void putDouble(String key, double value)
- abstract void putFloat(String key, float value)
- abstract void.putInt(String key, int value)
- abstract void.putLong(String key, long value)

Bildet eine Assoziation zwischen den Schlüsselnamen und dem Wert. Die Varianten mit den speziellen Datentypen nehmen intern eine einfache String-Umwandlung vor und sind nur kleine Hilfsmethoden; so steht in putDouble() nur put(key, Double.toString(value)). Die Hilfsmethode putByteArray() konvertiert die Daten nach der Base64-Kodierung und legt sie intern als String ab.

- abstract String get(String key, String def)
- abstract boolean.getBoolean(String key, boolean def)
- abstract byte[] getByteArray(String key, byte[] def)
- abstract double.getDouble(String key, double def)
- abstract float.getFloat(String key, float def)
- abstract int.getInt(String key, int def)
- abstract long.getLong(String key, long def)

Liefert den gespeicherten Wert typgerecht aus. Fehlerhafte Konvertierungen werden etwa mit einer NumberFormatException bestraft. Der zweite Parameter erlaubt die Angabe eines Alternativwerts, falls es keinen assoziierten Wert zu dem Schlüssel gibt.

- abstract String[] keys()
- Liefert alle Knoten unter der Wurzel, denen ein Wert zugewiesen wurde. Falls der Knoten keine Eigenschaften hat, liefert keys() ein leeres Feld.
- abstract void flush()
- Die Änderungen werden unverzüglich in den persistenten Speicher geschrieben.

Unser folgendes Programm richtet einen neuen Knoten unter /com/tutego/insel ein. Aus den über System.getProperties() ausgelesenen Systemeigenschaften sollen alle Eigenschaften, die mit »user.« beginnen, in die Registry übernommen werden:

**Listing 11.16:** com/tutego/insel/prefs/PropertiesInRegistry.java, Ausschnitt 1

```
static Preferences prefs = Preferences.userRoot().node("/com/tutego/insel");

static void fillRegistry()
{
```

```

for (Object o : System.getProperties().keySet())
{
 String key = o.toString();

 if (key.startsWith("user.") && System.getProperty(key).length() != 0)
 prefs.put(key, System.getProperty(key));
}
}

```

Um die Elemente auszulesen, kann ein bestimmtes Element mit `getXXX()` erfragt werden. Die Ausgabe aller Elemente unter einem Knoten gelingt am besten mit `keys()`. Das Auslesen kann eine `BackingStoreException` auslösen, falls der Zugriff auf den Knoten nicht möglich ist. Mit `get()` erfragen wir anschließend den mit dem Schlüssel assoziierten Wert. Wir geben »---« aus, falls der Schlüssel keinen assoziierten Wert besitzt:

**Listing 11.17:** com/tutego/insel/prefs/PropertiesInRegistry.java, Ausschnitt 2

```

static void display()
{
 try
 {
 for (String key : prefs.keys())
 System.out.println(key + ": " + prefs.get(key, "---"));
 }
 catch (BackingStoreException e)
 {
 System.err.println("Knoten können nicht ausgelesen werden: " + e);
 }
}

```



### Hinweis

Die Größen der Schlüssel und Werte sind beschränkt! Der Knoten- und Schlüsselname darf maximal `Preferences.MAX_NAME_LENGTH/MAX_KEY_LENGTH` Zeichen umfassen, und die Werte dürfen nicht größer als `MAX_VALUE_LENGTH` sein. Die aktuelle Belegung der Konstanten gibt 80 Zeichen und 8 KiB (8.192 Zeichen) an.

Um Einträge wieder loszuwerden, gibt es drei Methoden: `clear()`, `remove()` und `removeNode()`. Die Namen sprechen für sich.

### 11.9.3 Auslesen der Daten und Schreiben in einem anderen Format

Die Daten aus den Preferences lassen sich mit `exportNode(OutputStream)` beziehungsweise `exportSubtree(OutputStream)` im UTF-8-kodierten XML-Format in einen Ausgabestrom schreiben. `exportNode(OutputStream)` speichert nur einen Knoten, und `exportSubtree(OutputStream)` speichert den Knoten inklusive seiner Kinder. Und auch der umgekehrte Weg funktioniert: `importPreferences(InputStream)` importiert Teile in die Registrierung. Die Schreib- und Lesemethoden lösen eine `IOException` bei Fehlern aus, und eine `InvalidPreferencesFormatException` ist beim Lesen möglich, wenn die XML-Daten ein falsches Format haben.

### 11.9.4 Auf Ereignisse horchen

Änderungen an den Preferences lassen sich mit Listenern verfolgen. Zwei sind im Angebot:

- Der `NodeChangeListener` reagiert auf Einfüge- und Löschoperationen von Knoten.
- Der `PreferenceChangeListener` informiert bei Wertänderungen.

Es ist nicht gesagt, dass, wenn andere Applikationen die Einstellungen ändern, diese Änderungen vom Java-Programm auch erkannt werden.

Eine eigene Klasse `NodePreferenceChangeListener` soll die beiden Schnittstellen `NodeChangeListener` und `PreferenceChangeListener` implementieren und auf der Konsole die erkannten Änderungen ausgeben.

**Listing 11.18:** com/tutego/insel/prefs/ NodePreferenceChangeListener.java,  
`NodePreferenceChangeListener`

```
class NodePreferenceChangeListener implements
 NodeChangeListener, PreferenceChangeListener
{
 /* (non-Javadoc)
 * @see java.util.prefs.NodeChangeListener#childAdded(java.util.prefs.NodeChangeEvent)
 */
 @Override public void childAdded(NodeChangeEvent e)
 {
 Preferences parent = e.getParent(), child = e.getChild();
```

```

 System.out.println(parent.name() + " hat neuen Knoten " + child.name());
 }

/* (non-Javadoc)
 * @see java.util.prefs.NodeChangeListener#childRemoved
 * (java.util.prefs.NodeChangeEvent)
 */
@Override public void childRemoved(NodeChangeEvent e)
{
 Preferences parent = e.getParent(), child = e.getChild();

 System.out.println(parent.name() + " verliert Knoten " + child.name());
}

/* (non-Javadoc)
 * @see java.util.prefs.PreferenceChangeListener#preferenceChange
 * (java.util.prefs.PreferenceChangeEvent)
 */
@Override public void preferenceChange(PreferenceChangeEvent e)
{
 String key = e.getKey(), value = e.getNewValue();

 Preferences node = e.getNode();

 System.out.println(node.name() + " hat neuen Wert " + value + " für " + key);
}
}

```

Zum Anmelden eines Listeners bietet Preferences **zwei** addXXXChangeListener()-Methoden:

**Listing 11.19:** com/tutego/insel/prefs/PropertiesInRegistry.java, addListener()

```

NodePreferenceChangeListener listener = new NodePreferenceChangeListener();
prefs.addNodeChangeListener(listener);
prefs.addPreferenceChangeListener(listener);

```

### 11.9.5 Zugriff auf die gesamte Windows-Registry

Wird Java unter MS Windows ausgeführt, so ergibt sich hin und wieder die Aufgabe, Eigenschaften der Windows-Umgebung zu kontrollieren. Viele Eigenschaften des Windows-Betriebssystems sind in der Registry versteckt, und Java bietet als plattformunabhängige Sprache keine Möglichkeit, diese Eigenschaften in der Registry auszulesen oder zu verändern. (Die Schnittstelle `java.rmi.registry.Registry` ist eine Zentrale für entfernte Aufrufe und hat mit der Windows-Registry nichts zu tun. Auch das Paket `java.util.prefs` mit der Klasse `Preferences` erlaubt nur Modifikationen an einem ausgewählten Teil der Windows-Registry.)

Um von Java aus auf alle Teile der Windows-Registry zuzugreifen, gibt es mehrere Möglichkeiten, unter anderem:

- Um auf allen Werten der Windows-Registry, die dem Benutzer zugänglich sind, operieren zu können, lässt sich mit einem Trick ab Java 1.4 eine Klasse nutzen, die `Preferences` unter Windows realisiert: `java.util.prefs.WindowsPreferences`. Damit ist keine zusätzliche native Implementierung – und damit eine Windows-DLL im Klassennpfad – nötig. Die Bibliothek <https://sourceforge.net/projects/jregistrykey/> realisiert eine solche Lösung.
- eine native Bibliothek, wie das *Windows Registry API Native Interface* (<http://tutego.com/go/jnireg>), die frei zu benutzen ist und unter keiner besonderen Lizenz steht
- das Aufrufen des Konsolenregistrierungsprogramms `reg` zum Setzen und Abfragen von Schlüsselwerten

### Registry-Zugriff selbst gebaut

Für einfache Anfragen lässt sich der Registry-Zugriff schnell auch von Hand erledigen. Dazu rufen wir einfach das Kommandozeilenprogramm `reg` auf, um etwa den Dateinamen für den Desktop-Hintergrund anzuzeigen:

```
$ reg query "HKEY_CURRENT_USER\Control Panel\Desktop" /v Wallpaper
! REG.EXE VERSION 3.0

HKEY_CURRENT_USER\Control Panel\Desktop
Wallpaper REG_SZ C:\Dokumente und Einstellungen\tutego\Anwendungsdaten\Hg.bmp
```

Wenn wir *reg* von Java aufrufen, haben wir den gleichen Effekt:

**Listing 11.20:** com/tutego/insel/lang/JavaWinReg.java, main()

```
ProcessBuilder builder = new ProcessBuilder(
 "reg", "query",
 "\"HKEY_CURRENT_USER\\Control Panel\\Desktop\"", "/v", "Wallpaper");
Process p = builder.start();
Scanner scanner = new Scanner(p.getInputStream())
 .useDelimiter(" \\\\w+\\\\s+\\\\w+\\\\s+");
scanner.next();
System.out.println(scanner.next());
```

## 11.10 Zum Weiterlesen

Die Java-Bibliothek bietet zwar reichlich Klassen und Methoden, aber nicht immer das, was das aktuelle Projekt gerade benötigt. Die Lösung von Problemen, wie etwa Aufbau und Konfiguration von Java-Projekten, objekt-relationalen Mappern (<http://www.hibernate.org/>) oder Kommandozeilenparsern, liegt in diversen kommerziellen oder quelloffenen Bibliotheken und Frameworks. Während bei eingekauften Produkten die Lizenzfrage offensichtlich ist, ist bei quelloffenen Produkten eine Integration in das eigene Closed-Source-Projekt nicht immer selbstverständlich. Diverse Lizenzformen (<http://opensource.org/licenses/>) bei Open-Source-Software mit immer unterschiedlichen Vorgaben – Quellcode veränderbar, Derivate müssen frei sein, Vermischung mit proprietärer Software möglich – erschweren die Auswahl, und Verstöße (<http://gpl-violations.org/>) werden öffentlich angeprangert und sind unangenehm. Java-Entwickler sollten für den kommerziellen Vertrieb ihr Augenmerk verstärkt auf Software unter der BSD-Lizenz (die Apache-Lizenz gehört in diese Gruppe) und unter der LGPL-Lizenz richten. Die Apache-Gruppe hat mit den *Jakarta Commons* (<http://jakarta.apache.org/commons/>) eine hübsche Sammlung an Klassen und Methoden zusammengetragen, und das Studium der Quellen sollte für Softwareentwickler mehr zum Alltag gehören. Die Webseite <http://koders.com/> eignet sich dafür außerordentlich gut, da sie eine Suche über bestimmte Stichwörter durch mehr als 1 Milliarde Quellcodezeilen verschiedener Programmiersprachen ermöglicht; erstaunlich, wie viele Entwickler »F\*ck« schreiben. Und »Porn Groove« kannte ich vor dieser Suche auch noch nicht.



## Kapitel 12

# Einführung in die nebenläufige Programmierung

»Just Be.«

– Calvin Klein (\* 1942)

## 12.1 Nebenläufigkeit

12

Moderne Betriebssysteme geben dem Benutzer die Illusion, dass verschiedene Programme gleichzeitig ausgeführt werden – die Betriebssysteme nennen sich *multitaskingfähig*. Was wir dann wahrnehmen, ist eine Quasiparallelität, die im Deutschen auch *Nebenläufigkeit*<sup>1</sup> genannt wird. Diese Nebenläufigkeit der Programme wird durch das Betriebssystem gewährleistet, das auf Einprozessormaschinen die Prozesse alle paar Millisekunden umschaltet. Daher ist das Programm nicht wirklich parallel, sondern das Betriebssystem gaukelt uns dies durch eine verzahnte Bearbeitung der Prozesse vor. Wenn mehrere Prozessoren oder mehrere Prozessor-Kerne am Werke sind, werden die Programmteile tatsächlich parallel abgearbeitet. Aber ob nur ein kleines Männchen oder beliebig viele im Rechner arbeiten, soll uns egal sein.

Der Teil des Betriebssystems, der die Umschaltung übernimmt, heißt *Scheduler*. Die dem Betriebssystem bekannten aktiven Programme bestehen aus Prozessen. Ein Prozess setzt sich aus dem Programmcode und den Daten zusammen und besitzt einen eigenen Adressraum. Des Weiteren gehören Ressourcen wie geöffnete Dateien oder belegte Schnittstellen dazu. Die virtuelle Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse. Dadurch ist es nicht möglich, dass ein Prozess den Speicherraum eines anderen Prozesses korrumptiert; er sieht den anderen Speicherbereich nicht. Damit Prozesse untereinander Daten austauschen können, wird ein besonderer Speicherbereich als *Shared Memory* markiert. Amok laufende Programme sind zwar möglich, werden jedoch vom Betriebssystem gestoppt.

---

<sup>1</sup> Mitunter sind die Begriffe *parallel* und *nebenläufig* nicht äquivalent definiert. Wir wollen sie in diesem Zusammenhang aber synonym benutzen.

### 12.1.1 Threads und Prozesse

Bei modernen Betriebssystemen gehört zu jedem Prozess mindestens ein *Thread* (zu Deutsch *Faden* oder *Ausführungsstrang*), der den Programmcode ausführt. Damit werden also genau genommen die Prozesse nicht mehr parallel ausgeführt, sondern nur die Threads. Innerhalb eines Prozesses kann es mehrere Threads geben, die alle zusammen in demselben Adressraum ablaufen. Die einzelnen Threads eines Prozesses können untereinander auf ihre öffentlichen Daten zugreifen.

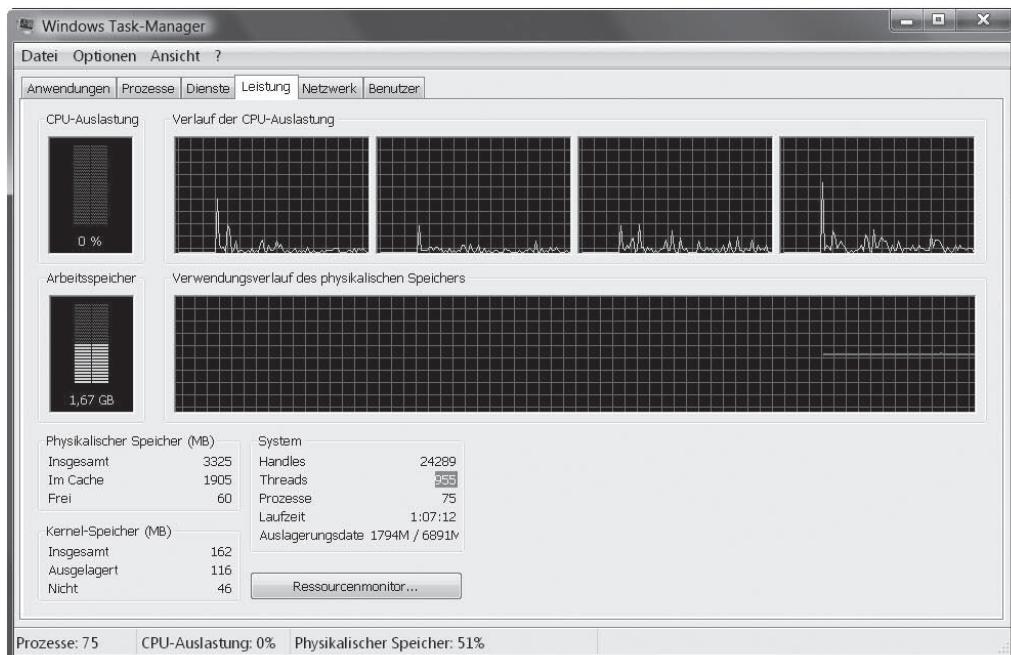


Abbildung 12.1: Windows zeigt im Task-Manager die Anzahl laufender Threads an.

Die Programmierung von Threads ist in Java einfach möglich, und die quasi parallel ablaufenden Aktivitäten geben dem Benutzer den Eindruck von Gleichzeitigkeit. In Java ist auch *multithreaded* Software möglich, wenn das Betriebssystem des Rechners keine Threads direkt verwendet. In diesem Fall simuliert die virtuelle Maschine die Parallelität, indem sie die Synchronisation und die verzahnte Ausführung regelt. Unterstützt das Betriebssystem Threads direkt, bildet die JVM die Thread-Verwaltung in der Regel auf das Betriebssystem ab. Dann haben wir es mit *nativen Threads* zu tun. Die 1:1-Abbildung ermöglicht eine einfache Verteilung auf Mehrprozessorsystemen, doch mit dem Nachteil, dass das Betriebssystem in den Threads auch Bibliotheksaufrufe ausführen kann, zum Beispiel, um das Ein- und Ausgabesystem zu verwenden oder für grafische

Ausgaben. Damit dies ohne Probleme funktioniert, müssen die Bibliotheken jedoch thread-sicher sein. Damit hatten die Unix-Versionen in der Vergangenheit Probleme: Insbesondere die grafischen Standardbibliotheken *X11* und *Motif* waren lange nicht thread-sicher. Um schwerwiegenden Problemen mit grafischen Oberflächen aus dem Weg zu gehen, haben die Entwickler daher auf eine native Multithreading-Umgebung zunächst verzichtet.

Ob die Laufzeitumgebung native Threads nutzt oder nicht, steht nicht in der Spezifikation der JVM. Auch die Sprachdefinition lässt bewusst die Art der Implementierung frei. Was die Sprache jedoch garantieren kann, ist die korrekt verzahnte Ausführung. Hier können Probleme auftreten, die Datenbankfreunde von Transaktionen her kennen. Es besteht die Gefahr konkurrierender Zugriffe auf gemeinsam genutzte Ressourcen. Um dies zu vermeiden, kann der Programmierer durch synchronisierte Programmblöcke einen gegenseitigen Ausschluss sicherstellen. Damit steigt aber auch die Gefahr von *Verklemmungen* (engl. *deadlocks*), die der Entwickler selbst vermeiden muss.

### 12.1.2 Wie parallele Programme die Geschwindigkeit steigern können

Auf den ersten Blick ist es nicht ersichtlich, warum auf einem Einprozessorsystem die nebenläufige Abarbeitung eines Programms geschwindigkeitssteigernd sein kann. Betrachten wir daher ein Programm, das eine Folge von Anweisungen ausführt. Die Programmsequenz dient zum Visualisieren eines Datenbank-Reports. Zunächst wird ein Fenster zur Fortschrittsanzeige dargestellt. Anschließend werden die Daten analysiert und der Fortschrittsbalken kontinuierlich aktualisiert. Schließlich werden die Ergebnisse in eine Datei geschrieben. Die Schritte sind:

1. Baue ein Fenster auf.
2. Öffne die Datenbank vom Netz-Server, und lies die Datensätze.
3. Analysiere die Daten, und visualisiere den Fortschritt.
4. Öffne die Datei, und schreibe den erstellten Report.

Was auf den ersten Blick wie ein typisches sequenzielles Programm aussieht, kann durch geschickte Parallelisierung beschleunigt werden.

Damit dies besser zu verstehen ist, ziehen wir noch einmal den Vergleich mit Prozessen. Nehmen wir an, auf einer Einprozessormaschine sind fünf Benutzer angemeldet, die im Editor Quelltext tippen und hin und wieder den Java-Compiler bemühen. Die Benutzer bekämen vermutlich die Belastung des Systems durch die anderen nicht mit, denn Editor-Operationen lasten den Prozessor nicht aus. Wenn Dateien kompiliert und somit

vom Hintergrundspeicher in den Hauptspeicher transferiert werden, ist der Prozessor schon besser ausgelastet, doch geschieht dies nicht regelmäßig. Im Idealfall übersetzen alle Benutzer nur dann, wenn die anderen gerade nicht übersetzen – im schlechtesten Fall möchten natürlich alle Benutzer gleichzeitig übersetzen.

Übertragen wir die Verteilung auf unser Problem, nämlich wie der Datenbank-Report schneller zusammengestellt werden kann. Beginnen wir mit der Überlegung, welche Operationen parallel ausgeführt werden können:

- Das Öffnen des Fensters der Ausgabedatei und das Öffnen der Datenbank kann parallel geschehen.
- Das Lesen neuer Datensätze und das Analysieren alter Daten kann gleichzeitig erfolgen.
- Alte analysierte Werte können während der neuen Analyse in die Datei geschrieben werden.

Wenn die Operationen wirklich parallel ausgeführt werden, lässt sich bei Mehrprozessorsystemen ein enormer Leistungszuwachs verzeichnen. Doch interessanterweise ergibt sich dieser auch bei nur einem Prozessor, was in den Aufgaben begründet liegt. Denn bei den gleichzeitig auszuführenden Aufgaben handelt es sich um unterschiedliche Ressourcen. Wenn die grafische Oberfläche das Fenster aufbaut, braucht sie dazu natürlich Rechenzeit. Parallel kann die Datei geöffnet werden, wobei weniger Prozessorleistung gefragt ist, da die vergleichsweise träge Festplatte angesprochen wird. Das Öffnen der Datenbank wird auf den Datenbank-Server im Netzwerk abgewälzt. Die Geschwindigkeit hängt von der Belastung des Servers und des Netzes ab. Wenn anschließend die Daten gelesen werden, muss die Verbindung zum Datenbank-Server natürlich stehen. Daher sollten wir zuerst die Verbindung aufbauen.

Ist die Verbindung hergestellt, lassen sich über das Netzwerk Daten in einen Puffer holen. Der Prozessor wird nicht belastet, vielmehr der Server auf der Gegenseite und das Netzwerk. Während der Prozessor also vor sich hin döst und sich langweilt, können wir ihn besser beschäftigen, indem er alte Daten analysiert. Wir verwenden hierfür zwei Puffer: In den einen lädt ein Thread die Daten, während ein zweiter Thread die Daten im anderen Puffer analysiert. Dann werden die Rollen der beiden Puffer getauscht. Jetzt ist der Prozessor beschäftigt. Er ist aber vermutlich fertig, bevor die neuen Daten über das Netzwerk eingetroffen sind. In der Zwischenzeit können die Report-Daten in den Report geschrieben werden; eine Aufgabe, die wieder die Festplatte belastet und weniger den Prozessor.

Wir sehen an diesem Beispiel, dass durch hohe Parallelisierung eine Leistungssteigerung möglich ist, da die bei langsamem Operationen anfallenden Wartezeiten genutzt werden können. Langsame Arbeitsschritte lasten den Prozessor nicht aus, und die Wartezeit, die für den Prozessor beim Netzwerkzugriff auf eine Datenbank anfällt, kann für andere Aktivitäten genutzt werden. Die Tabelle gibt die Elemente zum Kombinieren noch einmal an:

| Ressource             | Belastung                  |
|-----------------------|----------------------------|
| Hauptspeicherzugriffe | Prozessor                  |
| Dateioperationen      | Festplatte                 |
| Datenbankzugriff      | Server, Netzwerkverbindung |

Tabelle 12.1: Parallelisierbare Ressourcen

Das Beispiel macht auch deutlich, dass die Nebenläufigkeit gut geplant werden muss. Nur wenn verzahnte Aktivitäten unterschiedliche Ressourcen verwenden, resultiert daraus auf Einprozessorsystemen ein Geschwindigkeitsvorteil. Daher ist ein paralleler Sortieralgorithmus nicht sinnvoll. Das zweite Problem ist die zusätzliche Synchronisation, die das Programmieren erschwert. Wir müssen auf das Ergebnis einer Operation warten, damit wir mit der Bearbeitung fortfahren können.

### 12.1.3 Was Java für Nebenläufigkeit alles bietet

Für nebenläufige Programme sieht die Java-Bibliothek eine Reihe von Klassen, Schnittstellen und Aufzählungen vor:

- Thread: Jeder laufende Thread ist ein Exemplar dieser Klasse.
- Runnable: Beschreibt den Programmcode, den die JVM parallel ausführen soll.
- Lock: Dient zum Markieren von kritischen Abschnitten, in denen sich nur ein Thread befinden darf.
- Condition: Threads können auf die Benachrichtigung anderer Threads warten.

## 12.2 Threads erzeugen

Die folgenden Abschnitte verdeutlichen, wie der nebenläufige Programmcode in einen Runnable verpackt und dem Thread zur Ausführung vorgelegt wird.

### 12.2.1 Threads über die Schnittstelle Runnable implementieren

Damit der Thread weiß, was er ausführen soll, müssen wir ihm Anweisungsfolgen geben. Diese werden in einem Befehlsobjekt vom Typ `Runnable` verpackt und dem Thread übergeben. Wird der Thread gestartet, arbeitet er die Programmzeilen aus dem Befehlsobjekt parallel zum restlichen Programmcode ab. Die Schnittstelle `Runnable` ist schmal und schreibt nur eine `run()`-Methode vor.

```
interface java.lang.Runnable
```

- `void run()`

Implementierende Klassen realisieren die Operation und setzen dort den parallel auszuführenden Programmcode ein.

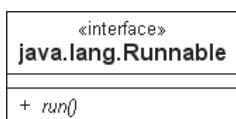


Abbildung 12.2: UML-Diagramm der einfachen Schnittstelle Runnable

Wir wollen zwei Threads angeben, wobei einer zwanzigmal das aktuelle Datum und die Uhrzeit ausgibt und der andere einfach eine Zahl:

**Listing 12.1:** com/tutego/insel/thread/DateCommand.java

```
package com.tutego.insel.thread;

public class DateCommand implements Runnable
{
 @Override public void run()
 {
 for (int i = 0; i < 20; i++)
 System.out.println(new java.util.Date());
 }
}
```

**Listing 12.2:** com/tutego/insel/thread/CounterCommand.java

```
package com.tutego.insel.thread;

class CounterCommand implements Runnable
{
```

```

@Override public void run()
{
 for (int i = 0; i < 20; i++)
 System.out.println(i);
}

```

Unser parallel auszuführender Programmcode in `run()` besteht aus einer Schleife, die in einem Fall ein aktuelles Date-Objekt ausgibt und im anderen Fall einen Schleifenzähler.

### 12.2.2 Thread mit Runnable starten

Nun reicht es nicht aus, einfach die `run()`-Methode einer Klasse direkt aufzurufen, denn dann wäre nichts nebenläufig, sondern wir würden einfach eine Methode sequenziell ausführen. Damit der Programmcode parallel zur Applikation läuft, müssen wir ein Thread-Objekt mit dem Runnable verbinden und dann den Thread explizit starten. Dazu übergeben wir dem Konstruktor der Klasse `Thread` eine Referenz auf das Runnable-Objekt und rufen `start()` auf. Nachdem `start()` für den Thread eine Ablaufumgebung geschaffen hat, ruft es intern selbstständig die Methode `run()` genau einmal auf. Läuft der Thread schon, so löst ein zweiter Aufruf der `start()`-Methode eine `IllegalThreadStateException` aus:

**Listing 12.3:** com/tutego/insel/thread/FirstThread.java, main()

```

Thread t1 = new Thread(new DateCommand());
t1.start();

```

```

Thread t2 = new Thread(new CounterCommand());
t2.start();

```

Beim Starten des Programms erfolgt eine Ausgabe auf dem Bildschirm, die in etwa so aussehen kann:

```

Tue Aug 21 16:59:58 CEST 2007
0
1
2
3
4
5

```

```

6
7
8
9
Tue Aug 21 16:59:58 CEST 2007
10
...

```

Deutlich ist die Verzahnung der beiden Threads zu erkennen. Was allerdings auf den ersten Blick etwas merkwürdig wirkt, ist die erste Zeile des Datum-Threads und viele weitere Zeilen des Zähl-Threads. Dies hat jedoch nichts zu bedeuten und zeigt deutlich den Nichtdeterminismus<sup>2</sup> bei Threads. Interpretiert werden kann dies jedoch durch die unterschiedlichen Laufzeiten, die für die Datums- und Zeitausgabe nötig sind.

```

class java.lang.Thread
 implements Runnable

```

- `Thread(Runnable target)`  
Erzeugt einen neuen Thread mit einem Runnable, das den parallel auszuführenden Programmcode vorgibt.
- `void start()`  
Ein neuer Thread – neben dem die Methode aufrufenden Thread – wird gestartet. Der neue Thread führt die `run()`-Methode nebenläufig aus. Jeder Thread kann nur einmal gestartet werden.

#### → Hinweis

Wenn ein Thread im Konstruktor einer Runnable-Implementierung gestartet wird, sollte die Arbeitsweise bei der Vererbung beachtet werden. Nehmen wir an, eine Klasse leitet von einer anderen Klasse ab, die im Konstruktor einen Thread startet. Bildet die Applikation ein Exemplar der Unterklasse, so werden bei der Bildung des Objekts immer erst die Konstruktoren der Oberklasse aufgerufen. Dies hat zur Folge, dass der Thread schon läuft, auch wenn das Objekt noch nicht ganz gebaut ist. Die Erzeugung ist erst abgeschlossen, wenn nach dem Aufruf der Konstruktoren der Oberklassen der eigene Konstruktor vollständig abgearbeitet wurde.

---

<sup>2</sup> Nicht vorhersehbar, bedeutet hier: Wann der Scheduler den Kontextwechsel vornimmt, ist unbekannt.

### 12.2.3 Die Klasse Thread erweitern

Da die Klasse `Thread` selbst die Schnittstelle `Runnable` implementiert und die `run()`-Methode mit leerem Programmcode bereitstellt, können wir auch `Thread` erweitern, wenn wir eigene parallele Aktivitäten programmieren wollen:

**Listing 12.4:** com/tutego/insel/thread/DateThread.java, DateThread

```
public class DateThread extends Thread
{
 @Override public void run()
 {
 for (int i = 0; i < 20; i++)
 System.out.println(new Date());
 }
}
```

Dann müssen wir kein `Runnable`-Exemplar mehr in den Konstruktor einfügen, denn wenn unsere Klasse eine Unterklasse von `Thread` ist, reicht ein Aufruf der geerbten Methode `start()`. Danach arbeitet das Programm direkt weiter, führt also kurze Zeit später die nächste Anweisung hinter `start()` aus:

**Listing 12.5:** com/tutego/insel/thread/DateThreadUser, main()

```
Thread t = new DateThread();
t.start();
new DateThread().start(); // (*)
```

Die (\*)-Zeile zeigt, dass das Starten sehr kompakt auch ohne Zwischenspeicherung der Objektreferenz möglich ist.

```
class java.lang.Thread
implements Runnable
```

- `void run()`

Diese Methode in `Thread` hat einen leeren Rumpf. Unterklassen überschreiben `run()`, sodass sie den parallel auszuführenden Programmcode enthält.

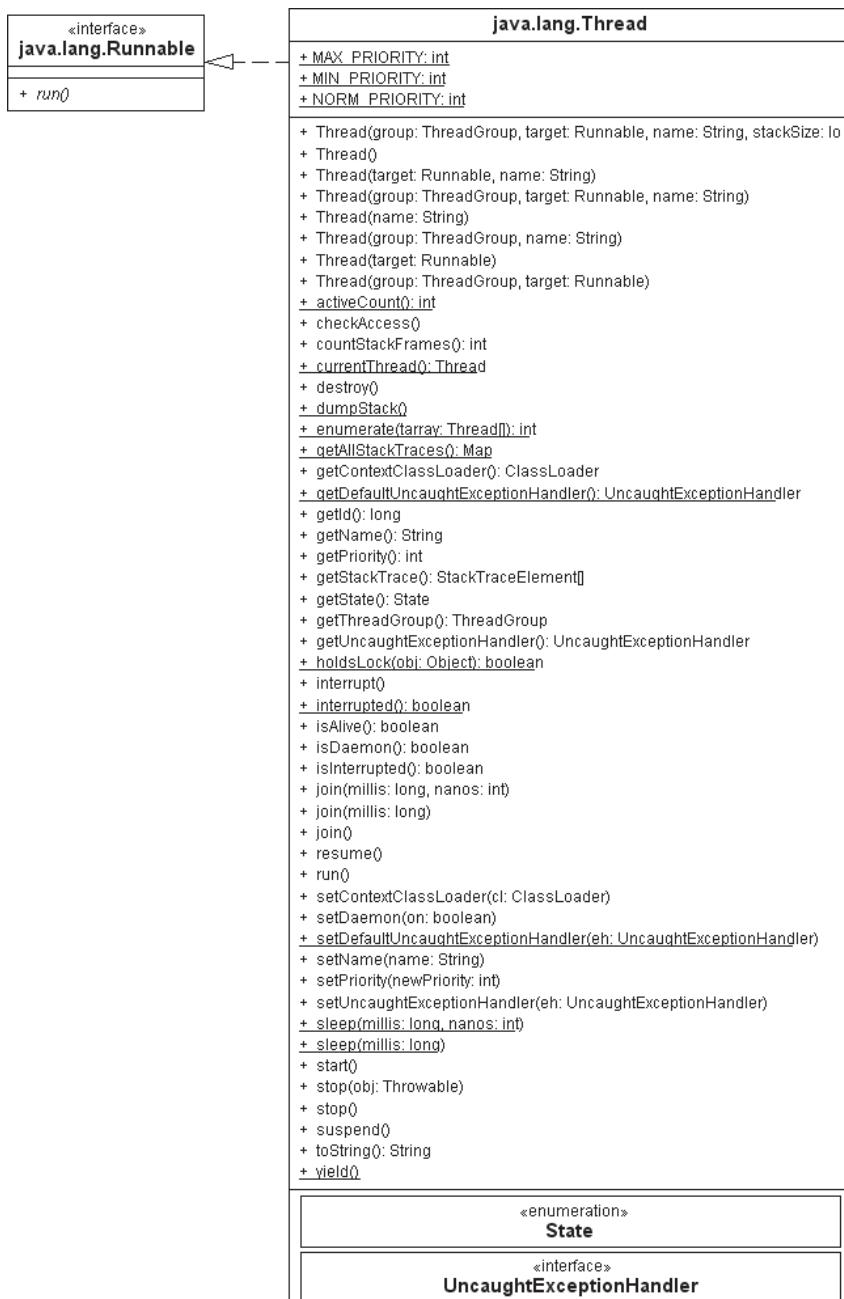


Abbildung 12.3: UML-Diagramm der Klasse Thread, die Runnable implementiert

## Überschreiben von start() und Selbststarter

Die Methode `start()` kann von uns auch überschrieben werden, was aber nur selten sinnvoll beziehungsweise nötig ist. Wir müssen dann darauf achten, `super.start()` aufzurufen, damit der Thread wirklich startet.

Damit wir als Thread-Benutzer nicht erst die `start()`-Methode aufrufen müssen, kann ein Thread sich auch selbst starten. Der Konstruktor ruft dazu einfach die eigene `start()`-Methode auf:

```
class DateThread extends Thread
{
 DateThread()
 {
 start();
 }
 // ... der Rest bleibt ...
}
```

### run() wurde statt start() aufgerufen: Ja, wo laufen sie denn?

Ein Programmierfehler, der Anfängern schnell unterläuft, ist folgender: Statt `start()` rufen sie aus Versehen `run()` auf dem Thread auf. Was geschieht? Fast genau das Gleiche wie bei `start()`, nur mit dem Unterschied, dass die Objektmethode `run()` nicht parallel zum übrigen Programm abgearbeitet wird. Der aktuelle Thread bearbeitet die `run()`-Methode sequenziell, bis sie zu Ende ist und die Anweisungen nach dem Aufruf an die Reihe kommen. Der Fehler fällt nicht immer direkt auf, denn die Aktionen in `run()` finden ja statt – nur eben nicht nebenläufig.

### Erweitern von Thread oder Implementieren von Runnable?

Die beste Idee wäre, `Runnable`-Objekte zu bauen, die dann dem Thread übergeben werden. Befehlsobjekte dieser Art sind recht flexibel, da die einfachen `Runnable`-Objekte leicht übergeben und sogar von Threads aus einem Thread-Pool ausgeführt werden können. Ein Nachteil der `Thread`-Erweiterung ist, dass die Einfachvererbung störend sein kann; erbt eine Klasse von `Thread`, ist die Erweiterung schon »aufgebraucht«. Doch, egal ob eine Klasse `Runnable` implementiert oder `Thread` erweitert, eines bleibt: eine neue Klasse.

## 12.3 Thread-Eigenschaften und -Zustände

Ein Thread hat eine ganze Reihe von Zuständen, wie einen Namen und eine Priorität, die sich erfragen und setzen lassen. Nicht jede Eigenschaft ist nach dem Start änderbar, doch welche das sind, zeigen die folgenden Abschnitte.

### 12.3.1 Der Name eines Threads

Ein Thread hat eine ganze Menge Eigenschaften – wie einen Zustand, eine Priorität und auch einen Namen. Dieser kann mit `setName()` gesetzt und mit `getName()` erfragt werden.

```
class java.lang.Thread
 implements Runnable
```

- `Thread(String name)`  
Erzeugt ein neues Thread-Objekt und setzt den Namen. Sinnvoll bei Unterklassen, die den Konstruktor über `super(name)` aufrufen.
- `Thread(Runnable target, String name)`  
Erzeugt ein neues Thread-Objekt mit einem `Runnable` und setzt den Namen.
- `final String getName()`  
Liefert den Namen des Threads. Der Name wird im Konstruktor angegeben oder mit `setName()` zugewiesen. Standardmäßig ist der Name »Thread-x«, wobei x eine eindeutige Nummer ist.
- `final void setName(String name)`  
Ändert den Namen des Threads.

### 12.3.2 Wer bin ich?

Eine Erweiterung der Klasse `Thread` hat den Vorteil, dass geerbte Methoden wie `getName()` sofort genutzt werden können. Wenn wir `Runnable` implementieren, genießen wir diesen Vorteil nicht.

Die Klasse `Thread` liefert mit der statischen Methode `currentThread()` die Objektreferenz für das `Thread`-Exemplar, das diese Anweisung gerade ausführt. Auf diese Weise lassen sich nicht-statische `Thread`-Methoden wie `getName()` verwenden.

zB

**Beispiel**

Gib die aktuelle Priorität des laufenden Threads und den Namen aus:

```
System.out.println(Thread.currentThread().getPriority()); // z.B. 5
System.out.println(Thread.currentThread().getName()); // z.B. main
```

Falls es in einer Schleife wiederholten Zugriff auf `Thread.currentThread()` gibt, sollte das Ergebnis zwischengespeichert werden, denn der Aufruf ist nicht ganz billig.

```
class java.lang.Thread
 implements Runnable
```

- static Thread currentThread()  
Liefert den Thread, der das laufende Programmstück ausführt.

12

**12.3.3 Schläfer gesucht**

Manchmal ist es notwendig, einen Thread eine bestimmte Zeit lang anzuhalten. Dazu lassen sich Methoden zweier Klassen nutzen:

- **die überladene statische Methode** `Thread.sleep()`: Etwas erstaunlich ist sicherlich, dass sie keine Objektmethode von einem Thread-Objekt ist, sondern eine statische Methode. Ein Grund wäre, dass dadurch verhindert wird, externe Threads zu beeinflussen. Es ist nicht möglich, einen fremden Thread, über dessen Referenz wir verfügen, einfach einige Sekunden lang schlafen zu legen und ihn so von der Ausführung abzuhalten.
- **die Objektmethode** `sleep() auf einem TimeUnit-Objekt`: Auch sie bezieht sich immer auf den ausführenden Thread. Der Vorteil gegenüber `sleep()` ist, dass hier die Zeiteinheiten besser sichtbar sind.

zB

**Beispiel**

Der ausführende Thread soll zwei Sekunden lang schlafen. Einmal mit `Thread.sleep()`:

```
try {
 Thread.sleep(2000);
} catch (InterruptedException e) { }
```

**zB Beispiel (Forts.)**

Dann mit TimeUnit:

```
try {
 TimeUnit.SECONDS.sleep(2);
} catch (InterruptedException e) { }
```

Der Schlaf kann durch eine `InterruptedException` unterbrochen werden, etwa durch `interrupt()`. Die Ausnahme muss behandelt werden, da sie keine `RuntimeException` ist.

Praktisch wird das Erweitern der Klasse `Thread` bei inneren anonymen Klassen. Die folgende Anweisung gibt nach zwei Sekunden Schlafzeit eine Meldung auf dem Bildschirm aus:

**Listing 12.6:** com/tutego/insel/thread/SleepInInnerClass.java, main()

```
new Thread() {
 @Override public void run() {
 try {
 Thread.sleep(2000);
 System.out.println("Zeit ist um.");
 } catch (InterruptedException e) { e.printStackTrace(); }
 }
}.start();
```

Da `new Thread(){...}` ein Exemplar der anonymen Unterklasse ergibt, lässt die auf dem Ausdruck aufgerufene Objektmethode `start()` den Thread gleich loslaufen. Aufgaben dieser Art lösen auch die Timer gut.

```
class java.lang.Thread
 implements Runnable
```

- static void sleep(long millis) throws InterruptedException

Der aktuell ausgeführte Thread wird mindestens `millis` Millisekunden schlafen gelegt. Unterbricht ein anderer Thread den schlafenden, wird vorzeitig eine `InterruptedException` ausgelöst.

- static void sleep(long millis, int nanos) throws InterruptedException

Der aktuell ausgeführte Thread wird mindestens `millis` Millisekunden und zusätzlich `nanos` Nanosekunden schlafen gelegt. Im Gegensatz zu `sleep(long)` wird bei einer nega-

tiven Millisekundenanzahl eine `IllegalArgumentException` ausgelöst; auch wird diese Exception ausgelöst, wenn die Nanosekundenanzahl nicht zwischen 0 und 999.999 liegt.

```
enum java.util.concurrent.TimeUnit
 extends Enum<TimeUnit>
 implements Serializable, Comparable<TimeUnit>
```

- NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS  
Aufzählungselemente von `TimeUnit`.
- `void sleep(long timeout) throws InterruptedException`  
Führt ein `Thread.sleep()` für die Zeiteinheit aus.

Eine überladene Methode `Thread.sleep(long, TimeUnit)` wäre nett, gibt es aber nicht.

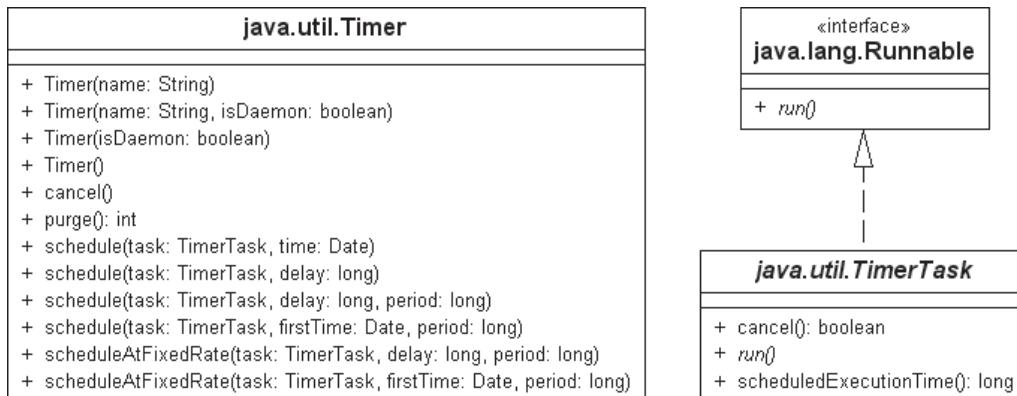


Abbildung 12.4: UML-Diagramme für Timer und TimerTask

### 12.3.4 Mit `yield()` auf Rechenzeit verzichten

Neben `sleep()` gibt es eine weitere Methode, um kooperative Threads zu programmieren: die Methode `yield()`. Sie funktioniert etwas anders als `sleep()`, da hier nicht nach Ablauf der genannten Millisekunden zum Thread zurückgekehrt wird, sondern `yield()` den Thread bezüglich seiner Priorität wieder in die Thread-Warteschlange des Systems einordnet. Einfach ausgedrückt, sagt `yield()` der Thread-Verwaltung: »Ich setze diese Runde aus und mache weiter, wenn ich das nächste Mal dran bin.«

```
class java.lang.Thread
 implements Runnable
```

- static void yield()

Der laufende Thread gibt freiwillig seine Rechenzeit ab. Die Methode ist für Implementierungen der JVM nicht verbindlich.

### 12.3.5 Der Thread als Dämon

Ein Server reagiert oft in einer Endlosschleife auf eingehende Aufträge vom Netzwerk und führt die gewünschte Aufgabe aus. In unseren bisherigen Programmen haben wir oft Endlosschleifen eingesetzt, sodass ein gestarteter Thread nie beendet wird. Wenn also `run()` wie in den vorangehenden Beispielen nie abbricht (Informatiker sprechen hier von *terminiert*), läuft der Thread immer weiter, auch wenn die Hauptapplikation beendet ist. Dies ist nicht immer beabsichtigt, da vielleicht Server-Funktionalität nach dem Beenden der Applikation nicht mehr gefragt ist. Dann sollte auch der endlos laufende Thread beendet werden. Um dies auszudrücken, erhält ein im Hintergrund arbeitender Thread eine spezielle Kennung: Der Thread wird als *Dämon*<sup>3</sup> gekennzeichnet. Standardmäßig ist ein aufgebauter Thread kein Dämon.



Ein Dämon ist wie ein Heinzelmannchen im Hintergrund mit einer Aufgabe beschäftigt. Wenn das Hauptprogramm beendet ist und die Laufzeitumgebung erkennt, dass kein normaler Thread mehr läuft, sondern nur Dämonen, dann ist das Ende der Dämonen eingeläutet, und die JVM kommt zum Ende. Denn Dämonen-Threads sind Zulieferer: Gibt es keine Klienten mehr, werden auch sie nicht mehr gebraucht. Das ist wie bei den Göttern der Scheibenwelt: Glaubt keiner an sie, hören sie auf zu existieren. Wir müssen uns also um das Ende des Dämons nicht kümmern. Gleichzeitig heißt das aber auch, dass ein Dämonen-Thread vorsichtig mit Ein-/Ausgabeoperationen sein muss, denn er kann jederzeit – auch etwa während einer Schreiboperation auf die Festplatte – abgebrochen werden, was zu beschädigten Daten führen kann.

---

<sup>3</sup> Das griechische δαίμων (engl. *daemon*) bezeichnet allerlei Wesen zwischen Gott und Teufel. Eine gute Einleitung gibt <http://de.wikipedia.org/wiki/D%C3%A4mon>.



### Hinweis

Der Garbage-Collector (GC) ist ein gutes Beispiel für einen Dämon. Nur, wenn es andere Threads gibt, muss der Speicher aufgeräumt werden. Gibt es keine anderen Threads mehr, kann auch die JVM mit beendet werden, was auch die Dämonen-Threads beendet.

### Wie ein Thread in Java zum Dämon wird

Einen Thread in Java als Dämon zu kennzeichnen, heißt, die Methode `setDaemon()` mit dem Argument `true` aufzurufen. Die Methode ist nur vor dem Starten des Threads erlaubt. Danach kann der Status nicht wieder vom Dämon in den normalen Benutzer-Thread umgesetzt werden. Die Auswirkungen von `setDaemon(true)` können wir am folgenden Programm ablesen:

Listing 12.7: com/tutego/insel/thread/DaemonThread.java

```
package com.tutego.insel.thread;

class DaemonThread extends Thread
{
 DaemonThread()
 {
 setDaemon(true);
 }

 @Override
 public void run()
 {
 while (true)
 System.out.println("Lauf, Thread, lauf");
 }

 public static void main(String[] args)
 {
 new DaemonThread().start();
 }
}
```

In diesem Programm wird der Thread gestartet, und danach ist die Anwendung sofort beendet. Vor dem Ende kann der neue Thread aber schon einige Zeilen auf der Konsole ausgeben. Klammern wir die Anweisung mit `setDaemon(true)` aus, läuft das Programm ewig, da die Laufzeitumgebung auf das natürliche Ende der Thread-Aktivität wartet.

```
class java.lang.Thread
 implements Runnable
```

- `final void setDaemon(boolean on)`

Markiert den Thread als Dämon oder normalen Thread. Die Methode muss aufgerufen werden, bevor der Thread gestartet wurde, andernfalls folgt eine `IllegalThreadStateException`. Mit anderen Worten: Nachträglich kann ein existierender Thread nicht mehr zu einem Dämon gemacht werden, und ihm kann auch nicht die Dämonenhafigkeit genommen werden, so er sie hat.

- `final boolean isDaemon()`

Testet, ob der Thread ein Dämon-Thread ist.

### 12.3.6 Das Ende eines Threads

Es gibt Threads, die dauernd laufen, weil sie zum Beispiel Serverfunktionen implementieren. Andere Threads führen einmalig eine Operation aus und sind danach beendet. Allgemein ist ein Thread beendet, wenn eine der folgenden Bedingungen zutrifft:

- Die `run()`-Methode wurde ohne Fehler beendet. Wenn wir eine Endlosschleife programmieren, würde diese potenziell einen nie endenden Thread bilden.
- In der `run()`-Methode tritt eine `RuntimeException` auf, die die Methode beendet. Das beendet weder die anderen Threads noch die JVM als Ganzes.
- Der Thread wurde von außen abgebrochen. Dazu dient die prinzipbedingt problematische Methode `stop()`, von deren Verwendung abgeraten wird und die auch veraltet ist.
- Die virtuelle Maschine wird beendet und nimmt alle Threads mit ins Grab.

### Wenn der Thread einen Fehler melden soll

Da ein Thread nebenläufig arbeitet, kann die `run()`-Methode synchron schlecht Exceptions melden oder einen Rückgabewert liefern. Wer sollte auch an welcher Stelle darauf hören? Eine Lösung für das Problem ist ein Listener, der sich beim Thread anmeldet und darüber informiert wird, ob der Thread seine Arbeit machen konnte oder nicht. Eine an-

dere Lösung gibt `Callable`, mit dem ein spezieller Fehlercode zurückgegeben oder eine `Exception` angezeigt werden kann. Speziell für ungeprüfte Ausnahmen kann ein `UncaughtExceptionHandler` weiterhelfen.

### 12.3.7 Einen Thread höflich mit Interrupt beenden

Der Thread ist in der Regel zu Ende, wenn die `run()`-Methode ordentlich bis zum Ende ausgeführt wurde. Enthält eine `run()`-Methode jedoch eine Endlosschleife – wie etwa bei einem Server, der auf eingehende Anfragen wartet –, so muss der Thread von außen zur Kapitulation gezwungen werden.

Wenn wir den Thread schon nicht von außen beenden wollen, können wir ihn immerhin bitten, seine Arbeit aufzugeben. Periodisch müsste er dann nur überprüfen, ob jemand von außen den Abbruchswunsch geäußert hat.

#### Die Methoden `interrupt()` und `isInterrupted()`

Die Methode `interrupt()` setzt von außen in einem Thread-Objekt ein internes Flag, das dann in der `run()`-Methode durch `isInterrupted()` periodisch abgefragt werden kann.

Das folgende Programm soll jede halbe Sekunde eine Meldung auf dem Bildschirm ausgeben. Nach zwei Sekunden wird der Unterbrechungswunsch mit `interrupt()` gemeldet. Auf dieses Signal achtet die sonst unendlich laufende Schleife und bricht ab:

**Listing 12.8:** com/tutego/insel/thread/ThreadusInterruptus.java, main()

```
Thread t = new Thread()
{
 @Override
 public void run()
 {
 System.out.println("Es gibt ein Leben vor dem Tod. ");

 while (! isInterrupted())
 {
 System.out.println("Und er läuft und er läuft und er läuft");

 try
 {
 Thread.sleep(500);
 }
 catch (InterruptedException e)
 {
 System.out.println("Der Thread wurde unterbrochen.");
 }
 }
 }
}
```

```

 Thread.sleep(500);
 }
 catch (InterruptedException e)
 {
 interrupt();
 System.out.println("Unterbrechung in sleep()");
 }
}

System.out.println("Das Ende");
}
};

t.start();
Thread.sleep(2000);
t.interrupt();

```

Die Ausgabe zeigt hübsch die Ablaufsequenz:

```

Es gibt ein Leben vor dem Tod.
Und er läuft und er läuft und er läuft
Und er läuft und er läuft und er läuft
Und er läuft und er läuft und er läuft
Und er läuft und er läuft und er läuft
Unterbrechung in sleep()
Das Ende

```

Die `run()`-Methode im `Thread` ist so implementiert, dass die Schleife genau dann verlassen wird, wenn `isInterrupted()` den Wert `true` ergibt, also von außen die `interrupt()`-Methode für dieses `Thread`-Exemplar aufgerufen wurde. Genau dies geschieht in der `main()`-Methode. Auf den ersten Blick ist das Programm leicht verständlich, doch vermutlich erzeugt das `interrupt()` im `catch`-Block die Aufmerksamkeit. Stünde diese Zeile dort nicht, würde das Programm aller Wahrscheinlichkeit nach nicht funktionieren. Das Geheimnis ist folgendes: Wenn die Ausgabe nur jede halbe Sekunde stattfindet, befindet sich der Thread fast die gesamte Zeit über in der Schlafmethode `sleep()`. Also wird vermutlich der `interrupt()` den Thread gerade beim Schließen stören. Genau dann wird `sleep()` durch `InterruptedException` unterbrochen, und der `catch`-Behandler fängt die Ausnahme ein. Jetzt passiert aber etwas Unerwartetes: Durch die Unterbrechung wird das interne Flag zurückgesetzt, sodass `isInterrupted()` meint, die Unterbrechung habe

gar nicht stattgefunden. Daher muss `interrupt()` erneut aufgerufen werden, da das Abbruch-Flag neu gesetzt werden muss und `isInterrupted()` das Ende bestimmen kann.

Wenn wir mit der Objektmethode `isInterrupted()` arbeiten, müssen wir beachten, dass neben `sleep()` auch die Methoden `join()` und `wait()` durch die `InterruptedException` das Flag löschen.

#### Hinweis

Die Methoden `sleep()`, `wait()` und `join()` lösen alle eine `InterruptedException` aus, wenn sie durch die Methode `interrupt()` unterbrochen werden. Das heißt, `interrupt()` beendet diese Methoden mit der Ausnahme.

### Zusammenfassung: `interrupt()`, `isInterrupted()` und `interrupt()`

Die Methodennamen sind verwirrend gewählt, sodass wir die Aufgaben noch einmal zusammenfassen wollen: Die Objektmethode `interrupt()` setzt in einem (anderen) Thread-Objekt ein Flag, dass es einen Antrag gab, den Thread zu beenden. Sie beendet aber den Thread nicht, obwohl es der Methodename nahelegt. Dieses Flag lässt sich mit der Objektmethode `isInterrupted()` abfragen. In der Regel wird dies innerhalb einer Schleife geschehen, die darüber bestimmt, ob die Aktivität des Threads fortgesetzt werden soll. Die statische Methode `interrupted()` ist zwar auch eine Anfragemethode und testet das entsprechende Flag des aktuell laufenden Threads, wie `Thread.currentThread().isInterrupted()`, aber zusätzlich löscht es den Interrupt-Status auch, was `isInterrupted()` nicht tut. Zwei aufeinanderfolgende Aufrufe von `interrupted()` führen daher zu einem `false`, es sei denn, in der Zwischenzeit erfolgt eine weitere Unterbrechung.

### 12.3.8 `UncaughtExceptionHandler` für unbehandelte Ausnahmen

Einer der Gründe für das Ende eines Threads ist eine unbehandelte Ausnahme, etwa von einer nicht aufgefangenen `RuntimeException`. Um in diesem Fall einen kontrollierten Abgang zu ermöglichen, lässt sich an den Thread ein `UncaughtExceptionHandler` hängen, der immer dann benachrichtigt wird, wenn der Thread wegen einer nicht behandelten Ausnahme endet.

`UncaughtExceptionHandler` ist eine in `Thread` deklarierte innere Schnittstelle, die eine Operation `void uncaughtException(Thread t, Throwable e)` vorschreibt. Eine Implementierung der Schnittstelle lässt sich entweder einem individuellen Thread oder allen Threads anhängen, sodass im Fall des Abbruchs durch unbehandelte Ausnahmen die JVM die Me-

thode uncaughtException() aufruft. Auf diese Weise kann die Applikation im letzten Atemzug noch den Fehler loggen, den die JVM über das Throwable e übergibt.

```
class java.lang.Thread
 implements Runnable
```

- void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)  
Setzt den UncaughtExceptionHandler für den Thread.
- Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()  
Liefert den aktuellen UncaughtExceptionHandler.
- Static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)  
Setzt den UncaughtExceptionHandler für alle Threads.
- static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()  
Liefert den zugewiesenen UncaughtExceptionHandler aller Threads.

Ein mit setUncaughtExceptionHandler() lokal gesetzter UncaughtExceptionHandler überschreibt den Eintrag für den setDefaultUncaughtExceptionHandler(). Zwischen dem mit dem Thread assoziierten Handler und dem globalen gibt es noch einen Handler-Typ für Thread-Gruppen, der jedoch seltener verwendet wird.

## 12.4 Der Ausführer (Executor) kommt

Zur parallelen Ausführung eines Runnable ist immer ein Thread notwendig. Obwohl die nebenläufige Abarbeitung von Programmcode ohne Threads nicht möglich ist, sind doch beide sehr stark verbunden, und es wäre gut, wenn das Runnable von dem tatsächlich abarbeitenden Thread etwas getrennt wäre. Das hat mehrere Gründe:

- Schon beim Erzeugen eines Thread-Objekts muss das Runnable-Objekt im Thread-Konstruktor übergeben werden. Es ist nicht möglich, das Thread-Objekt aufzubauen, dann über eine JavaBean-Setter-Methode das Runnable-Objekt zuzuweisen und anschließend den Thread mit start() zu starten.
- Wird start() auf dem Thread-Objekt zweimal aufgerufen, so führt der zweite Aufruf zu einer Ausnahme. Ein erzeugter Thread kann also ein Runnable durch zweimaliges Aufrufen von start() nicht gleich zweimal abarbeiten. Für eine erneute Abarbeitung eines Runnable ist also mit unseren bisherigen Mitteln immer ein neues Thread-Objekt nötig.

- Der Thread beginnt mit der Abarbeitung des Programmcodes vom Runnable sofort nach dem Aufruf von `start()`. Die Implementierung vom Runnable selbst müsste geändert werden, wenn der Programmcode nicht sofort, sondern später (nächste Tageschau) oder wiederholt (immer Weihnachten) ausgeführt werden soll.

Wünschenswert ist eine Abstraktion, die das Ausführen des Runnable-Programmcodes von der technischen Realisierung (etwa den Threads) trennt.

### 12.4.1 Die Schnittstelle Executor

Seit Java 5 gibt es eine Abstraktion für Klassen, die Befehle über Runnable ausführen. Die Schnittstelle Executor schreibt eine Methode vor:

```
interface java.util.concurrent.Executor
```

- `void execute(Runnable command)`

Wird später von Klassen implementiert, die ein Runnable abarbeiten können.

Jeder, der nun Befehle über Runnable abarbeitet, ist Executor.

12

### Konkrete Executoren

Von dieser Schnittstelle gibt es bisher zwei wichtige Implementierungen:

- `ThreadPoolExecutor`: Die Klasse baut eine Sammlung von Threads auf, den *Thread-Pool*. Ausführungsanfragen werden von den freien Threads übernommen.
- `ScheduledThreadPoolExecutor`: Eine Erweiterung von `ThreadPoolExecutor` um die Fähigkeit, zu bestimmten Zeiten oder mit bestimmten Wiederholungen Befehle abzuarbeiten.

Die beiden Klassen haben nicht ganz so triviale Konstruktoren, und eine Utility-Klasse vereinfacht den Aufbau dieser speziellen Executor-Objekte.

```
class java.util.concurrent.Executors
```

- `static ExecutorService newCachedThreadPool()`  
Liefert einen Thread-Pool mit wachsender Größe.
- `static ExecutorService newFixedThreadPool(int nThreads)`  
Liefert einen Thread-Pool mit maximal `nThreads`.
- `static ScheduledExecutorService newSingleThreadScheduledExecutor()`

- static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)  
Gibt spezielle Executor-Objekte zurück, um Wiederholungen festzulegen.

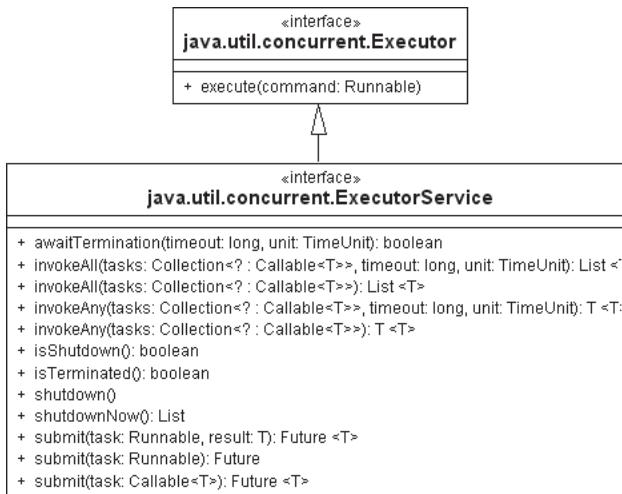


Abbildung 12.5: Die Schnittstelle ExecutorService, die Executor erweitert

ExecutorService ist eine Schnittstelle, die Executor erweitert. Unter anderem sind hier Operationen zu finden, die die Ausführer herunterfahren. Im Falle von Thread-Pools ist das nützlich, da die Threads ja sonst nicht beendet würden, weil sie auf neue Aufgaben warten.



Abbildung 12.6: Executors-Klasse mit statischen Methoden

### 12.4.2 Die Thread-Pools

Eine wichtige statische Methode der Klasse Executors ist newCachedThreadPool(). Das Ergebnis ist ein ExecutorService-Objekt, eine Implementierung von Executor mit der Methode execute(Runnable):

**Listing 12.9:** com/tutego/insel/thread/concurrent/ThreadPoolDemo.java, main() – 1

```
Runnable r1 = new Runnable() {
 @Override public void run() {
 System.out.println("A1 " + Thread.currentThread());
 System.out.println("A2 " + Thread.currentThread());
 }
};

Runnable r2 = new Runnable() {
 @Override public void run() {
 System.out.println("B1 " + Thread.currentThread());
 System.out.println("B2 " + Thread.currentThread());
 }
};
```

Jetzt lässt sich der Thread-Pool als ExecutorService beziehen und lassen sich die beiden Befehlsobjekte als Runnable über execute() ausführen:

**Listing 12.10:** com/tutego/insel/thread/concurrent/ThreadPoolDemo.java, main() – 2

```
ExecutorService executor = Executors.newCachedThreadPool();

executor.execute(r1);
executor.execute(r2);

Thread.sleep(500);

executor.execute(r1);
executor.execute(r2);

executor.shutdown();
```

Die Ausgabe zeigt sehr schön die Wiederverwendung der Threads:

```
A1 Thread[pool-1-thread-1,5,main]
A2 Thread[pool-1-thread-1,5,main]
B1 Thread[pool-1-thread-2,5,main]
B2 Thread[pool-1-thread-2,5,main]
B1 Thread[pool-1-thread-1,5,main]
B2 Thread[pool-1-thread-1,5,main]
A1 Thread[pool-1-thread-2,5,main]
A2 Thread[pool-1-thread-2,5,main]
```

Die `toString()`-Methode von `Thread` ist so implementiert, dass zunächst der Name der Threads auftaucht, den die Pool-Implementierung gesetzt hat, dann die Priorität und der Name des Threads, der den neuen Thread gestartet hat. Am neuen Namen ist abzulesen, dass hier zwei Threads von einem Thread-Pool 1 verwendet werden: `thread-1` und `thread-2`. Nach dem Ausführen der beiden Aufträge und der kleinen Warterei sind die Threads fertig und für neue Jobs bereit, sodass A1 und A2 beim zweiten Mal mit den wieder freien Threads abgearbeitet werden.

Interessant sind die folgenden drei Operationen zur Steuerung des Pool-Endes:

```
interface java.util.concurrent.ExecutorService
extends Executor
```

- `void shutdown()`  
Fährt den Thread-Pool herunter. Laufende Threads werden nicht abgebrochen, aber neue Anfragen werden nicht angenommen.
- `boolean isShutdown()`  
Wurde der Executor schon heruntergefahren?
- `List<Runnable> shutdownNow()`  
Gerade ausführende Befehle werden zum Stoppen angeregt. Die Rückgabe ist eine Liste der zu beendenden Kommandos.

## 12.5 Synchronisation über kritische Abschnitte

Wenn Threads in Java ein eigenständiges Leben führen, ist dieser Lebensstil nicht immer unproblematisch für andere Threads, insbesondere beim Zugriff auf gemeinsam genutzte Ressourcen. In den folgenden Abschnitten erfahren wir mehr über gemeinsam

genutzte Daten und Schutzmaßnahmen beim konkurrierenden Zugriff durch mehrere Threads.

### 12.5.1 Gemeinsam genutzte Daten

Ein Thread besitzt zum einen seine eigenen Variablen, etwa die Objektvariablen, kann aber auch statische Variablen nutzen, wie das folgende Beispiel zeigt:

```
class T extends Thread
{
 static int result;

 public void run() { ... }
}
```

In diesem Fall können verschiedene Exemplare der Klasse `T`, die jeweils einen Thread bilden, Daten austauschen, indem sie die Informationen in `result` ablegen oder daraus entnehmen. Threads können aber auch an einer zentralen Stelle eine Datenstruktur erfragen und dort Informationen entnehmen oder Zugriff auf gemeinsame Objekte über eine Referenz bekommen. Es gibt also viele Möglichkeiten, wie Threads – und damit potenziell parallel ablaufende Aktivitäten – Daten austauschen können.

### 12.5.2 Probleme beim gemeinsamen Zugriff und kritische Abschnitte

Da Threads ihre eigenen Daten verwalten – sie haben alle eigene lokale Variablen und einen Stack –, kommen sie sich gegenseitig nicht in die Quere. Auch wenn mehrere Threads gemeinsame Daten nur lesen, ist das unbedenklich; Schreiboperationen sind jedoch kritisch. Wenn sich zehn Nutzer einen Drucker teilen, der die Ausdrucke nicht als unteilbare Einheit bündelt, lässt sich leicht ausmalen, wie das Ergebnis aussieht. Seiten, Zeilen oder gar einzelne Zeichen aus verschiedenen Druckaufträgen werden bunt gemischt ausgedruckt.

Die Probleme haben ihren Ursprung in der Art und Weise, wie die Threads umgeschaltet werden. Der Scheduler unterbricht zu einem uns unbekannten Zeitpunkt die Abarbeitung eines Threads und lässt den nächsten arbeiten. Wenn nun der erste Thread gerade Programmzeilen abarbeitet, die zusammengehören, und der zweite Thread beginnt, parallel auf diesen Daten zu arbeiten, so ist der Ärger vorprogrammiert. Wir müssen also Folgendes ausdrücken können: »Wenn ich den Job mache, dann möchte ich der Einzige

sein, der die Ressource – etwa einen Drucker – nutzt.« Erst nachdem der Drucker den Auftrag eines Benutzers fertiggestellt hat, darf er den nächsten in Angriff nehmen.

### Kritische Abschnitte

Zusammenhängende Programmblöcke, denen während der Ausführung von einem Thread kein anderer Thread »reinwurschteln« sollte und die daher besonders geschützt werden müssen, nennen sich *kritische Abschnitte*. Wenn lediglich ein Thread den Programmteil abarbeitet, dann nennen wir dies *gegenseitigen Ausschluss* oder *atomar*. Wir könnten das etwas lockerer sehen, wenn wir wüssten, dass innerhalb der Programmblöcke nur von den Daten gelesen wird. Sobald aber nur ein Thread Änderungen vornehmen möchte, ist ein Schutz nötig. Denn arbeitet ein Programm bei nebenläufigen Threads falsch, ist es nicht *thread-sicher* (engl. *thread-safe*).

Wir werden uns nun Beispiele für kritische Abschnitte anschauen und dann sehen, wie wir diese in Java realisieren können.

### Nicht kritische Abschnitte

Wenn mehrere Threads auf das gleiche Programmstück zugreifen, muss das nicht zwangsläufig zu einem Problem führen, und Thread-Sicherheit ist immer gegeben. Immutable Objekte – nehmen wir an, ein Konstruktor belegt einmalig die Zustände – sind automatisch *thread-sicher*, da es keine Schreibzugriffe gibt und bei Lesezugriffen nichts schiefgehen kann. Immutable-Klassen wie `String` oder Wrapper-Klassen kommen daher ohne Synchronisierung aus.

Das Gleiche gilt für Methoden, die keine Objekteigenschaften verändern. Da jeder Thread seine *thread-eigenen* Variablen besitzt – jeder Thread hat einen eigenen Stack –, können lokale Variablen, auch Parametervariablen, beliebig gelesen und geschrieben werden. Wenn zum Beispiel zwei Threads die folgende statische Utility-Methode aufrufen, ist das kein Problem:

```
public static String reverse(String s)
{
 return new StringBuilder(s).reverse().toString();
}
```

Jeder Thread wird eine eigene Variablenbelegung für `s` haben und ein temporäres Objekt vom Typ `StringBuilder` referenzieren.

### Thread-sichere und nicht thread-sichere Klassen der Java Bibliothek

Es gibt in Java viele Klassen, die nicht thread-sicher sind – das ist sogar der Standard. So sind etwa alle Format-Klassen, wie `MessageFormat`, `NumberFormat`, `DecimalFormat`, `ChoiceFormat`, `DateFormat` und `SimpleDateFormat` nicht für den nebenläufigen Zugriff gemacht. In der Regel steht das in der JavaDoc, etwa bei `DateFormat`:

*»Synchronization. Date formats are not synchronized. It is recommended to create separate format instances for each thread. If multiple threads access a format concurrently, it must be synchronized externally.«*

Wer also Objekte nebenläufig verwendet, der sollte immer in der Java API-Dokumentation nachschlagen, ob es dort einen Hinweis gibt, ob die Objekte überhaupt thread-sicher sind.

In einigen wenigen Fällen haben Entwickler die Wahl zwischen thread-sicheren und nicht thread-sicheren Klassen im JDK:

| Nicht thread-sicher        | Thread-sicher             |
|----------------------------|---------------------------|
| <code>StringBuilder</code> | <code>StringBuffer</code> |
| <code>ArrayList</code>     | <code>Vector</code>       |
| <code>HashMap</code>       | <code>Hashtable</code>    |

Tabelle 12.2: Thread-sichere und nicht thread-sichere Klassen

Obwohl es die Auswahl bei den Datenstrukturen im Prinzip gibt, werden `Vector` und `Hashtable` dennoch nicht verwendet.

### 12.5.3 Punkte parallel initialisieren

Nehmen wir an, ein Thread T1 möchte ein `Point`-Objekt `p` mit den Werten (1,1) belegen und ein zweiter Thread T2 möchte eine Belegung mit den Werten (2,2) durchführen.

| Thread T1             | Thread T2             |
|-----------------------|-----------------------|
| <code>p.x = 1;</code> | <code>p.x = 2;</code> |
| <code>p.y = 1;</code> | <code>p.y = 2;</code> |

Tabelle 12.3: Zwei Threads belegen beide den Punkt p

Beide Threads können natürlich bei einem 2-Kern-Prozessor parallel arbeiten, aber da sie auf gemeinsame Variablen zugreifen, ist der Zugriff auf  $x$  bzw.  $y$  von  $p$  trotzdem sequenziell. Um es nicht allzu kompliziert zu machen, vereinfachen wir unser Ausführungsmodell so, dass wir zwar zwei Threads laufen haben, aber nur jeweils einer ausgeführt wird. Dann ist es möglich, dass T1 mit der Arbeit beginnt und  $x = 1$  setzt. Da der Thread-Scheduler einen Thread jederzeit unterbrechen kann, kann nun T2 an die Reihe kommen, der  $x = 2$  und  $y = 2$  setzt. Wird dann T1 wieder Rechenzeit zugeteilt, darf T1 an der Stelle weitermachen, wo er aufgehört hat, und  $y = 1$  folgt. In einer Tabelle ist das Ergebnis noch besser zu sehen:

| Thread T1  | Thread T2  | x/y |
|------------|------------|-----|
| $p.x = 1;$ |            | 1/0 |
|            | $p.x = 2;$ | 2/0 |
|            | $p.y = 2;$ | 2/2 |
| $p.y = 1;$ |            | 2/1 |

Tabelle 12.4: Mögliche sequentielle Abarbeitung der Punktbelegung

Wir erkennen das nicht beabsichtigte Ergebnis (2,1), es könnte aber auch (1,2) sein, wenn wir das gleiche Szenario beginnend mit T2 durchführen. Je nach zuerst abgearbeitetem Thread wäre jedoch nur (1,1) oder (2,2) korrekt. Die Threads sollen ihre Arbeit aber atomar erledigen, denn die Zuweisung bildet einen kritischen Abschnitt, der geschützt werden muss. Standardmäßig sind die zwei Zuweisungen nicht-atomare Operationen und können unterbrochen werden. Um dies an einem Beispiel zu zeigen, sollen zwei Threads ein Point-Objekt verändern. Die Threads belegen  $x$  und  $y$  immer gleich, und immer dann, wenn sich die Koordinaten unterscheiden, soll es eine Meldung geben:

Listing 12.11: com/tutego/insel/thread/concurrent/ParallelPointInit.java, main()

```
final Point p = new Point();

Runnable r = new Runnable()
{
 @Override public void run()
 {
 int x = (int)(Math.random() * 1000), y = x;

 while (true)
 {
 if (x != y)
 System.out.println("x=" + x + " y=" + y);
 }
 }
}
```

```

p.x = x; p.y = y; // *

int xc = p.x, yc = p.y; // *

if (xc != yc)
 System.out.println("Aha: x=" + xc + ", y=" + yc);
}

}

};

new Thread(r).start();
new Thread(r).start();

```

Die interessanten Zeilen sind mit \* markiert. `p.x = x; p.y = y;` belegt die Koordinaten neu, und `int xc = p.x, yc = p.y;` liest die Koordinaten erneut aus. Würden Belegung und Auslesen in einem Rutsch passieren, dürfte überhaupt keine unterschiedliche Belegung von `x` und `y` zu finden sein. Doch das Beispiel zeigt es anders:

```

Aha: x=58, y=116
Aha: x=116, y=58
Aha: x=58, y=116
Aha: x=58, y=116
...

```

Was wir mit den parallelen Punkten vor uns haben, sind Effekte, die von den Ausführungszeiten der einzelnen Operationen abhängen. In Abhängigkeit von dem Ort der Unterbrechung wird ein fehlerhaftes Verhalten produziert. Dieses Szenario nennt sich im Englischen *race condition* beziehungsweise *race hazard* (zu Deutsch auch *Wettkampfsituation*).

#### 12.5.4 Kritische Abschnitte schützen

Beginnen wir mit einem anschaulichen Alltagsbeispiel. Gehen wir aufs Klo, schließen wir die Tür hinter uns. Möchte jemand anderes auf die Toilette, muss er warten. Vielleicht kommen noch mehrere dazu, die müssen dann auch warten, und eine Warteschlange bildet sich. Dass die Toilette besetzt ist, signalisiert die abgeschlossene Tür.

Jeder Wartende muss so lange vor dem Klo ausharren, bis das Schloss geöffnet wird, selbst wenn der auf der Toilette Sitzende nach einer langen Nacht einnickten sollte.

Wie übertragen wir das auf Java? Wenn die Laufzeitumgebung nur einen Thread in einen Block lassen soll, ist ein *Monitor*<sup>4</sup> nötig. Ein Monitor wird mithilfe eines *Locks* (zu Deutsch *Schloss*) realisiert, das ein Thread öffnet oder schließt. Tritt ein Thread in den kritischen Abschnitt ein, muss Programmcode wie eine Tür abgeschlossen werden (engl. *lock*). Erst wenn der Abschnitt durchlaufen wurde, darf die Tür wieder aufgeschlossen werden (engl. *unlock*), und ein anderer Thread kann den Abschnitt betreten.



### Hinweis

Ein anderes Wort für Lock ist *Mutex* (engl. *mutual exclusion*, also »gegenseitiger Ausschluss«). Der Begriff *Monitor* wird oft mit *Lock (Mutex)* gleichgesetzt, doch kann ein Monitor mit Warten/Benachrichtigen mehr als ein klassischer Lock. In der Definition der Sprache Java (JLS Kapitel 17) tauchen die Begriffe *Mutex* und *Lock* allerdings nicht auf; die Autoren sprechen nur von den Monitor-Aktionen *lock* und *unlock*. Die Java Virtual Machine definiert dafür die Opcodes *monitorenter* und *monitorexit*.

## Java-Konstrukte zum Schutz der kritischen Abschnitte

Wenn wir auf unser Punkte-Problem zurückkommen, so stellen wir fest, dass zwei Zeilen auf eine Variable zugreifen:

```
p.x = x; p.y = y;
int xc = p.x, yc = p.y;
```

Diese beiden Zeilen bilden also einen kritischen Abschnitt, den jeweils nur ein Thread betreten darf. Wenn also einer der Threads mit `p.x = x` beginnt, muss er so lange den exklusiven Zugriff bekommen, bis er mit `yc = p.y` endet.

Aber wie wird nun ein kritischer Abschnitt bekannt gegeben? Zum Markieren und Abschließen dieser Bereiche gibt es zwei Konzepte:

---

<sup>4</sup> Der Begriff geht auf C. A. R. Hoare zurück, der in seinem Aufsatz »Communicating Sequential Processes« von 1978 erstmals dieses Konzept veröffentlichte.

| Konstrukt           | Eingebautes Schlüsselwort                       | Java-Standardbibliothek                                                    |
|---------------------|-------------------------------------------------|----------------------------------------------------------------------------|
| Schlüsselwort/Typen | synchronized                                    | java.util.concurrent.locks.Lock                                            |
| Nutzungsschema      | <pre> synchronized {     Tue1     Tue2 } </pre> | <pre> lock.lock(); {     Tue1     Tue2 } lock.unlock();<sup>5</sup> </pre> |

Tabelle 12.5: Lock-Konzepte

Beim `synchronized` entsteht Bytecode, der der JVM sagt, dass ein kritischer Block beginnt und endet. So überwacht die JVM, ob ein zweiter Thread warten muss, wenn er in einen synchronisierten Block eintritt, der schon von einem Thread ausgeführt wird. Bei `Lock` ist das Ein- und Austreten explizit vom Entwickler programmiert, und vergisst er das, ist das ein Problem. Und während bei der `Lock`-Implementierung das Objekt, an dem synchronisiert wird, offen hervortritt, ist das bei `synchronized` nicht so offensichtlich. Hier gilt es zu wissen, dass jedes Objekt in Java implizit mit einem Monitor verbunden ist. Da moderne Programme aber mittlerweile mit `Lock`-Objekten arbeiten, tritt die `synchronized`-Möglichkeit, die schon Java 1.0 zur Synchronisation bot, etwas in den Hintergrund.

Fassen wir zusammen: Nicht thread-sichere Abschnitte müssen geschützt werden. Sie können entweder mit `synchronized` geschützt werden, bei dem der Eintritt und Austritt implizit geregelt ist, oder durch `Lock`-Objekte. Befindet sich dann ein Thread in einem geschützten Block und möchte ein zweiter Thread in den Abschnitt, muss er so lange warten, bis der erste Thread den Block wieder freigibt. So ist die Abarbeitung über mehrere Threads einfach synchronisiert, und das Konzept eines Monitors gewährleistet seriellen Zugriff auf kritische Ressourcen. Die kritischen Bereiche sind nicht per se mit einem Monitor verbunden, sondern werden eingerahmt, und dieser Rahmen ist mit einem Monitor (`Lock`) verbunden.

Mit dem Abschließen und Aufschließen werden wir uns noch intensiver in den folgenden Abschnitten beschäftigen.

---

<sup>5</sup> Vereinfachte Darstellung, später mehr.

### 12.5.5 Kritische Abschnitte mit ReentrantLock schützen

Seit Java 5 gibt es die Schnittstelle `Lock`, mit der sich ein kritischer Block markieren lässt. Ein Abschnitt beginnt mit `lock()` und endet mit `unlock()`:

Listing 12.12: com/tutego/insel/thread/concurrent/ParallelPointInitSync.java, main()

```
final Lock lock = new ReentrantLock();
final Point p = new Point();

Runnable r = new Runnable()
{
 @Override public void run()
 {
 int x = (int)(Math.random() * 1000), y = x;

 while (true)
 {
 lock.lock();

 p.x = x; p.y = y; // *
 int xc = p.x, yc = p.y; // *

 lock.unlock();

 if (xc != yc)
 System.out.println("Aha: x=" + xc + ", y=" + yc);
 }
 }
};

new Thread(r).start();
new Thread(r).start();
```

Mit dieser Implementierung wird keine Ausgabe auf dem Bildschirm folgen.

## Die Schnittstelle `java.util.concurrent.locks.Lock`

`Lock` ist eine Schnittstelle, von der `ReentrantLock` die wichtigste Implementierung ist. Mit ihr lässt sich der Block betreten und verlassen.

```
interface java.util.concurrent.locks.Lock
```

- `void lock()`

Wartet so lange, bis der ausführende Thread den kritischen Abschnitt betreten kann, und markiert ihn dann als betreten. Hat schon ein anderer Thread an diesem Lock-Objekt ein `lock()` aufgerufen, so muss der aktuelle Thread warten, bis der Lock wieder frei ist. Hat der aktuelle Thread schon den Lock, kann er bei der Implementierung `ReentrantLock` wiederum `lock()` aufrufen und sperrt sich nicht selbst.

- `boolean tryLock()`

Wenn der kritische Abschnitt sofort betreten werden kann, ist die Funktionalität wie bei `lock()`, und die Rückgabe ist `true`. Ist der Lock gesetzt, so wartet die Methode nicht wie `lock()`, sondern kehrt mit einem `false` zurück.

- `boolean tryLock(long time, TimeUnit unit) throws InterruptedException`

Versucht in der angegebenen Zeitspanne den Lock zu bekommen. Das Warten kann mit `interrupt()` auf dem Thread unterbrochen werden, was `tryLock()` mit einer Exception beendet.

- `void unlock()`

Verlässt den kritischen Block.

- `void lockInterruptibly() throws InterruptedException`

Wartet wie `lock()`, um den kritischen Abschnitt betreten zu dürfen, kann aber mit einem `interrupt()` von außen abgebrochen werden (der `lock()`-Methode ist ein Interrupt egal). Implementierende Klassen müssen diese Vorgabe nicht zwingend umsetzen, sondern können die Methode auch mit einem einfachen `lock()` realisieren. `ReentrantLock` implementiert `lockInterruptibly()` erwartungsgemäß.

### Beispiel

zB

Wenn wir sofort in den kritischen Abschnitt gehen können, tun wir das; sonst tun wir etwas anderes:

```
Lock lock = ...;
if (lock.tryLock())
{
 try {
 ...
 }
}
```

**zB Beispiel (Forts.)**

```

 }
 finally { lock.unlock(); }
}
else
...

```

Die Implementierung ReentrantLock kann noch ein bisschen mehr als lock() und unlock():

```

class java.util.concurrent.locks.ReentrantLock
implements Lock, Serializable

```

- ReentrantLock()

Erzeugt ein neues Lock-Objekt, das nicht dem am längsten Wartenden den ersten Zugriff gibt.

- ReentrantLock(boolean fair)

Erzeugt ein neues Lock-Objekt mit fairem Zugriff, gibt also dem am längsten Wartenden den ersten Zugriff.

- boolean isLocked()

Fragt an, ob der Lock gerade genutzt wird und im Moment kein Betreten möglich ist.

- final int getQueueLength()

Ermittelt, wie viele auf das Betreten des Blocks warten.

- int getHoldCount()

Gibt die Anzahl der erfolgreichen lock()-Aufrufe ohne passendes unlock() zurück. Sollte nach Beenden des Vorgangs 0 sein.

**zB Beispiel**

Das Warten auf den Lock kann unterbrochen werden:

```

Lock l = new ReentrantLock();
try
{
 l.lockInterruptibly();
 try

```

zB

**Beispiel (Forts.)**

```
{
 ...
}
finally { l.unlock(); }
}
catch (InterruptedException e) { ... }
```

Wenn wir den Lock nicht bekommen haben, dürfen wir ihn auch nicht freigeben!

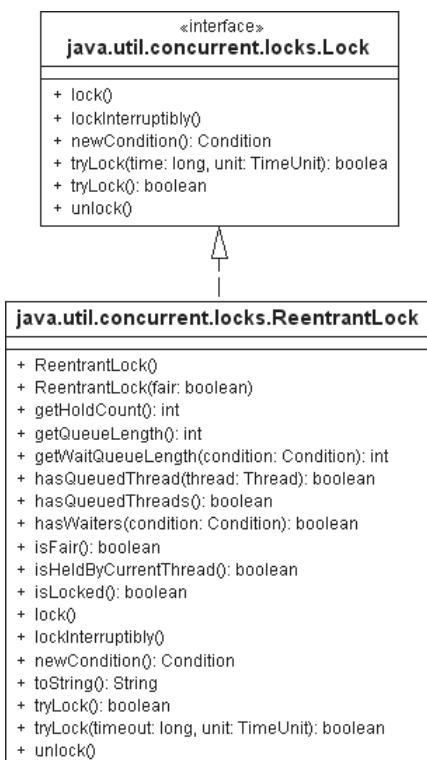


Abbildung 12.7: Die Klasse ReentrantLock implementiert die Schnittstelle Lock

## 12.6 Zum Weiterlesen

»Java 7 – Mehr als eine Insel« geht detaillierter auf die API ein.





## Kapitel 13

# Einführung in Datenstrukturen und Algorithmen

*Glück ist ganz einfach gute Gesundheit und ein schlechtes Gedächtnis.  
– Ernest Hemingway (1899–1961)*

Algorithmen<sup>1</sup> sind ein zentrales Thema der Informatik. Ihre Erforschung und Untersuchung nimmt dort einen bedeutenden Platz ein. Algorithmen operieren nur dann effektiv mit Daten, wenn diese geeignet strukturiert sind. Schon das Beispiel Telefonbuch zeigt, wie wichtig die Ordnung der Daten nach einem Schema ist. Die Suche nach einer Telefonnummer bei gegebenem Namen gelingt schnell, während die Suche nach einem Namen bei bekannter Telefonnummer ein mühseliges Unterfangen darstellt. Datenstrukturen und Algorithmen sind also eng miteinander verbunden, und die Wahl der richtigen Datenstruktur entscheidet über effiziente Laufzeiten; beide erfüllen allein nie ihren Zweck. Leider ist die Wahl der »richtigen« Datenstruktur nicht so einfach, wie es sich anhört, und diverse schwierige Probleme in der Informatik sind wohl deswegen noch nicht gelöst, weil eine passende Datenorganisation bis jetzt nicht gefunden wurde.

13

### 13.1 Datenstrukturen und die Collection-API

Dynamische Datenstrukturen passen ihre Größe der Anzahl der Daten an, die sie aufnehmen. Schon in Java 1.0 brachte die Standard-Bibliothek fundamentale Datenstrukturen mit, aber erst mit Java 1.2 wurde mit der Collection-API der Umgang mit Datenstrukturen und Algorithmen auf eine gute Basis gestellt. In Java 5 gab es große Anpassungen durch Einführung der Generics.

---

<sup>1</sup> Das Wort *Algorithmus* geht auf den Namen des persisch-arabischen Mathematikers Ibn Mûsâ Al-Chwârimî zurück, der im 9. Jahrhundert lebte.



### Begriffe

Ein Container ist ein Objekt, das wiederum andere Objekte aufnimmt und die Verantwortung für die Elemente übernimmt. Wir werden die Begriffe *Container*, *Sammlung* und *Collection* synonym verwenden.

#### 13.1.1 Designprinzip mit Schnittstellen, abstrakten und konkreten Klassen

Das Design der Collection-Klassen folgt vier Prinzipien:

- Schnittstellen legen Gruppen von Operationen für die verschiedenen Behältertypen fest. So gibt es zum Beispiel mit `List` eine Schnittstelle für Sequenzen (Listen) und mit `Map` eine Schnittstelle für Assoziativspeicher, die Schlüssel-Werte-Paare verbinden.
- Abstrakte Basisklassen führen die Operationen der Schnittstellen auf eine minimale Zahl von als abstrakt deklarierten Grundoperationen zurück. So greift etwa `addAll()` auf mehrere `add()`-Aufrufe zurück oder `isEmpty()` auf `getSize()`. (Mit den abstrakten Basisimplementierungen wollen wir uns nicht weiter beschäftigen. Sie sind interessanter, wenn eigene Datenstrukturen auf der Basis der Grundimplementierung entworfen werden.)
- Konkrete Klassen für bestimmte Behältertypen übernehmen die entsprechende abstrakte Basisklasse und ergänzen die unbedingt erforderlichen Grundoperationen (und ergänzen einige die Performance steigernde Methoden gegenüber der allgemeinen Lösung in der Oberklasse). Sie sind in der Nutzung unsere direkten Ansprechpartner. Für eine Liste können wir zum Beispiel die konkrete Klasse `ArrayList` und als Assoziativspeicher die Klasse `TreeMap` nutzen.
- Algorithmen, wie die Suche nach einem Element, gehören zum Teil zur Schnittstelle der Datenstrukturen. Zusätzlich gibt es mit der Klasse `Collections` eine Utility-Klasse mit weiteren Algorithmen.

#### 13.1.2 Die Basis-Schnittstellen Collection und Map

Alle Datenstrukturen aus der Collection-API fußen entweder auf der Schnittstelle `java.util.Collection` (für Listen, Mengen, Schlangen) oder auf `java.util.Map` (für Assoziativspeicher). Durch die gemeinsame Schnittstelle erhalten alle implementierenden Klassen einen gemeinsamen Rahmen. Die Operationen lassen sich grob einteilen in:

- Basisoperationen zum Erfragen der Elementanzahl und zum Hinzufügen, Löschen, Selektieren und Finden von Elementen
- Mengenoperationen, um etwa andere Sammlungen einzufügen
- Feldoperationen bei `Collection`, um die Sammlung in ein Array zu konvertieren, und bei `Map` in Operationen, um alternative Ansichten von Schlüsseln oder Werten zu bekommen.

### 13.1.3 Die Utility-Klassen Collections und Arrays

Datenstrukturen implementieren Algorithmen, etwa die Verwaltung von Elementen in einem Binärbaum oder einem Array. Java bietet zudem zwei besondere Klassen für insbesondere Such- und Sortier-Algorithmen: die Utility-Klasse `Collections` bietet Hilfsmethoden für `Collection`-Objekte – und einige wenige Methoden für Mengen –, und `Arrays` bietet statische Hilfsmethoden für Felder. Wichtig ist, auf die Schreibweise zu achten: `Collection` vs. `Collections`. Die Namensgebung ist jedoch einheitlich, denn in der gesamten Java API gibt es mehrere Beispiele für Utility-Klassen, die ein »s« am Ende bekommen und ausschließlich statische Methoden bereitstellen.

### 13.1.4 Das erste Programm mit Container-Klassen

Bis auf Assoziativspeicher implementieren alle Container-Klassen das Interface `Collection` und haben dadurch schon wichtige Methoden, um Daten aufzunehmen, zu manipulieren und auszulesen. Das folgende Programm erzeugt als Datenstruktur eine *verkettete Liste*, fügt Strings ein und gibt zum Schluss die Sammlung auf der Standardausgabe aus:

**Listing 13.1:** com/tutego/insel/util/MyFirstCollection.java, MyFirstCollection

```
public class MyFirstCollection
{
 private static void fill(Collection<String> c)
 {
 c.add("Juvy");
 c.add("Tina");
 c.add("Joy");
 }
}
```

```

public static void main(String[] args)
{
 List<String> c = new LinkedList<String>();
 fill(c);
 System.out.println(c); // [Juvy, Tina, Joy]
 Collections.sort(c);
 System.out.println(c); // [Joy, Juvy, Tina]
}
}

```

Das Beispiel zeigt unterschiedliche Aspekte der Collection-API:

- Seit Java 5 sind alle Datenstrukturen generisch deklariert. Statt `new LinkedList()` schreiben wir `new LinkedList<String>()`.
- Unserer eigenen statischen Methode `fill()` ist es egal, welche `Collection` wir ihr geben. Sie arbeitet nicht nur auf der `LinkedList`, sondern genauso auf einer `ArrayList` und auf Mengen (Set-Objekte), denn Set-Klassen implementieren ebenfalls `Collection`.
- Eine Liste lässt sich mit `add()` füllen. Die Methode schreibt die Schnittstelle `Collection` vor, und `LinkedList` realisiert die Operation aus der Schnittstelle.
- Während `Collection` eine Schnittstelle ist, die von unterschiedlichen Datenstrukturen implementiert wird, ist `Collections` eine Utility-Klasse mit vielen Hilfsmethoden, etwa zum Sortieren mit `Collections.sort()`.



### Tipp

Nutze immer den kleinste möglichen Typ! Wir haben das an zwei Stellen getan. Statt `fill(LinkedList<String> c)` deklariert das Programm `fill(Collection<String> c)`, und statt `LinkedList<String> c = new LinkedList<String>()` nutzt es `List<String> c = new LinkedList<String>()`. Mit dieser Schreibweise lassen sich unter softwaretechnischen Gesichtspunkten leicht die konkreten Datenstrukturen ändern, aber etwa die Methodensignatur ändert sich nicht und ist breiter aufgestellt. Es ist immer schön, wenn wir – etwa aus Gründen der Geschwindigkeit oder Speicherplatzbeschränkung – auf diese Weise leicht die Datenstruktur ändern können und der Rest des Programms unverändert bleibt. Das ist die Idee der schnittstellenorientierten Programmierung, und es ist in Java selten nötig, den konkreten Typ einer Klasse direkt anzugeben.

### 13.1.5 Die Schnittstelle Collection und Kernkonzepte

Unterschnittstellen erweitern Collection und schreiben Verhalten vor, ob etwa der Container die Reihenfolge des Einfügens beachtet, Werte doppelt beinhaltet darf oder die Werte sortiert hält; List, Set, Queue, Deque und NavigableSet sind dabei die wichtigsten.

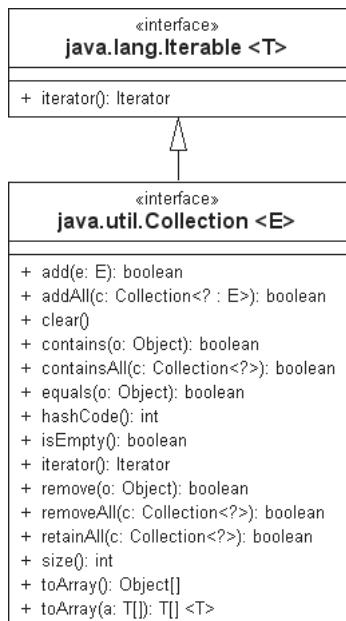


Abbildung 13.1: UML-Diagramm der Schnittstelle Collection

Es folgt eine Übersicht über alle Methoden:

```

interface java.util.Collection<E>
extends Iterable<E>

```

- `boolean add(E o)`

Optional. Fügt dem Container ein Element hinzu und gibt true zurück, falls sich das Element einfügen lässt. Gibt false zurück, wenn schon ein Objekt gleichen Werts vorhanden ist und doppelte Werte nicht erlaubt sind. Diese Semantik gilt etwa bei Mengen. Erlaubt der Container das Hinzufügen grundsätzlich nicht, löst er eine `UnsupportedOperationException` aus.

- `boolean addAll(Collection<? extends E> c)`

Fügt alle Elemente der Collection c dem Container hinzu.

- `void clear()`  
Optional. Löscht alle Elemente im Container. Wird dies vom Container nicht unterstützt, wird eine `UnsupportedOperationException` ausgelöst.
- `boolean contains(Object o)`  
Liefert true, falls der Container ein inhaltlich gleiches Element enthält.
- `boolean containsAll(Collection<?> c)`  
Liefert true, falls der Container alle Elemente der Collection c enthält.
- `boolean isEmpty()`  
Liefert true, falls der Container keine Elemente enthält.
- `Iterator<E> iterator()`  
Liefert ein Iterator-Objekt über alle Elemente des Containers.
- `boolean remove(Object o)`  
Optional. Entfernt das angegebene Objekt aus dem Container, falls es vorhanden ist.
- `boolean removeAll(Collection<?> c)`  
Optional. Entfernt alle Objekte der Collection c aus dem Container.
- `boolean retainAll(Collection<?> c)`  
Optional. Entfernt alle Objekte, die nicht in der Collection c vorkommen.
- `int size()`  
Gibt die Anzahl der Elemente im Container zurück.
- `Object[] toArray()`  
Gibt ein Array mit allen Elementen des Containers zurück.
- `<T> T[] toArray(T[] a)`  
Gibt ein Array mit allen Elementen des Containers zurück. Verwendet das als Argument übergebene Array als Zielcontainer, wenn es groß genug ist. Sonst wird ein Array passender Größe angelegt, dessen Laufzeittyp a entspricht.
- `boolean equals(Object o)`  
Prüft, a) ob das angegebene Objekt o ein kompatibler Container ist, und b), ob alle Elemente aus dem eigenen Container `equals()`-gleich der Elemente des anderen Containers sind, und c) ob sie – falls vorhanden – die gleiche Ordnung haben.
- `int hashCode()`  
Liefert den Hash-Wert des Containers. Dies ist wichtig, wenn der Container als Schlüssel in Hash-Tabellen verwendet wird. Dann darf der Inhalt aber nicht mehr geändert werden, da der Hash-Wert von allen Elementen des Containers abhängt.



### Hinweis

Der Basistyp `Collection` ist typisiert, genauso wie die Unterschnittstellen und implementierenden Klassen. Auffällig sind die Methoden `remove(Object)` und `contains(Object)`, die gerade nicht mit dem generischen Typ `E` versehen sind, was zur Konsequenz hat, dass diese Methoden mit beliebigen Objekten aufgerufen werden können. Fehler schleichen sich schnell ein, wenn der Typ der eingefügten Objekte ein anderer ist als der beim Löschversuch, etwa bei `HashSet<Long>` `set` mit anschließendem `set.add(1L)` und `remove(1)`.

### Anzeige der Veränderungen durch boolesche Rückgaben

Der Rückgabewert einiger Methoden wie `add()` oder `remove()` ist ein `boolean` und könnte natürlich auch `void` sein. Doch die Collection-API signalisiert über die Rückgabe, ob eine Änderung der Datenstruktur erfolgte oder nicht. Bei Mengen liefert `add()` etwa `false`, wenn ein gleiches Element schon in der Menge ist; `add()` ersetzt das alte nicht durch das neue.

13

### Vergleiche im Allgemeinen auf Basis von `equals()`

Der Methode `equals()` kommt bei den Elementen, die in die Datenstrukturen wandern, eine besondere Rolle zu. Jedes Objekt, das eine `ArrayList`, `LinkedList`, `HashSet` und alle anderen Datenstrukturen<sup>2</sup> aufnehmen soll, muss zwingend `equals()` implementieren. Denn Methoden wie `contains()`, `remove()` vergleichen die Elemente mit `equals()` auf Gleichheit und nicht mit `==` auf Identität.

### Beispiel

zB

Ein neues Punkt-Objekt kommt in die Datenstruktur. Nun wird es mit einem anderen `equals()`-gleichen Objekt auf das Vorkommen in der `Collection` geprüft und gelöscht:

```
Collection<Point> list = new ArrayList<Point>();
list.add(new Point(47, 11));
System.out.println(list.size()); // 1
System.out.println(list.contains(new Point(47, 11))); // true
```

---

<sup>2</sup> Lassen wir die besondere Klasse `IdentityHashMap` außen vor.

**zB Beispiel (Forts.)**

```
list.remove(new Point(47, 11));
System.out.println(list.size()); // 0
```

Eigene Klassen müssen folglich equals() aus der absoluten Oberklasse Object überschreiben. Umgekehrt heißt das auch, dass Objekte, die kein sinnvolles equals() besitzen, nicht von den Datenstrukturen aufgenommen werden können; ein Beispiel hierfür ist StringBuilder/StringBuffer.

### 13.1.6 Schnittstellen, die Collection erweitern und Map

Es gibt einige elementare Schnittstellen, die einen Container weiter untergliedern, etwa in der Art, wie Elemente gespeichert werden.

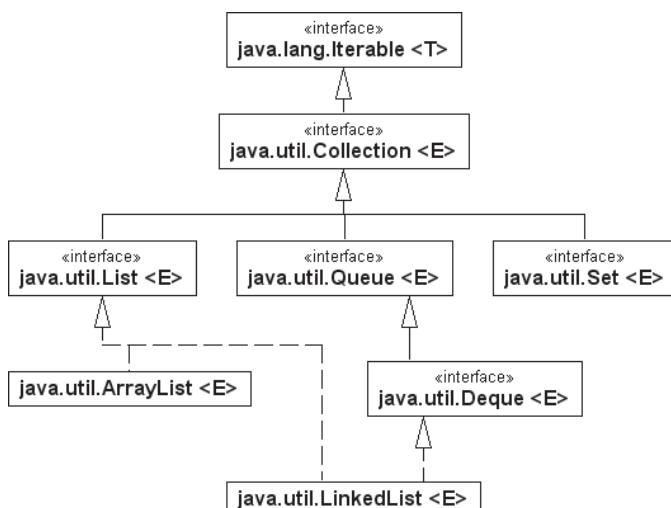


Abbildung 13.2: Zentrale Schnittstellen und Klassen der Collection-API

### Die Schnittstelle List für Sequenzen

Die Schnittstelle `List`<sup>3</sup>, die die Collection-Schnittstelle erweitert, enthält zusätzliche Operationen für eine geordnete Liste (auch *Sequenz* genannt) von Elementen. Auf die

<sup>3</sup> Wie in der Collection-Design-FAQ unter <http://java.sun.com/javase/6/docs/technotes/guides/collections/designfaq.html#11> nachzulesen ist, hätte die Schnittstelle durchaus Sequence heißen können.

Elemente einer Liste lässt sich über einen ganzzahligen Index zugreifen, und es kann linear nach Elementen gesucht werden. Doppelte Elemente sind erlaubt, auch beliebig viele `null`-Einträge.

Zwei bekannte implementierende Klassen sind `LinkedList` sowie `ArrayList`. Weil das AWT-Paket eine Klasse mit dem Namen `List` deklariert, muss bei der import-Deklaration darauf geachtet werden, das richtige `java.util.List` statt `java.awt.List` zu verwenden.

### Die Schnittstelle Set für Mengen

Ein `Set` ist eine im mathematischen Sinne definierte Menge von Objekten. Wie von mathematischen Mengen bekannt, darf ein `Set` keine doppelten Elemente enthalten. Für zwei nicht identische Elemente `e1` und `e2` eines `Set`-Objekts liefert der Vergleich `e1.equals(e2)` also immer `false`. Genauer gesagt: Aus `e1.equals(e2)` folgt, dass `e1` und `e2` identische Objektreferenzen sind, sich also auf dasselbe Mengenelement beziehen.

Besondere Beachtung muss Objekten geschenkt werden, die ihren Wert nachträglich ändern, da so zunächst ungleiche Mengenelemente inhaltlich gleich werden können. Dies kann ein `Set` nicht kontrollieren. Als weitere Einschränkung gilt, dass eine Menge sich selbst nicht als Element enthalten darf. Die wichtigste konkrete Mengen-Klasse ist `HashSet`.

`NavigableSet` – beziehungsweise ihr Muttertyp `SortedSet` – erweitert `Set` um die Eigenschaft, Elemente sortiert auslesen zu können. Das Sortierkriterium wird durch ein Exemplar der Hilfsklasse `Comparator` bestimmt, oder die Elemente implementieren `Comparable`. Die Klassen `TreeSet` und `ConcurrentSkipListSet` implementieren die Schnittstellen und erlauben mit einem Iterator oder einer Feld-Repräsentation Zugriff auf die sortierten Elemente.

### Die Schnittstelle Queue für (Warte-)Schlangen

Eine `Queue` arbeitet nach dem FIFO-Prinzip (First in, First out); zuerst eingefügte Elemente werden zuerst wieder ausgegeben, getreu nach dem Motto »Wer zuerst kommt, mahlt zuerst«. Die Schnittstelle `Queue` deklariert Operationen für alle Warteschlangen und wird etwa von den Klassen `LinkedList` und `PriorityQueue` implementiert.

### Queue mit zwei Enden

Während die `Queue` Operationen bietet, um an einem Ende Daten anzuhängen und zu erfragen, bietet die Datenstruktur `Deque` (vom Englischen »double-ended queue«) Opera-

tionen an beiden Enden. Die Klasse `LinkedList` ist zum Beispiel eine Implementierung von `Deque`. Die Datenstruktur wird wie »Deck« ausgesprochen.

## Die Schnittstelle `Map`

Eine Datenstruktur, die einen *Schlüssel* (engl. *key*) mit einem *Wert* (engl. *value*) verbindet, heißt *assoziativer Speicher*. Sie erinnert an ein Gedächtnis und ist mit einem Wörterbuch oder Nachschlagewerk vergleichbar. Betrachten wir ein Beispiel: Auf einem Personalausweis findet sich eine eindeutige Nummer, eine ID, die einmalig für jeden Bundesbürger ist. Wenn nun in einem Assoziativspeicher alle Passnummern gespeichert sind, lässt sich leicht über die Passnummer (Schlüssel) die Person (Wert) herausfinden, also der Name der Person, die Gültigkeit des Ausweises usw. In die gleiche Richtung geht ein Beispiel, das ISB-Nummern mit Büchern verbindet. Ein Assoziativspeicher könnte zu der ISB-Nummer zum Beispiel das Erscheinungsjahr assoziieren, ein anderer Assoziativspeicher eine Liste von Rezensionen.



### Hinweis

Gerne wird als Beispiel für einen Assoziativspeicher ein Telefonbuch mit einer Assoziation zwischen Namen und Telefonnummern genannt. Wenn das mit einem Assoziativspeicher realisiert werden muss, reicht natürlich der Name alleine nicht aus, sondern der Ort/das Land müssen dazukommen (ich bin zum Beispiel nicht der einzige Christian Ullensboom; in Erlangen wohnt mein Namensvetter). Auch ist es weniger ein Problem, dass in einem Familienhaushalt mehrere Personen die gleiche Telefonnummer besitzen. Vielmehr wird die Tatsache zum Problem, dass eine Person unterschiedliche Telefonnummern, etwa eine Mobil- und Festnetznummer, besitzen kann. Damit das Modell korrekt bleibt, muss eine Assoziation zwischen einem Namen und einer Liste von Telefonnummern bestehen. Ein Assoziativspeicher ist flexibel genug dafür: Der assoziierte Wert muss kein einfacher Wert wie eine Zahl oder String sein, sondern kann eine komplexe Datenstruktur sein.

In Java schreibt die Schnittstelle `Map` Verhalten für einen Assoziativspeicher vor. `Map` ist ein wenig anders als die anderen Schnittstellen. So erweitert die Schnittstelle `Map` die Schnittstelle `Collection` nicht. Das liegt daran, dass bei einem Assoziativspeicher Schlüssel und Wert immer zusammen vorkommen müssen und die Datenstruktur eine Operation wie `add(Object)` nicht unterstützen kann. Im Gegensatz zu `List` gibt es bei einer `Map` auch keine Position.

Die Schlüssel einer `Map` können mithilfe eines Kriteriums sortiert werden. Ist das der Fall, implementieren diese speziellen Klassen die Schnittstelle `NavigableMap` (bezie-

hungswise der Muttertyp `SortedSet`), die `Map` direkt erweitert. Das Sortierkriterium wird entweder über ein externes `Comparator`-Objekt festgelegt, oder die Elemente in der `Map` sind vom Typ `Comparable`. Damit kann ein Iterator in einer definierten Reihenfolge einen assoziativen Speicher ablaufen. Bisher implementieren `TreeMap` und `ConcurrentSkipListMap` die Schnittstelle `NavigableMap`.

### 13.1.7 Konkrete Container-Klassen

Alle bisher vorgestellten Schnittstellen und Klassen dienen zur Modellierung und dem Programmieren nur als Basistyp. Die folgenden Klassen sind konkrete Klassen und können von uns benutzt werden:

| Typ                         | Implementierung | Erklärung                                                                                                                                                                                                                                       |
|-----------------------------|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Listen<br>(List)            | ArrayList       | Implementiert Listen-Funktionalität durch die Abbildung auf ein Feld; implementiert die Schnittstelle <code>List</code> .                                                                                                                       |
|                             | LinkedList      | LinkedList ist eine doppelt verkettete Liste, also eine Liste von Einträgen mit einer Referenz auf den jeweiligen Nachfolger und Vorgänger. Das ist nützlich beim Einfügen und Löschen von Elementen an beliebigen Stellen innerhalb der Liste. |
| Mengen<br>(Set)             | HashSet         | Eine Implementierung der Schnittstelle <code>Set</code> durch ein schnelles Hash-Verfahren.                                                                                                                                                     |
|                             | TreeSet         | Implementierung von <code>Set</code> durch einen Baum, der alle Elemente sortiert hält.                                                                                                                                                         |
|                             | LinkedHashSet   | Eine schnelle Mengen-Implementierung, die sich parallel auch die Reihenfolge der eingefügten Elemente merkt.                                                                                                                                    |
| Assoziativspeicher<br>(Map) | HashMap         | Implementiert einen assoziativen Speicher durch ein Hash-Verfahren.                                                                                                                                                                             |
|                             | TreeMap         | Exemplare dieser Klasse halten ihre Elemente in einem Binärbaum sortiert; implementiert <code>NavigableMap</code> .                                                                                                                             |
|                             | LinkedHashMap   | Ein schneller Assoziativspeicher, der sich parallel auch die Reihenfolge der eingefügten Elemente merkt.                                                                                                                                        |
|                             | WeakHashMap     | Verwaltet Elemente mit schwachen Referenzen, sodass die Laufzeitumgebung bei Speicherknappheit Elemente entfernen kann.                                                                                                                         |

Tabelle 13.1: Konkrete Container-Klassen

| Typ                 | Implementierung     | Erklärung                                                |
|---------------------|---------------------|----------------------------------------------------------|
| Schlange<br>(Queue) | LinkedList          | Die verkettete Liste implementiert Queue und auch Deque. |
|                     | ArrayBlocking-Queue | Eine blockierende Warteschlange                          |
|                     | PriorityQueue       | Prioritätswarteschlange                                  |

Tabelle 13.1: Konkrete Container-Klassen (Forts.)

### 13.1.8 Generische Datentypen in der Collection-API

Seit Java 5 macht die Collection-API massiv Gebrauch von Generics. Das fällt unter anderem dadurch auf, dass die API-Dokumentation einen parametrisierten Typ erwähnt und die Collection-Schnittstelle zum Beispiel nicht `add(Object e)` deklariert, sondern `add(E e)`. Generics gewährleisten bessere Typsicherheit, da nur spezielle Objekte in die Datenstruktur kommen. Mit den Generics lässt sich bei der Konstruktion einer Collection-Datenstruktur angeben, welche Typen zum Beispiel in der Datenstruktur-Liste erlaubt sind. Soll eine Spielerliste `players` nur Objekte vom Typ `Player` aufnehmen, so sieht die Deklaration so aus:

```
List<Player> players = new ArrayList<Player>();
```

Mit dieser Schreibweise lässt die Liste nur den Typ `Player` beim Hinzufügen und Anfragen zu, nicht aber andere Typen, wie etwa Zeichenketten. Das ist eine schöne Sicherheit für den Programmierer.

### Geschachtelte Generics

Die Schreibweise `List<String>` deklariert eine Liste, die `Strings` enthält. Um eine verkettete Liste aufzubauen, deren Elemente wiederum Listen mit `Strings` sind, lassen sich die Deklarationen auch zusammenführen:<sup>4</sup>

```
List<List<String>> las = new LinkedList<List<String>>();
```

---

<sup>4</sup> Das erinnert mich immer unangenehm an C: ein Feld von Pointern, die auf Strukturen zeigen, die Pointer enthalten.

### 13.1.9 Die Schnittstelle Iterable und das erweiterte for

Das erweiterte `for` erwartet rechts vom Doppelpunkt den Typ `java.lang.Iterable`, um durch eine Sammlung laufen zu können. Praktisch ist, dass alle `java.util.Collection`-Klassen die Schnittstelle `Iterable` implementieren, denn damit kann das erweiterte `for` leicht über diverse Sammlungen laufen.

#### Beispiel

zB

Füge Zahlen in eine sortierte Menge ein, und gib sie aus.

```
Collection<Integer> numbers = new TreeSet<Integer>();
numbers.add(10); numbers.add(2); numbers.add(5);
for (Integer number : numbers)
 System.out.println(number); // 2 5 10
```

Von der Datenstruktur übernimmt das erweiterte `for` den konkreten generischen Typ, hier `Integer`.

Ist die Sammlung nicht typisiert, wird die lokale Variable vom erweiterten `for` nicht den Typ bekommen können, sondern nur `Object`. Dann muss eine explizite Typanpassung im Inneren der Schleife vorgenommen werden.

#### Hinweis

←

Ist die Datenstruktur `null`, so führt das zu einer `NullPointerException`:

```
Collection<String> list = null;
for (String s : list) // ☹ NullPointerException zur Laufzeit
 ;
```

Es wäre interessant, wenn Java dann die Schleife überspringen würde, aber der Grund für die Ausnahme ist, dass die Realisierung vom erweiterten `for` versucht, eine Methode vom `Iterable` aufzurufen, was natürlich bei `null` schiefgeht. Bei Feldern gilt übrigens das Gleiche, auch wenn hier keine Methode aufgerufen wird.

## 13.2 Listen

Eine Liste steht für eine Sequenz von Daten, bei der die Elemente eine feste Reihenfolge besitzen. Die Schnittstelle `java.util.List` schreibt Verhalten vor, die alle konkreten Listen implementieren müssen. Interessante Realisierungen der `List`-Schnittstelle sind:

- `java.util.ArrayList`: Liste auf der Basis eines Feldes
- `java.util.LinkedList`: Liste durch verkettete Elemente
- `java.util.concurrent.CopyOnWriteArrayList`: schnelle Liste, optimal für häufige nebenläufige Lesezugriffe
- `java.util.Vector`: synchronisierte Liste seit Java 1.0, die der `ArrayList` wich. Die Klasse ist zwar nicht deprecated, sollte aber nicht mehr verwendet werden.

Die Methoden zum Zugriff über die gemeinsame Schnittstelle `List` sind immer die gleichen. So ermöglicht jede Liste einen Punktzugriff über `get(index)`, und jede Liste kann alle gespeicherten Elemente sequenziell über einen Iterator geben. Doch die Realisierungen einer Liste unterscheiden sich in Eigenschaften wie der Performance, dem Speicherplatzbedarf oder der Möglichkeit der sicheren Nebenläufigkeit.

Da in allen Datenstrukturen jedes Exemplar einer von `Object` abgeleiteten Klasse Platz findet, sind die Listen grundsätzlich nicht auf bestimmte Datentypen fixiert, doch Generics spezifizieren diese Typen genauer.

### 13.2.1 Erstes Listen-Beispiel

Listen haben die wichtige Eigenschaft, dass sie sich die Reihenfolge der eingefügten Elemente merken und dass Elemente auch doppelt vorkommen können. Wir wollen diese Listenfähigkeit für ein kleines Gedächtnisspiel nutzen. Der Anwender gibt Städte für eine Route vor, die sich das Programm in einer Liste merkt. Nach der Eingabe eines neuen Ziels auf der Route soll der Anwender alle Städte in der richtigen Reihenfolge wiedergeben. Hat er das geschafft, kommt eine neue Stadt hinzu. Im Prinzip ist das Spiel unendlich, doch da sich kein Mensch unendlich viele Städte in der Reihenfolge merken kann, wird es zu einer Falscheingabe kommen, was das Programm beendet.

**Listing 13.2:** com/tutego/insel/util/list/HowDoesYourRouteLooksLike.java

```
package com.tutego.insel.util.list;

import java.text.*;
import java.util.*;

public class HowDoesYourRouteLooksLike
{
 public static void main(String[] args)
 {
```

```

List<String> cities = new ArrayList<String>();

while (true)
{
 System.out.println("Welche neue Stadt kommt hinzu?");
 String newCity = new Scanner(System.in).nextLine();
 cities.add(newCity);

 System.out.printf("Wie sieht die gesamte Route aus? (Tipp: %d %s)%n",
 cities.size(),
 cities.size() == 1 ? "Stadt" : "Städte");

 for (String city : cities)
 {
 String guess = new Scanner(System.in).nextLine();
 if (! city.equalsIgnoreCase(guess))
 {
 System.out.printf("%s ist nicht richtig, %s wäre korrekt. Schade!%n",
 guess, city);
 return;
 }
 }
 System.out.println("Prima, alle Städte in der richtigen Reihenfolge!");
}
}

```

Die Methoden der Schnittstelle `List` sind:

```

interface java.util.List<E>
extends Collection<E>

```

- `boolean add(E o)`  
Fügt das Element am Ende der Liste an. Eine optionale Operation.
- `void add(int index, E element)`  
Fügt ein Objekt an der angegebenen Stelle in die Liste ein. Eine optionale Operation.

- `boolean addAll(int index, Collection<? extends E> c)`  
Fügt alle Elemente der Collection an der angegebenen Stelle in die Liste ein. Eine optionale Operation.
- `void clear()`  
Löscht alle Elemente aus der Liste. Eine optionale Operation.
- `boolean contains(Object o)`  
Liefert true, wenn das Element o in der Liste ist. Den Vergleich übernimmt `equals()`, und es ist kein Referenz-Vergleich.
- `boolean containsAll(Collection<?> c)`  
Liefert true, wenn alle Elemente der Sammlung c in der aktuellen Liste sind.
- `E get(int index)`  
Wird das Element an dieser angegebenen Stelle der Liste liefern.
- `int indexOf(Object o)`  
Liefert die Position des ersten Vorkommens für o oder -1, wenn kein Listenelement mit o inhaltlich – also per `equals()` und nicht per Referenz – übereinstimmt. Leider gibt es keine Methode, um ab einer bestimmten Stelle weiterzusuchen, so wie sie die Klasse `String` bietet. Dafür lässt sich jedoch eine Teilliste einsetzen, die `subList()` bildet – eine Methode, die später in der Aufzählung folgt.
- `boolean isEmpty()`  
Liefert true, wenn die Liste leer ist.
- `Iterator<E> iterator()`  
Liefert den Iterator. Die Methode ruft aber `listIterator()` auf und gibt ein `ListIterator`-Objekt zurück.
- `int lastIndexOf(Object o)`  
Sucht von hinten in der Liste nach dem ersten Vorkommen von o und liefert -1, wenn kein Listenelement inhaltlich mit o übereinstimmt.
- `ListIterator<E> listIterator()`  
Liefert einen Listen-Iterator für die ganze Liste. Ein Listen-Iterator bietet gegenüber dem allgemeinen Iterator für Container zusätzliche Operationen.
- `ListIterator<E> listIterator(int index)`  
Liefert einen Listen-Iterator, der die Liste ab der Position index durchläuft.
- `E remove(int index)`  
Entfernt das Element an der Position index aus der Liste.

- `boolean remove(Object o)`  
Entfernt das erste Objekt in der Liste, das `equals()`-gleich mit `o` ist. Liefert `true`, wenn ein Element entfernt wurde. Eine optionale Operation.
- `boolean removeAll(Collection<?> c)`  
Löscht in der eigenen Liste die Elemente aus `c`. Eine optionale Operation.
- `boolean retainAll(Collection<?> c)`  
Optional. Entfernt alle Objekte aus der Liste, die nicht in der Collection `c` vorkommen.
- `E set(int index, E element)`  
Ersetzt das Element an der Stelle `index` durch `element`. Eine optionale Operation.
- `List<E> subList(int fromIndex, int toIndex)`  
Liefert den Ausschnitt dieser Liste von Position `fromIndex` (einschließlich) bis `toIndex` (nicht mit dabei). Die zurückgelieferte Liste stellt eine Ansicht eines Ausschnitts der Originalliste dar. Änderungen an der Teilliste wirken sich auf die ganze Liste aus und umgekehrt (soweit sie den passenden Ausschnitt betreffen).
- `boolean equals(Object o)`  
Vergleicht die Liste mit einer anderen Liste. Zwei Listen-Objekte sind gleich, wenn ihre Elemente paarweise gleich sind.
- `int hashCode()`  
Liefert den Hashcode der Liste.

Was `List` der Collection hinzufügt, sind also die Index-basierten Methoden `add(int index, E element)`, `addAll(int index, Collection<? extends E> c)`, `get(int index)`, `indexOf(Object o)`, `lastIndexOf(Object o)`, `listIterator()`, `listIterator(int index)`, `remove(int index)`, `set(int index, E element)` und `subList(int fromIndex, int toIndex)`.

## 13.3 Mengen (Sets)

Eine Menge ist eine (erst einmal) ungeordnete Sammlung von Elementen. Jedes Element darf nur einmal vorkommen. Für Mengen sieht die Java-Bibliothek die Schnittstelle `java.util.Set` vor. Beliebte implementierende Klassen sind:

- `HashSet`: Schnelle Mengenimplementierung durch Hashing-Verfahren (dahinter steckt die `HashMap`)
- `TreeSet`: Mengen werden durch balancierte Binäräbäume realisiert, die eine Sortierung ermöglichen.

- **LinkedHashSet:** Schnelle Mengenimplementierung unter Beibehaltung der Einfügereihenfolge
- **EnumSet:** Eine spezielle Menge ausschließlich für Enum-Objekte
- **CopyOnWriteArrayList:** Schnelle Datenstruktur für viele lesende Operationen

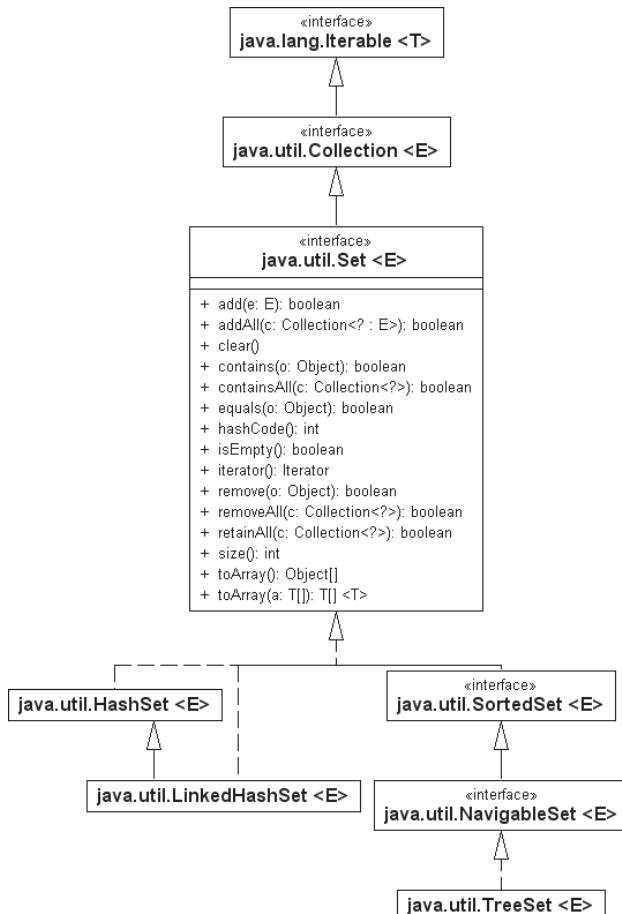


Abbildung 13.3: UML-Diagramm der Schnittstelle List

### 13.3.1 Ein erstes Mengen-Beispiel

Das folgende Programm analysiert einen Text und erkennt Städte, die vorher in eine Datenstruktur eingetragen wurden. Alle Städte, die im Text vorkommen, werden gesammelt und später ausgegeben.

**Listing 13.3:** com/tutego/insel/util/set/WhereHaveYouBeen

```
package com.tutego.insel.util.set;

import java.text.BreakIterator;
import java.util.*;

public class WhereHaveYouBeen
{
 public static String join(Iterable<?> iterable)
 {
 StringBuilder result = new StringBuilder();
 for (Object o : iterable)
 {
 if (result.length() != 0)
 result.append(", ");
 result.append(o.toString());
 }
 return result.toString();
 }

 public static void main(String[] args)
 {
 // Menge mit Städten aufbauen

 Set<String> allCities = new HashSet<String>();
 allCities.add("Sonsbeck");
 allCities.add("Düsseldorf");
 allCities.add("Manila");
 allCities.add("Seol");
 allCities.add("Siquijor");

 // Menge für besuchte Städte aufbauen

 Set<String> visitedCities = new TreeSet<String>();
 }
}
```

```

// Satz parsen und in Wörter zerlegen. Alle gefundenen Städte
// in neue Datenstruktur aufnehmen

String sentence = "Von Sonsbeck fahre ich nach Düsseldorf und fliege nach Manila.";
BreakIterator iter = BreakIterator.getWordInstance();
iter.setText(sentence);

for (int first = iter.first(), last = iter.next();
 last != BreakIterator.DONE;
 first = last, last = iter.next())
{
 String word = sentence.subSequence(first, last).toString();
 if (allCities.contains(word))
 visitedCities.add(word);
}

// Kleine Statistik

System.out.println("Anzahl besuchter Städte: " + visitedCities.size());
System.out.println("Anzahl nicht besuchter Städte: " +
 (allCities.size() - visitedCities.size()));
System.out.println("Besuchte Städte: " + join(visitedCities));
Set<String> unvisitedCities = new TreeSet<String>(allCities);
unvisitedCities.removeAll(visitedCities);
System.out.println("Unbesuchte Städte: " + join(unvisitedCities));
}
}

```

Insgesamt kommen drei Mengen im Programm vor:

- `allCities` speichert alle möglichen Städte. Die Wahl fällt auf den Typ `HashSet`, da die Menge nicht sortiert sein muss, Nebenläufigkeit kein Thema ist und `HashSet` eine gute Zugriffszeit bietet.
- Ein `TreeSet` `visitCities` merkt sich die besuchten Städte. Auch dieses Set ist schnell, aber hat den Vorteil, dass es die Elemente sortiert hält. Das ist später hübsch in der Ausgabe.

- Um alle nicht besuchten Städte herauszufinden, berechnet das Programm die Differenzmenge zwischen allen Städte und besuchten Städten. Es gibt in der Schnittstelle Set keine Methode, die das direkt macht, genau genommen gibt es keine Operation in Set, die den Rückgabetyp Set oder Collection hat. Also können wir nur mit einer Methode wie `removeAll()` arbeiten, die aus der Menge aller Städte die besuchten entfernt, um zu denen zu kommen, die noch nicht besucht wurden. Das »Problem« der `removeAll()`-Methode ist aber ihre zerstörerische Art – die Elemente werden genau aus der Menge gelöscht. Da die Originalmenge jedoch nicht verändert werden soll, kopieren wir alle Städte in einen Zwischenspeicher (`unvisitedCities`) und löschen aus diesem Zwischenspeicher, was die Originalmenge unangetastet lässt.

### 13.3.2 Methoden der Schnittstelle Set

Eine Mengenklasse deklariert neben Operationen für die Anfrage und das Einfügen von Elementen auch Methoden für Schnitt und Vereinigung von Mengen.

```
interface java.util.Set<E>
 extends Collection<E>
```

- `boolean add(E o)`  
Setzt `o` in die Menge, falls es dort noch nicht vorliegt. Liefert `true` bei erfolgreichem Einfügen.
- `boolean addAll(Collection<? extends E> c)`  
Fügt alle Elemente von `c` in das Set ein und liefert `true` bei erfolgreichem Einfügen. Ist `c` ein anderes Set, so steht `addAll()` für die Mengenvereinigung.
- `void clear()`  
Löscht das Set.
- `boolean contains(Object o)`  
Ist das Element `o` in der Menge?
- `boolean containsAll(Collection<?> c)`  
Ist `c` eine Teilmenge von Set?
- `boolean isEmpty()`  
Ist das Set leer?
- `Iterator<E> iterator()`  
Gibt einen Iterator für das Set zurück.
- `boolean remove(Object o)`  
Löscht `o` aus dem Set, liefert `true` bei erfolgreichem Löschen.

- `boolean removeAll(Collection<?> c)`  
Löscht alle Elemente der Collection aus dem Set und liefert true bei erfolgreichem Löschen.
- `boolean retainAll(Collection<?> c)`  
Bildet die Schnittmenge mit c.
- `int size()`  
Gibt die Anzahl der Elemente in der Menge zurück.
- `Object[] toArray()`  
Erzeugt zunächst ein neues Feld, in dem alle Elemente der Menge Platz finden, und kopiert anschließend die Elemente in das Feld.
- `<T> T[] toArray(T[] a)`  
Ist das übergebene Feld groß genug, dann werden alle Elemente der Menge in das Feld kopiert. Ist das Feld zu klein, wird ein neues Feld vom Typ T angelegt, und alle Elemente werden vom Set in das Array kopiert und zurückgegeben.

In der Schnittstelle Set werden die aus Object stammenden Methoden equals() und hashCode() mit ihrer Funktionalität bei Mengen in der API-Dokumentation präzisiert.



#### Hinweis

In einem Set gespeicherte Elemente müssen immutable bleiben. Einerseits sind sie nach einer Änderung vielleicht nicht wiederzufinden, und andererseits können Elemente auf diese Weise doppelt in der Menge vorkommen, was der Philosophie der Schnittstelle widerspricht.

## 13.4 Assoziative Speicher

Ein assoziativer Speicher verbindet einen Schlüssel mit einem Wert. Java bietet für Datenstrukturen dieser Art die allgemeine Schnittstelle Map mit wichtigen Operationen wie `put(key, value)` zum Aufbau einer Assoziation und `get(key)` zum Erfragen eines assozierten Wertes.

### 13.4.1 Die Klassen HashMap und TreeMap

Die Java-Bibliothek implementiert assoziativen Speicher mit einigen Klassen, wobei wir unser Augenmerk zunächst auf zwei wichtige Klassen richten wollen:

- Eine schnelle Implementierung ist die *Hash-Tabelle* (engl.  *hashtable*), die in Java durch `java.util.HashMap` implementiert ist. Vor Java 1.2 wurde `java.util.Hashtable` verwendet. Die Schlüsselobjekte müssen »hashbar« sein, also `equals()` und `hashCode()` konkret implementieren. Eine besondere Schnittstelle für die Elemente ist nicht nötig.
- Daneben existiert die Klasse `java.util.TreeMap`, die etwas langsamer im Zugriff ist, doch dafür alle Schlüsselobjekte immer sortiert hält. Sie sortiert die Elemente in einen internen Binärbaum ein. Die Schlüssel müssen sich in eine Ordnung bringen lassen, wozu etwas Vorbereitung nötig ist.

### Beispiel

zB

Ein Assoziativspeicher, dem wir Werte<sup>5</sup> hinzufügen:

```
Map<String, String> aldiSupplier = new HashMap<String, String>();
aldiSupplier.put("Carbo, spanischer Sekt", "Freixenet");
aldiSupplier.put("ibu Stapelchips", "Bahlsen Chipsletten");
aldiSupplier.put("Ko-kra Katzenfutter", "felix Katzenfutter");
aldiSupplier.put("Küchenpapier", "Zewa");
aldiSupplier.put("Nuss-Nougat-Creme", "Zentis");
aldiSupplier.put("Pommes Frites", "McCaine");
```

Die zweite `HashMap` soll Strings mit Zahlen assoziieren:

```
Map<String, Number> num = new HashMap<String, Number>();
num.put("zwei", 2); // Boxing durch Integer.valueOf(2)
num.put("drei", 3.0); // Boxing durch Double.valueOf(3.0)
```

13

Während also bei den Assoziativspeichern nach dem Hashing-Verfahren eine `hashCode()`- und `equals()`-Methode bei den Schlüssel-Objekten essenziell ist, ist das bei den Baum-orientierten Verfahren nicht nötig – hier muss nur eine Ordnung zwischen den Elementen entweder mit `Comparable` oder `Comparator` her.

Ein Assoziativspeicher arbeitet nur in einer Richtung schnell. Wenn etwa im Fall eines Telefonbuchs ein Name mit einer Nummer assoziiert wurde, kann die Datenstruktur die Frage nach einer Telefonnummer schnell beantworten. In die andere Richtung dauert es wesentlich länger, weil hier keine Verknüpfung besteht. Sie ist immer nur einseitig. Auf wechselseitige Beziehungen sind die Klassen nicht vorbereitet.

---

<sup>5</sup> Siehe dazu auch [http://www.aldibaran.de/?page\\_id=13#2](http://www.aldibaran.de/?page_id=13#2).

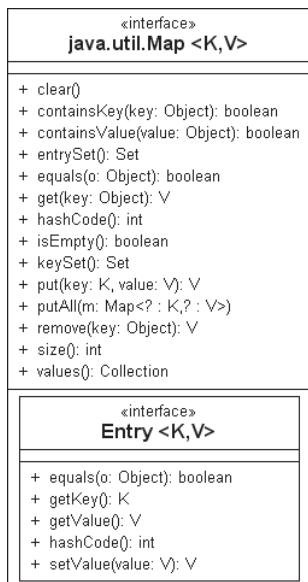


Abbildung 13.4: Klassendiagramm der Schnittstelle Map

### Die Klasse HashMap

Die Klasse `HashMap` eignet sich ideal dazu, viele Elemente unsortiert zu speichern und sie über die Schlüssel schnell wieder verfügbar zu machen. Das interne Hashing-Verfahren ist schnell, eine Sortierung der Schlüssel nach einem gegebenen Kriterium aber nicht möglich.

```

class java.util.HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable

```

- `HashMap()`  
Erzeugt eine neue Hash-Tabelle.
- `HashMap(Map<? extends K, ? extends V> m)`  
Erzeugt eine neue Hash-Tabelle aus einer anderen Map.

### Die Klasse TreeMap und die Schnittstelle SortedMap/NavigableMap

Eine `TreeMap` implementiert seit Java 6 die Schnittstelle `NavigableMap`, die wiederum von der Schnittstelle `SortedMap`<sup>6</sup> erbt, wobei diese wiederum `Map` erweitert. Eine `NavigableMap` sortiert die Elemente eines Assoziativspeichers nach Schlüsseln und bietet Zugriff auf

das kleinste oder größte Element mit Methoden wie `firstKey()`, `lastKey()` und kann mit `subMap()` und `tailMap()` Teilansichten des Assoziativspeichers bilden.

Damit die Schlüssel in einer `TreeMap` sortiert werden können, gilt das Gleiche wie beim `TreeSet`: Die Elemente müssen eine natürliche Ordnung besitzen, oder ein externer `Comparator` muss die Ordnung festlegen.

```
class java.util.TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

- `TreeMap()`  
Erzeugt eine neue `TreeMap`, die eine natürliche Ordnung von ihren Elementen erwartet.
- `TreeMap(Comparator<? super K> comparator)`  
Erzeugt eine neue `TreeMap` mit einem `Comparator`, sodass die Elemente keine natürliche Ordnung besitzen müssen.
- `TreeMap(Map<? extends K, ? extends V> m)`  
Erzeugt eine `TreeMap` mit eingesortierten Elementen aus `m`, die eine natürliche Ordnung besitzen müssen.
- `TreeMap(SortedMap<K, ? extends V> m)`  
Erzeugt eine `TreeMap` mit eingesortierten Elementen aus `m` und übernimmt von `m` auch die Ordnung.

Um die Sortierung zu ermöglichen, ist der Zugriff etwas langsamer als über `HashMap`, aber mit dem Hashing-Verfahren lassen sich Elemente nicht sortieren.

### 13.4.2 Einfügen und Abfragen der Datenstruktur

Wir haben gesagt, dass die Elemente des Assoziativspeichers Paare aus Schlüssel und zugehörigem Wert sind. Das Wiederfinden der Werte ist effizient nur über Schlüssel möglich.

#### Daten einfügen

Zum Hinzufügen von Schlüssel-Werte-Paaren dient die Methode `put(key, value)`. Das erste Argument ist der Schlüssel und das zweite Argument der mit dem Schlüssel zu assoziierende Wert. Der Schlüssel und der Wert können `null` sein.

---

<sup>6</sup> Vor Java 6 war dies die implementierte Schnittstelle.

```
interface java.util.Map<K,V>
```

- `V put(K key, V value)`  
Speichert den Schlüssel und den Wert in der Hash-Tabelle. Falls sich zu diesem Schlüssel schon ein Eintrag in der Hash-Tabelle befand, wird der alte Wert überschrieben und der vorherige Wert zum Schlüssel zurückgegeben (das ist anders als beim Set, wo die Operation dann nichts tut). Ist der Schlüssel neu, liefert `put()` den Rückgabewert `null`. Das heißt natürlich auch, dass mit `put(key, value) == null` nicht klar ist, ob `put()` einen Wert überschreibt und der alte Wert `null` war, oder ob noch kein Schlüssel-Werte-Paar in dem Assoziativspeicher lag.
- `void putAll(Map<? extends K, ? extends V> m)`  
Fügt alle Schlüssel-Werte-Paare aus `m` in die aktuelle Map ein. Auch diese Methode überschreibt unter Umständen vorhandene Schlüssel.

## Daten auslesen

Um wieder ein Element auszulesen, deklariert Map die Operation `get(key)`. Das Argument identifiziert das zu findende Objekt über den Schlüssel, indem dasjenige Objekt aus der Datenstruktur herausgesucht wird, das im Sinne von `equals()` mit dem Anfrageobjekt gleich ist. Wenn das Objekt nicht vorhanden ist, ist die Rückgabe `null`. Allerdings kann auch `null` der mit einem Schlüssel assoziierte Wert sein, da `null` als Wert durchaus erlaubt ist.

### zB Beispiel

Erfrage den Assoziativspeicher nach »zwei«. Das Ergebnis wird ein Number-Objekt sein:

```
Map<String, Number> num = new HashMap<String, Number>();
Number number = num.get("zwei");
if (number != null)
 System.out.println(number.intValue());
```

Mit Generics kann eine Typanpassung entfallen, wenn – wie in unserem Beispiel – Number-Objekte mit dem String assoziiert waren. Wurde der Typ nicht angegeben, ist eine Typanpassung nötig.

```
interface java.util.Map<K,V>
```

- `V get(Object key)`  
Liefert das mit dem entsprechenden Schlüssel verbundene Objekt. Falls kein passendes Objekt vorhanden ist, liefert die Methode `null`.

## Existiert der Schlüssel, existiert der Wert?

Neben `get()` kann auch mit einer anderen Methode das Vorhandensein eines Schlüssels getestet werden: `containsKey()` überprüft, ob ein Schlüssel in der Tabelle vorkommt, und gibt dann ein `true` zurück. Die Implementierung unterscheidet sich nicht wesentlich von `get()`.

Im Gegensatz zu `get()` und `containsKey()`, die das Auffinden eines Werts bei gegebenem Schlüssel erlauben, lässt sich auch nur nach den Werten ohne Schlüssel suchen. Dies ist allerdings wesentlich langsamer, da alle Werte der Reihe nach durchsucht werden müssen. Die Klasse bietet hierzu `containsValue()` an.

```
interface java.util.Map<K,V>
```

- `boolean containsKey(Object key)`

Liefert `true`, falls der Schlüssel in der Hash-Tabelle vorkommt. Den Vergleich auf Gleichheit führt `HashMap` mit `equals()` durch. Demnach sollte das zu vergleichende Objekt diese Methode aus `Object` passend überschreiben. `hashCode()` und `equals()` müssen miteinander konsistent sein. Aus der Gleichheit zweier Objekte unter `equals()` muss auch jeweils die Gleichheit von `hashCode()` folgen.

- `boolean containsValue(Object value)`

Liefert `true`, falls der Assoziativspeicher einen oder mehrere Werte enthält, die mit dem Objekt inhaltlich (also per `equals()`) übereinstimmen.

## Einträge und die Map löschen

Zum Löschen eines Elements gibt es `remove()`, und zum Löschen der gesamten Map gibt es die Methode `clear()`.

```
interface java.util.Map<K,V>
```

- `V remove(Object key)`

Löscht den Schlüssel und seinen zugehörigen Wert. Wenn der Schlüssel nicht in der Hash-Tabelle ist, so bewirkt die Methode nichts. Im letzten Atemzug wird noch der Wert zum Schlüssel zurückgegeben.

- `void clear()`

Löscht die Hash-Tabelle so, dass sie keine Werte mehr enthält.

## Größe und Leertest

Mit `size()` lässt sich die Anzahl der Werte in der Hash-Tabelle erfragen. `isEmpty()` entspricht einem `size() == 0`, gibt also `true` zurück, falls die Hash-Tabelle keine Elemente enthält.

### 13.4.3 Über die Bedeutung von `equals()` und `hashCode()`

Wenn wir Assoziativspeicher wie eine `HashMap` nutzen, dann sollte uns bewusst sein, dass Vergleiche nach dem Hashcode und der Gleichheit durchgeführt werden, nicht aber nach der Identität. Die folgenden Zeilen zeigen ein Beispiel:

**Listing 13.4:** com/tutego/insel/util/map/HashMapAndEquals.java(), main()

```
Map<Point, String> map = new HashMap<Point, String>();
Point p1 = new Point(10, 20);
map.put(p1, "Point p1");
```

Die `HashMap` assoziiert den Punkt `p1` mit einer Zeichenkette. Was ist nun, wenn wir ein zweites Punkt-Objekt mit den gleichen Koordinaten bilden und die `Map` nach diesem Objekt fragen?

```
Point p2 = new Point(10, 20);
System.out.println(map.get(p2)); // ???
```

Die Antwort ist die Zeichenfolge »`Point p1`«. Das liegt daran, dass zunächst der Hashcode von `p1` und `p2` gleich ist. Des Weiteren liefert auch `equals()` ein `true`, sodass dies als ein Fund zu werten ist (das liefert noch einmal einen wichtigen Hinweis darauf, dass immer beide Methoden `equals()` und `hashCode()` in Unterklassen zu überschreiben sind).

Mit etwas Überlegung folgt dieser Punkt fast zwangsläufig, denn bei einer Anfrage ist ja das zu erfragende Objekt nicht bekannt. Daher kann der Vergleich nur auf Gleichheit, nicht aber auf Identität stattfinden.

## 13.5 Mit einem Iterator durch die Daten wandern

Wenn wir mit einer `ArrayList` oder `LinkedList` arbeiten, so haben wir zumindest eine gemeinsame Schnittstelle `List`, um an die Daten zu kommen. Doch was vereinigt eine Menge (`Set`) und eine Liste, sodass sich die Elemente der Sammlungen mit gleichem Programmcode erfragen lassen? Listen geben als Sequenz den Elementen zwar Positio-

nen, aber in einer Menge hat kein Element eine Position. Hier bieten sich *Iteratoren* beziehungsweise *Enumeratoren* an, die unabhängig von der Datenstruktur alle Elemente auslesen – wir sagen dann, dass sie »über die Datenstruktur iterieren«. Und nicht nur eine Datenstruktur kann Daten liefern; eine Dateioperation könnte genauso gut Datengeber für alle Zeilen sein.

In Java gibt es für Iteratoren zum einen die Schnittstelle `java.util.Iterator` und zum anderen den älteren `java.util Enumeration`. Der Enumerator ist nicht mehr aktuell, daher konzentrieren wir uns zunächst auf den Iterator.

| <code>java.util Enumeration &lt;E&gt;</code>               | <code>java.util Iterator &lt;E&gt;</code>              |
|------------------------------------------------------------|--------------------------------------------------------|
| <pre>+ hasMoreElements(): boolean + nextElement(): E</pre> | <pre>+ hasNext(): boolean + next(): E + remove()</pre> |

Abbildung 13.5: Iterator und Enumeration

### 13.5.1 Die Schnittstelle Iterator

Ein Iterator ist ein Datengeber, der über eine Methode verfügen muss, um das nächste Element zu liefern. Dann muss es eine zweite Methode geben, die Auskunft darüber gibt, ob der Datengeber noch weitere Elemente zur Verfügung stellt. Zwei Operationen der Schnittstelle Iterator sind daher:

|          | Hast du mehr?          | Gib mir das Nächste! |
|----------|------------------------|----------------------|
| Iterator | <code>hasNext()</code> | <code>next()</code>  |

Tabelle 13.2: Zwei zentrale Methoden des Iterators

Die Methode `hasNext()` ermittelt, ob es überhaupt ein nächstes Element gibt, und wenn ja, ob `next()` das nächste Element erfragen darf. Bei jedem Aufruf von `next()` erhalten wir ein weiteres Element der Datenstruktur. So kann der Iterator einen Datengeber (in der Regel eine Datenstruktur) Element für Element ablaufen. Übergehen wir ein `false` von `hasNext()` und fragen trotzdem mit `next()` nach dem nächsten Element, bestraft uns eine `NoSuchElementException`.

Prinzipiell könnte die Methode, die das nächste Element liefert, auch per Definition `null` zurückgeben und so anzeigen, dass es keine weiteren Elemente mehr gibt. Allerdings kann `null` dann kein gültiger Iterator-Wert sein, und das wäre ungünstig.

```
interface java.util.Iterator<E>
```

- `boolean hasNext()`  
Liefert true, falls die Iteration weitere Elemente bietet.
- `E next()`  
Liefert das nächste Element in der Aufzählung oder `NoSuchElementException`, wenn keine weiteren Elemente mehr vorhanden sind.
- `void remove()`

Die Schnittstelle `Iterator` erweitert selbst keine weitere Schnittstelle.<sup>7</sup> Die Deklaration ist generisch, da das, was der Iterator liefert, immer von einem bekannten Typ ist.

### zB Beispiel

Die Aufzählung erfolgt meistens über einen Zweizeiler. Da jede Collection eine Methode `iterator()` besitzt, lassen sich alle Elemente wie folgt auf dem Bildschirm ausgeben:

```
Collection<String> set = new TreeSet<String>();
Collections.addAll(set, "Horst", "Schlämmer", "Hape", "Kerkeling");
for (Iterator<String> iter = set.iterator(); iter.hasNext();)
 System.out.println(iter.next());
```

Das erweiterte `for` macht das Ablaufen aber noch einfacher, und der gleiche Iterator steckt dahinter.

### Beim Iterator geht es immer nur vorwärts

Im Gegensatz zum Index eines Felds können wir beim Iterator ein Objekt nicht noch einmal auslesen (`next()` geht automatisch zum nächsten Element), nicht vorspringen beziehungsweise hin und her springen. Ein Iterator gleicht anschaulich einem Datenstrom; wollten wir ein Element zweimal besuchen, zum Beispiel eine Datenstruktur von rechts nach links noch einmal durchwandern, dann müssen wir wieder ein neues Iterator-Objekt erzeugen oder uns die Elemente zwischendurch merken. Nur bei Listen und sortierten Datenstrukturen ist die Reihenfolge der Elemente vorhersehbar.

---

<sup>7</sup> Konkrete Enumeratoren (und Iteratoren) können nicht automatisch serialisiert werden; die realisierenden Klassen müssen hierzu die Schnittstelle `Serializable` implementieren.

**Hinweis**

In Java steht der Iterator nicht auf einem Element, sondern zwischen Elementen.



### 13.5.2 Der Iterator kann (eventuell auch) löschen

Die Schnittstelle `Iterator` bietet prinzipiell die Möglichkeit, das zuletzt aufgezählte Element aus dem zugrunde liegenden Container mit `remove()` zu entfernen. Vor dem Aufruf muss also `next()` das zu löschende Element als Ergebnis geliefert haben. Eine Enumeration kann die aufgezählte Datenstruktur grundsätzlich nicht verändern.

**Beispiel****zB**

Ein `LinkedHashSet` ist eine auf dem `HashSet` basierende Datenstruktur, die sich aber zusätzlich die Einfügereihenfolge merkt. Ein Programm soll die ältesten Einträge löschen und nur noch die neusten zwei Elemente behalten:

```
LinkedHashSet<Integer> set = new LinkedHashSet<Integer>();
set.addAll(Arrays.asList(3, 2, 1, 6, 5, 4));
System.out.println(set); // [3, 2, 1, 6, 5, 4]
for (Iterator<Integer> iter = set.iterator(); iter.hasNext();)
{
 iter.next();
 if (set.size() > 2)
 iter.remove();
}
System.out.println(set); // [5, 4]
```

13

`interface java.util.Iterator<E>`

- `boolean hasNext()`
- `E next()`
- `void remove()`

Entfernt das Element, das der Iterator zuletzt bei `next()` liefert hat. Kann ein Iterator keine Elemente löschen, so löst er eine `UnsupportedOperationException` aus.

In der Dokumentation ist die Methode `remove()` als optional gekennzeichnet. Das heißt, dass ein konkreter Iterator kein `remove()` können muss – auch eine `UnsupportedOperation`-

tionException ist möglich. Das ist etwa dann der Fall, wenn ein Iterator von einer unveränderbaren Datenstruktur kommt.



### Hinweis

Warum es die Methode `remove()` im Iterator gibt, ist eine interessante Frage. Die Erklärung dafür: Der Iterator kennt die Stelle, an der sich die Daten befinden (eine Art Cursor). Darum können die Daten dort auch effizient und direkt gelöscht werden. Das erklärt jedoch nicht unbedingt, warum es keine Einfüge-Methode gibt. Ein allgemeiner Grund mag sein, dass bei vielen Container-Typen das Einfügen an einer bestimmten Stelle keinen Sinn ergibt, etwa bei einem sortierten NavigableSet oder NavigableMap. Dort ist die Einfügeposition durch die Sortierung vorgegeben oder belanglos (beziehungsweise bei HashSet durch die interne Realisierung bestimmt), also kein Fall für einen Iterator. Dazu wirft das Einfügen weitere Fragen auf: vor oder nach dem zuletzt per `next()` gelieferten Element? Soll das neue Element mit aufgezählt werden oder nicht? Soll es auch dann nicht aufgezählt werden, wenn es in der Sortierung erst später an die Reihe käme? Eine Löschen-Methode ist problemloser und universell anwendbar.

## 13.6 Algorithmen in Collections

Um Probleme in der Informatik zu lösen, ist die Wahl einer geeigneten Datenstruktur nur der erste Schritt. Im zweiten Schritt müssen Algorithmen implementiert werden. Da viele Algorithmen immer wiederkehrende (Teil-)Probleme lösen, hilft uns auch hier die Java-Bibliothek mit einigen Standardalgorithmen weiter. Dazu zählen etwa Methoden zum Sortieren und Suchen in Containern und zum Füllen von Containern. Einige Algorithmen sind Teil der jeweiligen Datenstruktur selbst, andere wiederum befinden sich in der Extraklasse `java.util.Collections`. Diese Utility-Klasse, die wir nicht mit dem Interface `Collection` verwechseln dürfen, bietet Methoden, um zum Beispiel

- Listen zu sortieren, zu mischen, umzudrehen, zu kopieren und zu füllen,
- Elemente nach der Halbierungssuche zu finden,
- die Anzahl `equals()`-gleicher Elemente zu ermitteln,
- Extremwerte zu bestimmen,
- Elemente in einer Liste zu ersetzen und
- Wrapper um existierende Datenstrukturen zu legen.

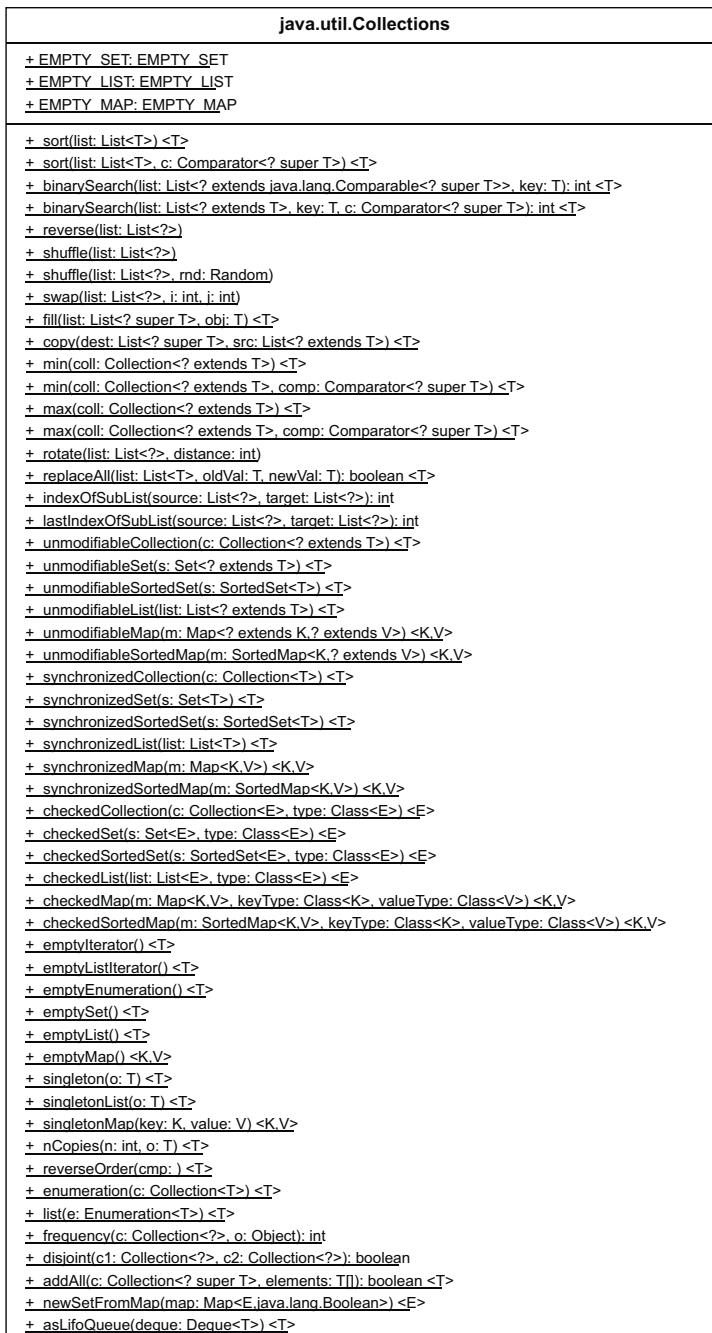


Abbildung 13.6: UML-Diagramm von Collections

Viele Algorithmen sind nur auf List-Objekten definiert, denn der einfache Typ Collection reicht oft nicht aus. Das ist nicht erstaunlich, denn wenn ein Container keine Ordnung definiert, kann er nicht sortiert werden. Auch die binäre Suche erfordert Container mit einer impliziten Reihenfolge der Elemente. Nur Min- und Max-Methoden arbeiten auf allgemeinen Collection-Objekten. Nutzt die Collections-Klasse keine List-Objekte, arbeitet sie doch nur mit Collection-Objekten und nicht mit Iteratoren.

Alle Methoden sind statisch, sodass Collections eine Utility-Klasse wie Math ist. Ein Exemplar von Collections lässt sich nicht anlegen – der Konstruktor ist privat.

### **zB Beispiel**

Fülle eine Liste mit Strings, sortiere die Strings, und suche nach einem String:

```
List<String> list = new ArrayList<String>();
Collections.addAll(list, "Doha,Berlin,Wesel".split(","));
Collections.sort(list);
System.out.println(Collections.binarySearch(list, "Wesel") >= 0); // true
```

#### **13.6.1 Die Bedeutung von Ordnung mit Comparator und Comparable**

Die Ordnung der Elemente spielt bei Daten eine große Rolle. Um Elemente in einem TreeSet sortiert zu halten oder in einer Liste das größte Element zu finden, muss die Ordnung definiert sein.

Um Ordnung herzustellen, unterscheidet Java zwei Wege:

- Elemente können eine *natürliche Ordnung* haben. Dann implementieren Klassen die Schnittstelle Comparable. Beispiele sind String, Date und Integer.
- Ein externes Vergleichsobjekt, das die Schnittstelle Comparator implementiert, stellt fest, wie die Ordnung für zwei Elemente ist.

Um Such- oder Sortieroperationen möglichst unabhängig von Klassen zu machen, die eine natürliche Ordnung besitzen oder die eine Ordnung über einen externen Comparator definiert bekommen, haben Utility-Klassen wie java.util.Arrays oder java.util.Collections oft zwei Arten von Methoden: einmal mit einem zusätzlichen Comparator-Parameter und einmal ohne. Wird kein Comparator angegeben, so müssen die Objekte vom Typ Comparable sein.

**class java.util.Arrays**

- static void sort(Object[] a)  
Sortiert die Elemente. Zum Vergleichen wird vorausgesetzt, dass sie die Klasse Comparable implementieren. Falls sie dies nicht tun, wird eine Ausnahme ausgelöst.
- static <T> void sort(T[] a, Comparator<? super T> c)  
Vergleicht die Objekte mit einem externen Comparator. Falls die Objekte auch noch Comparable implementieren, wird diese Sortierordnung nicht genutzt.
- static int binarySearch(Object[] a, Object key)  
Sucht binär nach key. Die Objekte im Feld müssen Comparable implementieren.
- static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)  
Sucht im sortierten Feld. Der Comparator bestimmt das Sortierkriterium.

**class java.util.Collections**

- static <T extends Comparable<? super T>> void sort(List<T> list)
- static <T> void sort(List<T> list, Comparator<? super T> c)
- static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
- static <T> int binarySearch(List<? extends T> list, T key,  
Comparator<? super T> c)

Die Datenstrukturen, die eine Sortierung verlangen, wie TreeSet oder TreeMap, nehmen entweder einen Comparator entgegen oder erwarten von den Elementen eine Implementierung von Comparable.

### 13.6.2 Sortieren

Mit zwei statischen sort()-Methoden bietet die Utility-Klasse Collections die Möglichkeit, die Elemente einer Liste stabil zu sortieren.

**class java.util.Collections**

- static <T extends Comparable<? super T>> void sort(List<T> list)  
Sortiert die Elemente der Liste gemäß ihrer natürlichen Ordnung, die ihnen die Implementierung über Comparable gibt.
- static <T> void sort(List<T> list, Comparator<? super T> c)  
Sortiert die Elemente der Liste gemäß der Ordnung, die durch den Comparator c festgelegt wird. Eine mögliche natürliche Ordnung spielt keine Rolle.

Die Sortiermethode arbeitet nur mit `List`-Objekten. Bei den anderen Datenstrukturen wäre das ohnehin kaum sinnvoll, weil diese entweder unsortiert sind oder extern eine bestimmte Ordnung aufweisen, wie oben schon angemerkt wurde. Eine analoge Sortiermethode `sort()` für die Elemente von Arrays bietet die Klasse `Arrays`.

### Beispielprogramm zum Sortieren

Das folgende Programm sortiert eine Reihe von Zeichenketten aufsteigend.

**Listing 13.5:** com/tutego/insel/util/CollectionsSortDemo.java, main()

```
List<String> list = Arrays.asList(
 "Saskia", "Regina", "Angela", "Astrid", "Manuela", "Silke",
 "Linda", "Daniela", "Silvia", "Samah", "Radhia", "Mejda", "Tanja"
);
Collections.sort(list);
System.out.println(list);
```

Die statische Methode `Arrays.asList()` baut aus einem String-Feld (getarnt als Vararg) eine Liste auf. Die Liste im Ergebnis ist veränderbar.

### Strings sortieren, auch unabhängig von der Groß- und Kleinschreibung

Die Klasse `String` realisiert über die Implementierung von `Comparable` eine natürliche Sortierung. Alle `String`-Objekte, die in einem Feld sind, können problemlos über `Array.sort()` sortiert werden, und alle Strings in Collection-Sammlungen können über `Collections.sort()` sortiert werden.

Um unabhängig von der Groß- und Kleinschreibung zu sortieren, bietet die Klasse `String` eine praktische Konstante: `String.CASE_INSENSITIVE_ORDER`. Das ist ein `Comparator<String>`, der gut als Argument für `sort()` passt. Im Übrigen ist es die einzige statische Variable der Klasse.

**zB**

#### Beispiel

Füge in ein `TreeSet` eine Liste von `Strings` ein, die sortiert unabhängig von der Groß-/Kleinschreibung gehalten werden.

**Beispiel (Forts.)**

zB

```
<TreeSet<String> set = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
Collections.addAll(set, "noah", "abraham", "Isaak", "Ismael",
 "moses", "JESUS", "Muhammed");
System.out.println(set); // [abraham, Isaak, Ismael, JESUS, moses, Muhammed, noah]
```

**Tipp**

+

Für länderabhängige Vergleiche helfen spezielle Untertypen von Comparator: die Collator-Objekte.

```
List<String> list = Arrays.asList("A", "b", "ä", "ß", "S", "s");
Comparator<Object> collator = Collator.getInstance(Locale.GERMAN);
Collections.sort(list, collator);
System.out.println(list); // [A, ä, b, s, S, ß]
```

13

**Daten in umgekehrter Reihenfolge sortieren**

Da es keine spezielle Methode reverseSort() gibt, ist hier ein spezielles Comparator-Objekt im Einsatz, um Daten entgegengesetzt zu ihrer natürlichen Reihenfolge zu sortieren. Mit der statischen Methode reverseOrder() der Klasse Collections können wir ein geeignetes Comparator-Exemplar anfordern.

**Beispiel**

zB

Sortiere Zeichenfolgen absteigend:

```
List<String> list = new ArrayList<String>();
Collections.addAll(list, "Adam", "Eva", "Set", "Enosch", "Kenan", "Mahalalel");
Comparator<String> comparator = Collections.reverseOrder();
Collections.sort(list, comparator);
System.out.println(list); // [Set, Mahalalel, Kenan, Eva, Enosch, Adam]
```

Eine andere Möglichkeit für umgekehrt sortierte Listen besteht darin, erst die Liste mit Collections.sort() zu sortieren und anschließend mit Collections.reverse(List<?> list) umzudrehen. Das Umdrehen ist jedoch ein zusätzlicher Durchlauf, der mit dem reverseOrder()-Comparator vermieden wird. Zudem ist die Lösung mit einem Comparator

über `reverseOrder()` stabil. Für einen existierenden `Comparator` liefert `Collections.reverseOrder(Comparator<T> cmp)` einen `Comparator<T>`, der genau umgekehrt arbeitet.

### 13.6.3 Den größten und kleinsten Wert einer Collection finden

Bisher kennen wir die überladenen statischen Methoden `min()` und `max()` der Utility-Klasse `Math` für numerische Datentypen. Es gibt aber auch statische Methoden `min()` und `max()` in `Collections` und `Arrays`, die das kleinste und größte Element einer Sammlung ermitteln. Die Laufzeit ist linear zur Größe der Sammlung. Die Methoden unterscheiden nicht, ob die Elemente der Datenstruktur schon sortiert sind oder nicht.

```
class java.util.Collections
■ static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)
■ static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
■ static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
■ static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
```

Wir sehen, dass es eine überladene Version der jeweiligen Methode gibt, da für beliebige Objekte eventuell ein `Comparator`-Objekt erforderlich ist, das den Vergleich vornimmt. Es sei auch bemerkt, dass dies mit die komplexesten Beispiele für Generics sind.

### Aufbauen, Schütteln, Beschneiden, Größensuche

In unserem nächsten Beispiel geht es darum, dass zwei Spieler aus einem Kartenspiel mit 56 Karten je 10 zufällige Karten bekommen. Die Karten sind Java-enums. Derjenige mit der höchstwertigen Karte (ein Ass ist zum Beispiel mehr »wert« als eine Sieben) gewinnt, wobei auch beide gewinnen können, wenn sie die gleiche höchstwertige Karte haben. Auf die Anzahl der Karten kommt es nicht an.

**Listing 13.6:** com/tutego/insel/util/BestCard.java

```
package com.tutego.insel.util;

import java.util.*;

public class BestCard
{
 enum Cards { ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
 NINE, TEN, JACK, QUEEN, KING, ALAS }
```

```
public static void main(String[] args)
{
 // Initialisiere Kartenspiel, beginne mit 14 Karten
 List<Cards> cards = new ArrayList<Cards>(EnumSet.allOf(Cards.class));

 // Verdopple zweimal die Karten auf insgesamt 56
 cards.addAll(cards);
 cards.addAll(cards);

 // Vermische Karten
 Collections.shuffle(cards);

 // Der erste Spieler bekommt die ersten 10 Karten
 List<Cards> player1 = new ArrayList<Cards>(cards.subList(0, 10));

 // Der zweite Spieler bekommt die nächsten 10 Karten
 List<Cards> player2 = new ArrayList<Cards>(cards.subList(11, 20));

 // Größte Karte suchen, also die Karte mit der größten Ordinalzahl
 System.out.printf("Spieler 1: %s%nSpieler 2: %s%n", player1, player2);
 Cards bestCardPlayer1 = Collections.max(player1);
 Cards bestCardPlayer2 = Collections.max(player2);
 System.out.println("Beste Karte Spieler 1: " + bestCardPlayer1);
 System.out.println("Beste Karte Spieler 2: " + bestCardPlayer2);

 // Enums implementieren Comparable. Ausgeben, wer die beste Karte hat
 int winner = bestCardPlayer1.compareTo(bestCardPlayer2);
 if (winner > 0)
 System.out.println("Spieler 1 gewinnt");
 else if (winner < 0)
 System.out.println("Spieler 2 gewinnt");
 else
 System.out.println("Beide Spieler gewinnen");
}
```

Das Beispiel baut im ersten Schritt eine Liste mit Card-Objekten auf und schüttelt diese mit `shuffle()` durch. Dann ordnet `subList()` je zehn Karten dem ersten und dem zweiten Spieler zu. In dem Kontext wäre nicht unbedingt eine Kopie in eine neue `ArrayList` nötig gewesen, doch das ist bei einer Weiterführung des Beispiels vermutlich nützlicher, denn `subList()` ist eine Live-Ansicht, und wir könnten sonst keine Elemente aus der Unterliste löschen und hinzufügen, ohne dabei gleichzeitig die gesamten 56 Karten anzugreifen.

Aufzählungen sind `Enum`-Objekte, die `Comparable` implementieren und daher eine natürliche Ordnung haben. Aus diesem Grunde funktioniert `max()` überhaupt, was ein Ordnungskriterium zur Extremwertsuche benötigt. Aus den Karten der beiden Spieler ist so die größte Karte leicht ermittelt.

### Implementierung der Extremwertmethoden bei Comparable-Objekten

Wenn wir ein `String`-Objekt in eine Liste packen oder ein `Double`-Objekt in eine Menge, werden sie korrekt gesucht, da insbesondere die Wrapper-Klassen die Schnittstelle `Comparable` implementieren.

An der Implementierung `Collections.min()` ohne extra `Comparator` lässt sich gut der Aufruf von `compareTo()` ablesen:

**Listing 13.7:** `java/util/Collections.java, min()`

```
public static <T extends Object & Comparable<? super T>>
 T min(Collection<? extends T> coll)
{
 Iterator<? extends T> i = coll.iterator();
 T candidate = i.next();

 while(i.hasNext())
 {
 T next = i.next();
 if (next.compareTo(candidate) < 0)
 candidate = next;
 }
 return candidate;
}
```

Die generische Schreibweise verlangt, dass die Elemente in der Collection vom Typ Comparable sein müssen und somit eine `compareTo()`-Methode vorhanden ist.

## 13.7 Zum Weiterlesen

»Java 7 – Mehr als eine Insel« behandelt ausführlich Datenstrukturen und Algorithmen und diskutiert gleichzeitig, wie Threads und Datenstrukturen effizient zusammenarbeiten.





## Kapitel 14

# Einführung in grafische Oberflächen

»Wenn die Reklame keinen Erfolg hat, muss man die Ware ändern.«  
– Edgar Faure (1908–1988)

## 14.1 Das Abstract Window Toolkit und Swing

Die Programmiersprache Java, die sich das Ziel gesetzt hat, plattformunabhängige Softwareentwicklung zu unterstützen, muss auch eine Bibliothek anbieten, um grafische Oberflächen zu gestalten. Eine Bibliothek sollte dabei im Wesentlichen die folgenden Bereiche abdecken:

- Sie beherrscht das Zeichnen grafischer Grundelemente wie Linien und Polygone und ermöglicht das Setzen von Farben und die Auswahl von Zeichensätzen.
- Sie bietet grafische Komponenten (GUI-Komponenten), auch *Steuerelemente* oder *Widgets* genannt, wie zum Beispiel Fenster, Schaltflächen, Textfelder und Menüs.
- Sie definiert ein Modell zur Behandlung von Ereignissen, wie etwa Mausbewegungen.

14

### 14.1.1 SwingSet-Demos

Um sich einen Überblick über die Swing-Komponenten zu verschaffen, hat Oracle unter den JFC-Demos des JDK (etwa *C:\Program Files\Java\jdk1.7.0\demo\jfc*) verschiedene Beispiele veröffentlicht. (Achtung: Der Ordner ist nur dann vorhanden, wenn bei der Installation explizit die Demos mitinstalliert wurden.) Die in den weiteren Unterordnern enthaltenen Demos sind als Jar-Datei verpackt und können mit einem Doppelklick gestartet werden. Seit Java 6 Update 10 ist *SwingSet3* mit dabei, ein interessantes Swing-Demo, das über die Technologie WebStart aus dem Internet gestartet wird. Die *readme.html*-Datei referenziert auf die URL <http://download.java.net/javadesktop/swingset3/SwingSet3.jnlp>, die wir auch in den Browser einsetzen können, um das Beispiel zu starten.



Abbildung 14.1: SwingSet3 Demo

### 14.1.2 Abstract Window Toolkit (AWT)

Die erste API zum Aufbau grafischer Oberflächen war das *Abstract Window Toolkit* (AWT). Sie bietet Methoden für die Primitivoperationen zum Zeichnen, zur Ereignisbehandlung und einen Satz von GUI-Komponenten. Da das AWT jedoch sehr einfach gehalten ist und professionelle Oberflächen nur mit Mühe erstellbar sind, sind für die Abkürzung »AWT« noch einige hämische Deutungen im Umlauf: »Awful Window Toolkit«, »Awkward Window Toolkit« oder »Annoying Window Toolkit«.

#### Peer-Klassen

Eine Besonderheit des AWT ist, dass es jede grafische Komponente in Java auf eine Komponente der darunterliegenden Plattform abbildet. Dazu bedient sich das AWT sogenannter *Peer-Klassen*, also Partnern auf der Seite der speziellen Benutzeroberfläche. Eine Schaltfläche unter AWT leitet somit die Visualisierung und Interaktion an eine Peer-Klasse auf der Betriebssystemseite weiter. Damit sehen AWT-Anwendungen unter Windows so aus wie jede andere Windows-Anwendung, und für Anwendungen unter Mac OS oder X11 gilt das Gleiche.

Die Partner haben Vor- und Nachteile:

- Durch die nativen Peer-Klassen verhält sich die Oberfläche exakt so wie erwartet und ist optisch nicht von anderen nativen Programmen zu unterscheiden.
- Leider zeigen die Programme unter den verschiedenen Betriebssystemen bisweilen merkwürdige Seiteneffekte. So kann ein Textfeld unter Windows weniger als 64 KiB Zeichen aufnehmen, bei anderen Oberflächen ist dies egal.
- Da das AWT auch nur Komponenten anbietet, die auf jeder Plattform verfügbar sind, ist das Angebot an Widgets sehr beschränkt. Moderne grafische Elemente, sei es auch nur ein Icon auf einer Schaltfläche, bietet das AWT nicht an.

Da jede AWT-Komponente Ressourcen von der nativen Plattform bezieht und diese außerhalb der Speicherverwaltung von Java liegen, nennen sich diese Komponenten *schwergewichtige Komponenten* (engl. *heavyweight components*).

### 14.1.3 Java Foundation Classes

Obwohl das Abstract Window Toolkit das Problem einer einheitlichen Benutzeroberfläche lösen sollte, ist dies Sun damals nicht ganz gelungen. Das AWT war von Anfang an zusammengepfuscht. So meinte auch James Gosling:

*»The AWT was something we put together in six weeks to run on as many platforms as we could, and its goal was really just to work. So we came out with this very simple, lowest-common-denominator thing that actually worked quite well. But we knew at the time we were doing it that it was really limited. After that was out, we started doing the Swing thing, and that involved working with Netscape and IBM and folks from all over the place.«<sup>1</sup>*

Von AWT 1.02 auf AWT 1.1 wurde ein anderes Ereignismodell eingeführt, das die Basis für Swing legte.

Da Sun das AWT einfach hielt, Entwickler von Oberflächen jedoch einen unstillbaren Hunger nach Komponenten haben, konzipierte Netscape die *Internet Foundation Classes* (IFC), die das AWT in wesentlichen Punkten ergänzten. Im April des Jahres 1997 einigten sich Sun, Netscape und IBM auf eine GUI-Bibliothek, die auf Netscapes IFC aufbaut

---

<sup>1</sup> Das Interview vom 24. März 1998 ist leider unter [http://java.sun.com/javaone/javaone98/keynotes/gosling/transcript\\_gosling.html](http://java.sun.com/javaone/javaone98/keynotes/gosling/transcript_gosling.html) nicht mehr online – Oracle hat die Seite gelöscht.

und das AWT in der Java-Version 1.2 erweitert. Der Name des Toolkits, mit dem wir heute noch arbeiten, ist JFC (*Java Foundation Classes*).

### Bestandteile der Java Foundation Classes

Die Java Foundation Classes bestehen im Wesentlichen aus:

- **Swing-GUI-Komponenten:** Unter die Swing-Set-Komponenten fallen ganz neue grafische Elemente. Diese sind, anders als die plattformabhängigen Peer-Komponenten des herkömmlichen AWT, fast vollständig in Java implementiert. Während viele Swing-Komponenten gar keine Beziehung zu AWT-Komponenten haben, gilt das nicht für alle. Ein `javax.swing.JFrame` basiert zum Beispiel auf der AWT-Komponente `java.awt.Frame`, denn `JFrame` ist eine Unterklasse von `Frame`.
- **Pluggable Look & Feel:** Dies gibt uns die Möglichkeit, das Aussehen der Komponenten zur Laufzeit zu ändern, ohne das Programm neu zu starten. Alle Komponenten des Swing-Sets haben diese Fähigkeit automatisch.
- **Java 2D API:** Die 2D-Klassenbibliothek ist eine neue Technik, die über eine Objektbeschreibung – ähnlich wie PostScript – Objekte bildet und diese auf dem Bildschirm darstellt. Zu den Fähigkeiten der Bibliothek gehört es, komplexe Objekte durch Pfade zu bilden und darauf Bewegungs- und Verschiebeoperationen anzuwenden.
- **Drag & Drop:** Daten können mittels Drag & Drop leicht von einer Applikation zur anderen übertragen werden. Dabei profitieren Java-Programme auch davon, Daten zu nutzen, die nicht aus Java-Programmen stammen.
- **Accessibility** (Unterstützung für Menschen mit Behinderungen): Diese API erlaubt mit neuen Interaktionstechniken den Zugriff auf die JFC- und AWT-Komponenten. Zu diesen Techniken zählen unter anderem Lesegeräte für Blinde, eine Lupe für den Bildschirm und auch die Spracherkennung.

Swing-Komponenten sind ein wesentlicher Bestandteil der JFC, und oft wird in der Öffentlichkeit »Swing« als Synonym für JFC verstanden.

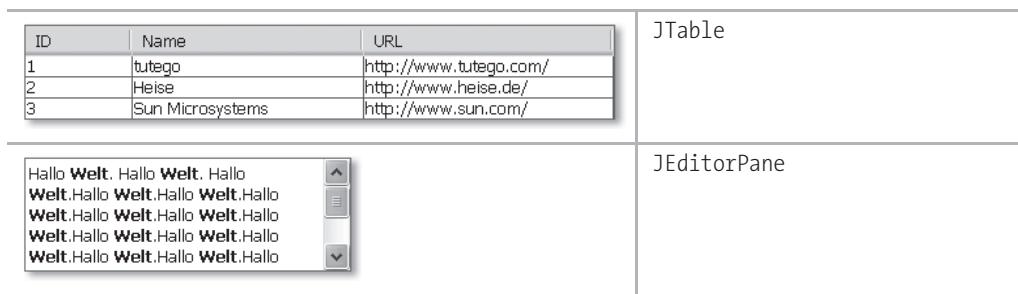


#### Warum Swing Swing heißt

Als 1997 in San Francisco auf der JavaOne die neuen Komponenten vorgestellt wurden, entschied sich Georges Saab, ein Mitglied des JFC-Teams, für Musik parallel zur Präsentation, und zwar für Swing-Musik, weil der Entwickler glaubte, dass sie wieder in Mode käme. Dementsprechend wurden die neuen grafischen Elemente in einem Paket namens *Swing* abgelegt. Obwohl der Name offiziell dem Kürzel JFC weichen musste, war er doch so populär, dass er bestehen blieb.

## Übersicht über Swing-Komponenten

|  |                |
|--|----------------|
|  | JLabel         |
|  | JButton        |
|  | JCheckBox      |
|  | JRadioButton   |
|  | JTextField     |
|  | JPasswordField |
|  | JComboBox      |
|  | JScrollBar     |
|  | JSlider        |
|  | JSpinner       |
|  | JProgressBar   |
|  | JList          |
|  | JTabbedPane    |
|  | JToolBar       |
|  | JMenu          |
|  | JScrollPane    |
|  | JTree          |



## Leichtgewichtige Swing-Komponenten

Eine *Leichtgewicht-Komponente* (engl. *lightweight component*) verfügt über keinen Peer, also über keine direkte Repräsentation im Fenstersystem. Somit gibt es keine speziellen Implementierungen des Systems beispielsweise auf Windows, Mac OS oder X11. Alle Komponenten werden mit primitiven Zeichenoperationen gemalt, so etwa eine Schaltfläche aus einem Rechteck mit Schatten und einem Text in der Mitte. Ein Vorteil: Eine Leichtgewicht-Komponente kann durchsichtig sein und muss nicht mehr in einen rechteckigen Bereich passen. Da alle Komponenten nun gemalt werden, lässt sich alles ohne Rücksicht auf das zugrunde liegende grafische System zeichnen. Dieser Weg ist also plattformunabhängiger, aber nutzt nicht unbedingt alle optimalen Möglichkeiten, wie zum Beispiel Antialiasing, des Betriebssystems oder die Möglichkeiten einer Plattformkomponente, wie den komplexeren Dateiauswahldialog.

### 14.1.4 Was Swing von AWT unterscheidet

Wir werden in diesem Buch nicht mit AWT-Komponenten arbeiten, aber dennoch wesentliche Unterschiede aufzählen:

- Swing bietet viel mehr Komponenten als AWT. Das AWT bietet zum Beispiel keine Tabellen oder Bäume.
- Schaltflächen und Labels nehmen Symbole auf, die sie beliebig um Text angeordnet darstellen.
- Swing-Komponenten können transparent und beliebig geformt sein; eine Schaltfläche kann wie unter Mac OS X abgerundet sein.
- Jede Swing-Komponente kann einen Rahmen bekommen.
- AWT-Komponenten arbeiten nicht nach dem Model/View-Prinzip, nach dem die Daten getrennt von den Komponenten gehalten werden.

- Die AWT-Methoden sind thread-sicher, es können also mehrere Threads zur gleichen Zeit Methoden der AWT-Komponenten aufrufen. Die meisten Swing-Methoden sind nicht thread-sicher, und Entwickler müssen darauf achten, dass Parallelität keine problematischen Zustände erzeugt.

### Wofür Java in der Praxis eingesetzt wird

Mit den JFC lassen sich attraktive, gut funktionierende grafische Oberflächen entwickeln. Eine Untersuchung der Evans Data Corporation aus dem Jahr 2005 fand heraus, dass sich 43 % der Java-Entwickler mit Desktop-Applikationen beschäftigen, 41 % mit Java EE-Technologien und 4 % mit Mobile Java. Die Untersuchung beweist, dass Java nicht ausschließlich im Bereich Middleware (Stichwort Java EE) zu finden ist, sondern eine ausgezeichnete Umgebung für GUI-Applikationen unter Windows, Linux, Mac OS X ... bildet.



#### 14.1.5 GUI-Builder für AWT und Swing

Der Bau von grafischen Oberflächen in Java weist die Besonderheit auf, dass das Design der Oberfläche in Java-Code gegossen werden muss. Jede Komponente muss mit `new` erzeugt werden und mithilfe eines Layouts explizit angeordnet werden. Wir nennen das *programmierte Oberflächen*. Die Änderung des Layouts ist natürlich sehr schwierig, da mitunter auch für kleinste Änderungen viel Quellcode bewegt wird.

Programmierte Oberflächen stehen im Gegensatz zu *deklarativen Oberflächen*, bei denen die Beschreibung des Layouts und die Anordnung der Komponenten nicht in Java formuliert wird, sondern in einer externen Datei. Die Beschreibung kann etwa im XML-Format sein, die dann beschreibt, wie das Objektgeflecht aussieht.

Für deklarative Oberflächen hat sich in den letzten Jahren kein Standard gebildet, und Oberflächen werden heute noch so programmiert wie von 15 Jahren. Eine Sache hat jedoch die Entwicklung massiv vereinfacht: GUI-Builder. Diese Softwarelösungen bieten eine WYSIWYG-Oberfläche mit allen Komponenten, und Entwickler können sich jedes Layout zusammenklicken. Im Hintergrund erzeugt der GUI-Builder den Programmcode. Für die Laufzeitumgebung hat sich also nichts verändert, aber für uns schon.

14

#### Auswahl von GUI-Buildern

NetBeans ist neben Eclipse eine bekannte Java-Entwicklungsumgebung. Sie bietet eine sehr gute Unterstützung im Entwurf grafischer Oberflächen und gibt uns eine gute

Möglichkeit, Swing spielerisch zu erfahren.<sup>2</sup> Ein GUI-Bilder ist gleich integriert, und eine Zusatzinstallation ist nicht nötig.

Bei Eclipse ist standardmäßig kein GUI-Builder integriert, weder in der normalen Eclipse-Version noch in der Java EE-Version. Es gilt also, ein Plugin nachzuinstallieren. In den letzten Jahren kamen und gingen verschiedene GUI-Builder, aber letztendlich hat sich der Windows Builder (<http://code.google.com/intl/de-DE/javadevtools/wbpro/index.html>) von Google als De-facto-Standard etabliert. Über den Update-Mechanismus von Eclipse wird er installiert. Eine Installationsanleitung findet sich unter <http://code.google.com/intl/de-DE/javadevtools/wbpro/installation/index.html>.

## 14.2 Mit NetBeans zur ersten Oberfläche

Ohne uns daher groß mit den Klassen auseinanderzusetzen, wollen wir ein erstes Beispiel programmieren und das Swing-Wissen sozusagen im Vorbeigehen mitnehmen. Das eigene Programm *Bing*, das im Folgenden entwickelt wird, bietet einen Schieberegler, mit der sich eine Zeit einstellen lässt, nach dem eine Meldung auf dem Bildschirm erscheint. Wer vor lauter Java immer vergisst, den Teebeutel aus der Tasse zu nehmen, für den ist diese Applikation genau richtig!



### Hinweis

NetBeans nutzt keine proprietären Klassen, sodass der Programmcode 1:1 auch in ein Eclipse-Projekt kopiert werden kann und dort ohne Anpassung läuft. Auch funktioniert es problemlos, zwei Entwicklungsumgebungen, also NetBeans und Eclipse, auf einen Projektordner »loszulassen«. Am einfachsten geht das so: Erst wird ein Java-Projekt mit NetBeans angelegt. Dann wird unter Eclipse ein Java-Projekt angelegt, aber der Projekt-pfad auf das Verzeichnis des existierenden NetBeans-Projekt gelegt. Dann können beide IDEs gleichzeitig das gleiche Projekt verarbeiten.

### 14.2.1 Projekt anlegen

Nach dem Start von NetBeans wählen wir FILE • NEW PROJECT.

---

<sup>2</sup> Didaktiker nennen das »exploratives Lernen«.

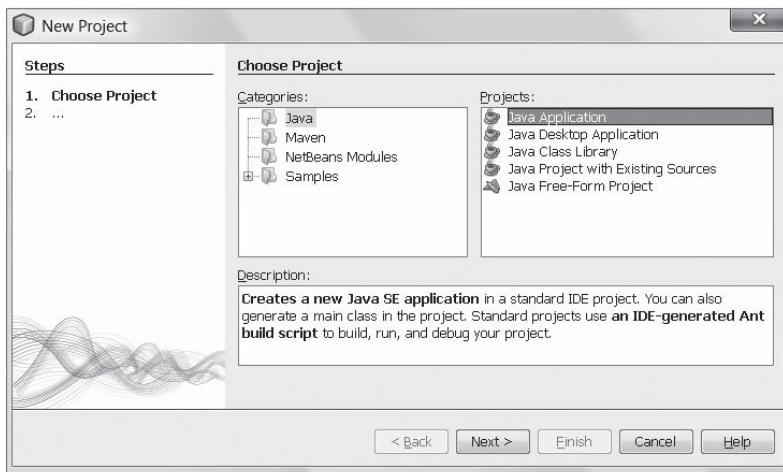


Abbildung 14.2: Projekttyp in NetBeans auswählen

Anschließend wählen wir JAVA APPLICATION und dann NEXT. Den Projektnamen und Paketnamen setzen wir auf etwas Sinnvolleres, die Einstellungen könnten aber auch so bleiben:

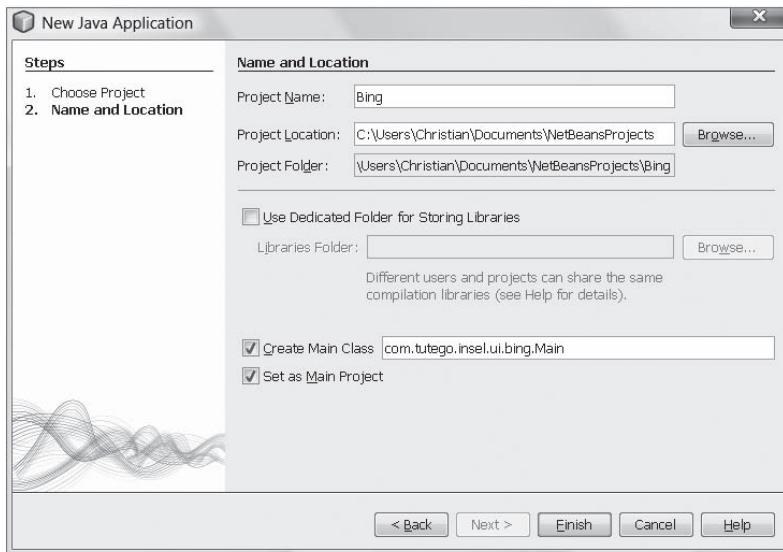


Abbildung 14.3: Neues Java-Projekt anlegen

Nach FINISH öffnet NetBeans den Editor mit der Klasse. **[F6]** startet das Programm, aber ohne Funktion bleibt das langweilig.

### 14.2.2 Eine GUI-Klasse hinzufügen

Fügen wir eine GUI-Klasse hinzu. Dazu wählen wir FILE • NEW FILE... und im Dialog anschließend bei SWING GUI FORM den Typ JFRAME FORM.

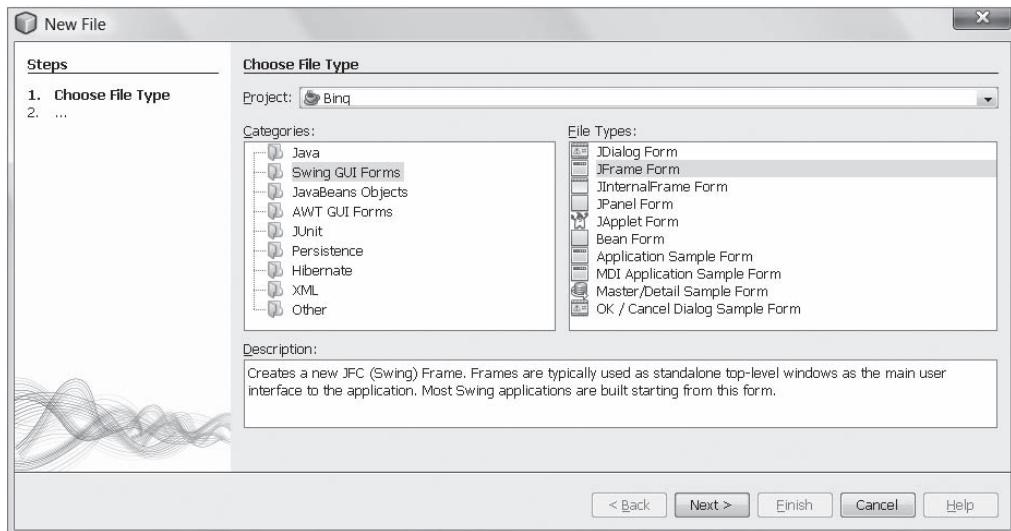


Abbildung 14.4: Swing-Form in das Projekt einfügen

Nach NEXT geben wir einen passenden Klassennamen ein:

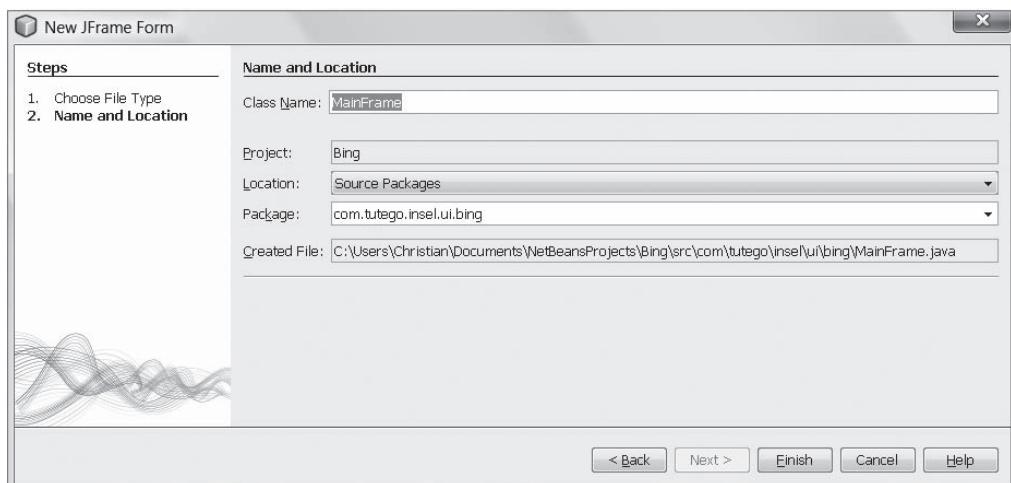


Abbildung 14.5: Klassename und Paket für das JFrame wählen

NetBeans erzeugt eine neue Klasse und öffnet den *grafischen Editor*, der auch *Matisse* heißt.

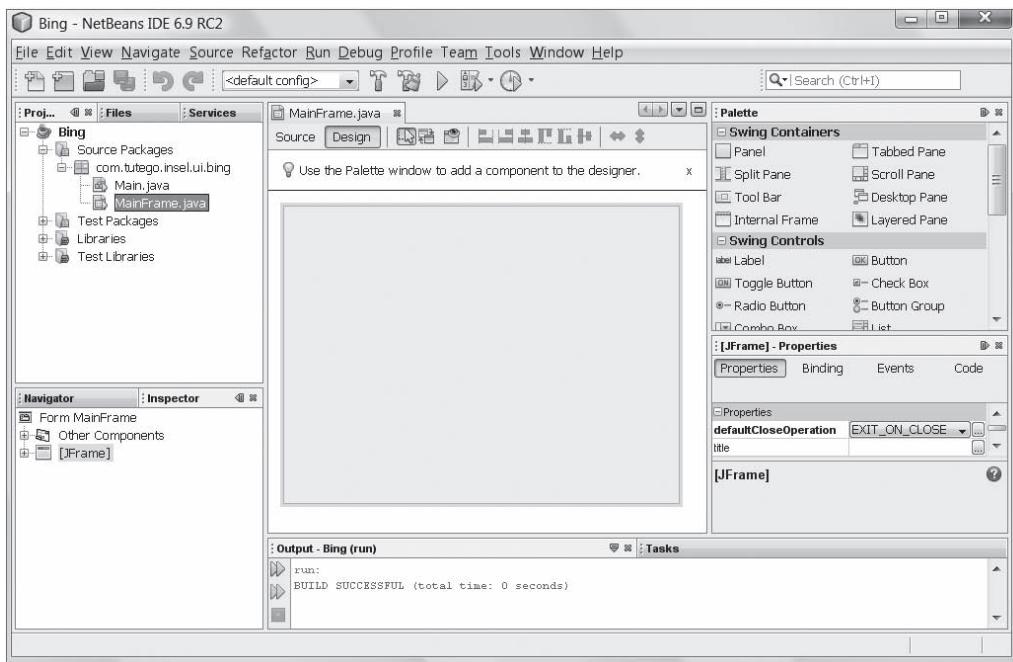


Abbildung 14.6: NetBeans-Oberfläche mit dem Formular und der Komponentenpalette

NetBeans zeigt in unterschiedlichen Ansichten unterschiedliche Details an. In der Mitte steht die Hauptansicht mit dem grafischen Editor. Rechts sind unter PALETTE die Komponenten aufgelistet, die wir per Drag & Drop auf den Formular-Designer ziehen können. Ebenfalls rechts bei PROPERTIES finden wir die Eigenschaften von Komponenten, etwa den Titel des Fensters.

Interessant an Matisse ist, dass die grafische Oberfläche direkt in Quellcode gegossen wird. Den Quellcode können wir einsehen, indem wir von DESIGN auf SOURCE wechseln.

Auffällig sind graue Blöcke, die geschützt sind. Der Grund ist, dass NetBeans den Quellcode aktualisiert, wann immer es über den GUI-Designer Veränderungen gibt. Den Quellcode direkt zu ändern, wäre töricht, denn so könnte NetBeans mitunter die Quellen nicht mehr einlesen, und das ganze Projekt wäre kaputt.

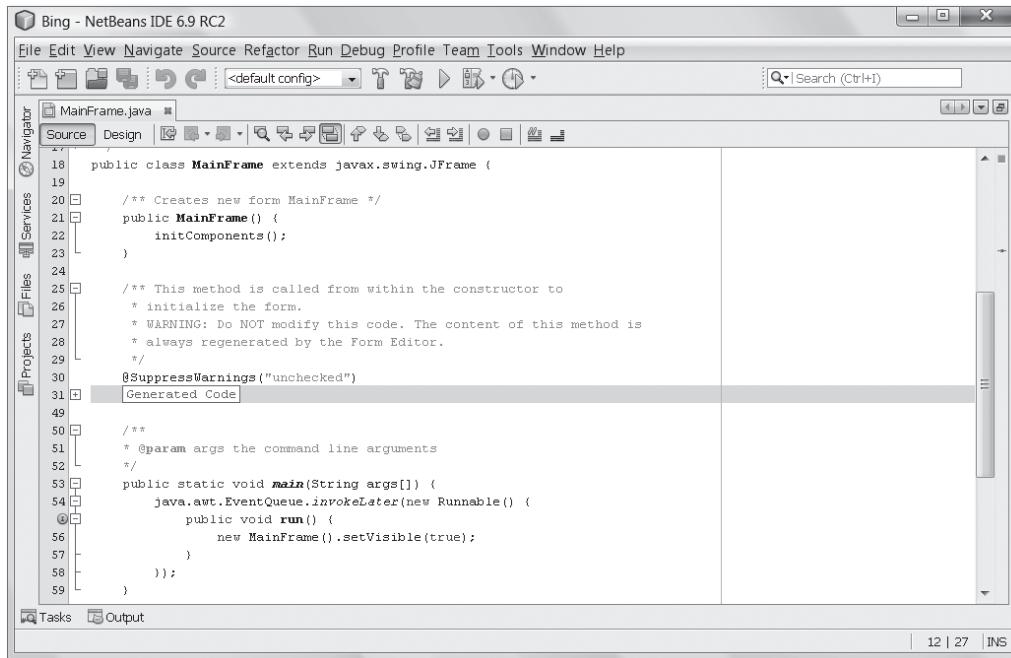


Abbildung 14.7: Code-Editor und geschützte Bereiche

### 14.2.3 Programm starten

Im Quellcode lässt sich ablesen, dass die Klasse schon eine `main()`-Methode hat, sodass wir `MainFrame` starten können. Drei Varianten bieten sich an:

1. Mit **Shift** + **F6** lässt sich direkt das Programm starten, dessen Editor offen ist. Ist es die Klasse `MainFrame`, bekommen wir anschließend ein leeres Fenster.
2. In unserer eigentlichen Hauptklasse, die bei NetBeans eingetragen ist und standardmäßig mit **F6** startet, lässt sich eine Umleitung einbauen, sodass in die dortige `main()`-Methode ein `MainFrame.main(args);` kommt.
3. Die Klasse `MainFrame` lässt sich als Startklasse eintragen. Dazu tragen wir vom Projekt BING im Kontextmenü **PROPERTIES** im Zweig **RUN** bei **MAIN CLASS** statt `com.tutego.insel.ui.bing.Main` die Klasse `com.tutego.insel.ui.bing.MainFrame` ein.

Welche Variante es wird, ist für das Demo egal; Variante 2 ist nicht schlecht. Mit **F6** springt dann ein unspektakuläres leeres Fenster auf.

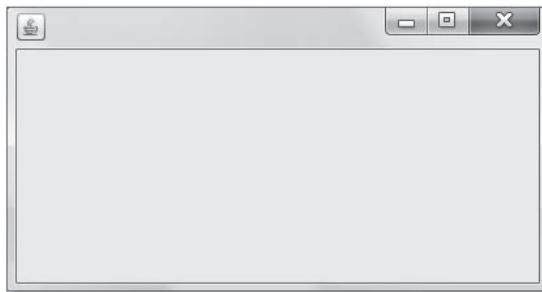


Abbildung 14.8: Nach dem Start gibt es nur ein leeres Fenster.

Eine Vorschau gibt es übrigens auch. Rechts neben den Schaltflächen für SOURCE und DESIGN gibt es ein kleines Fenster mit Auge, das über einen Klick einen ersten Eindruck vom Design vermittelt.

#### 14.2.4 Grafische Oberfläche aufbauen

Kommen wir zurück zum Designer. In der Palette bei SWING CONTROLS suchen wir LABEL und ziehen es per Drag & Drop auf die graue Designerfläche. Bemerkenswert ist, dass Matisse vorgibt, was eine gute Position für die Beschriftung ist. Positionieren wir sie links oben, so rastet sie quasi ein.

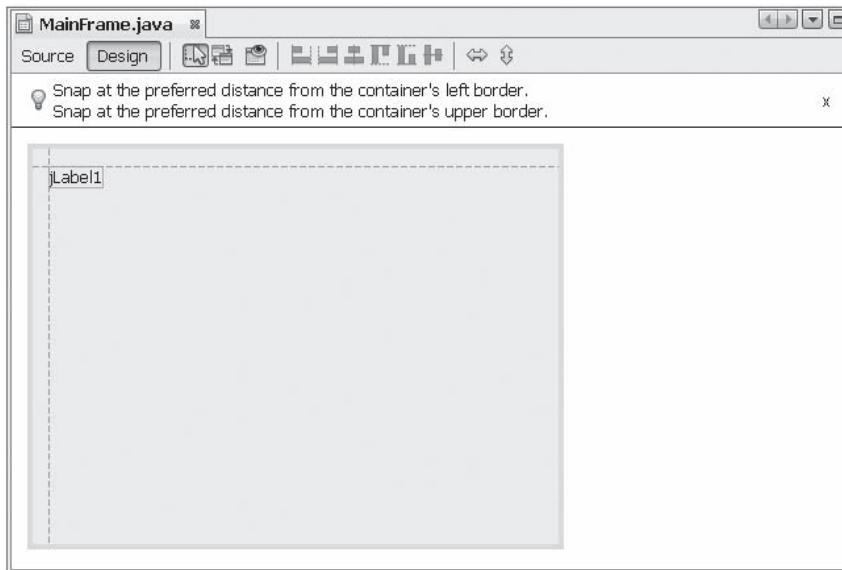


Abbildung 14.9: Das Anordnen der Komponenten ist »magnetisch«.

Das hat zwei Konsequenzen: Zum einen ergibt sich automatisch eine gut aussehende Oberfläche mit sinnvollen Abständen, und zum anderen »kleben« die Komponenten so aneinander, dass sie bei einer Größenanpassung nicht auseinandergerissen werden.

Nachdem das Label positioniert ist, geben wir ihm einen Namen. Dazu kann rechts bei den PROPERTIES der Text verändert werden oder auch im Designer über einen Doppelklick auf den Text.

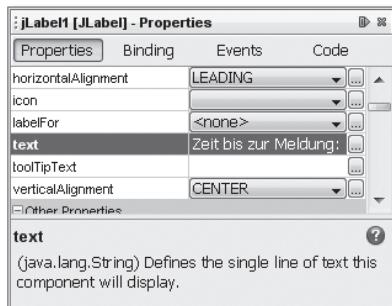


Abbildung 14.10: Eigenschaften des JLabel-Objekts

Jetzt, wo die erste Beschriftung steht, komplettieren wir die GUI. Unter der Beschriftung setzen wir einen SLIDER, allerdings nicht auf die ganze Breite, sondern etwa bis zur Hälfte. Unter den PROPERTIES auf der rechten Seite gibt es Eigenschaften für MINIMUM (0) und MAXIMUM (100). Das Minimum 0 erhöhen wir auf 1 und das Maximum auf 1.440 (24 Stunden sollten reichen).

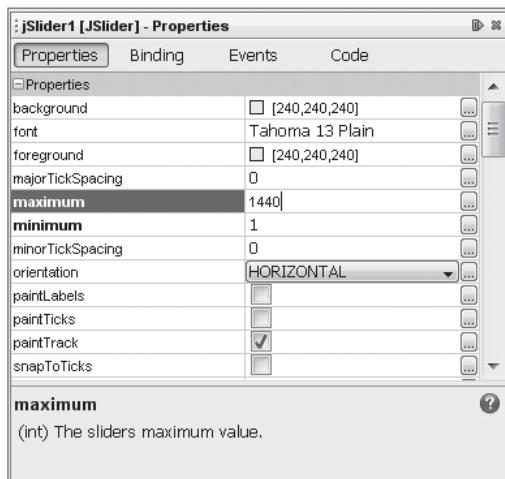


Abbildung 14.11: Eigenschaften des JSlider-Objekts

Rechts vom SLIDER setzen wir ein TEXT FIELD und wiederum rechts davon ein neues LABEL. Das Label hängt am rechten Fensterrand, und wir beschriften es mit MINUTEN. Den Inhalt des Textfeldes (jTextField1) löschen wir mit einem Doppelklick in die TextBox (oder rechts bei den PROPERTIES unter TEXT). Die TextBox wird dann klein, doch wir können sie etwas größer ziehen. Anschließend wird die TextBox rechts an das Minuten-Label und der SLIDER rechts an die TextBox gesetzt, sodass alle drei gut ausgerichtet sind. Bei einem Klick auf das magische Auge sollte die Vorschau so aussehen:

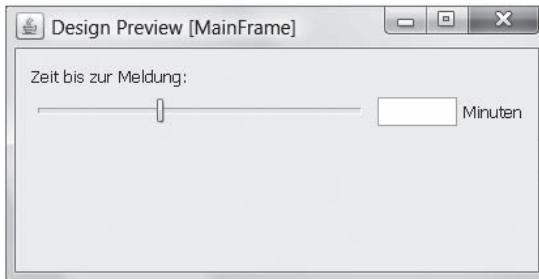


Abbildung 14.12: Anwendung jetzt mit Schieberegler und Textfeld

Das Schöne an den automatischen Ausrichtungen ist, dass wir die Breite verändern können und die Komponenten alle mitlaufen, also nicht absolut positioniert sind; so sollte eine grafische Oberfläche sein!

Die Oberfläche ist jetzt schon fast fertig. Geben wir noch zwei Labels und zwei Schaltflächen hinzu.

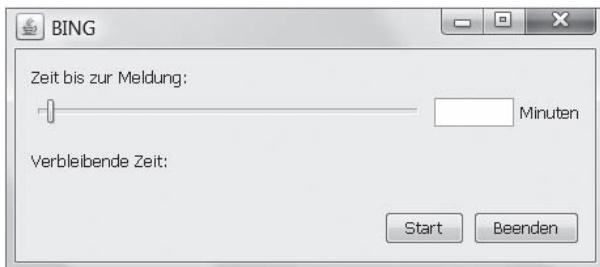


Abbildung 14.13: Vollständige Anwendung

Zwei Labels nebeneinander (das zweite enthält nur ein Leerzeichen) stehen unter dem SLIDER. Startet später die Anwendung, soll im jetzt unsichtbaren Label die Restzeit eingeblendet werden. Die Schaltflächen sind bei den SWING CONTROLS als BUTTON aufgeführt. Zwei soll es geben: eine zum Starten der Applikation und eine zum Beenden. Über

den Property-Editor geben wir gleichzeitig noch dem Fenster einen Titel (BING), und fertig ist die Oberfläche.

### 14.2.5 Swing-Komponenten-Klassen

Die Oberfläche ist jetzt fertig, und in der Ansicht SOURCE lässt sich ablesen, dass viel Quellcode für die Ausrichtung erstellt wurde. Der Quellcode gliedert sich in folgende Teile:

- einen Standardkonstruktor: Er ruft `initComponents()` auf. Eigene Funktionalität können wir hier hinzuschreiben.
- die Methode `initComponents()`: Sie ist geschützt und initialisiert die Komponenten und setzt sie auf den `JFrame`.
- Die statische `main()`-Methode könnte das Fenster gleich starten, denn sie baut ein Exemplar der eigenen Klasse, die ja Unterklasse von `JFrame` ist, auf und zeigt es mit `setVisible(true)` an.
- Am Ende finden sich die Komponenten. Es sind Objektvariablen, sodass jede Objektmethode auf sie Zugriff hat. Sie sehen etwa so aus:

```
// Variables declaration - do not modify
private javax.swing.JButton jButton1;
private javax.swing.JButton jButton2;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JSlider jSlider1;
private javax.swing.JTextField jTextField1;
// End of variables declaration
```

Es lässt sich ablesen, dass für Schaltflächen die Klasse `JButton`, für Beschriftungen die Klasse `JLabel`, für den Slider ein `JSlider` und für einfache Textfelder die Klasse `JTextField` zum Einsatz kommen.

### Variablen umbenennen

Die Variablen sind standardmäßig privat und nichtssagend benannt. Wir wollen die Variablennamen ändern, sodass klarer wird, was welche Komponenten sind. Jetzt kommt

auf der linken Seite der NAVIGATOR/INSPECTOR ins Spiel. Er zeigt die hierarchische Struktur der Komponenten an. Mit der Taste **F2** lässt sich jeder Variablenname ändern. Das wollen wir machen.

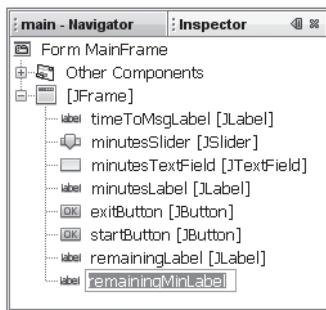


Abbildung 14.14: Umbenennung der Variablen im Komponentenbaum

In der Quellcodeansicht sind die Komponenten jetzt besser unterscheidbar:

```
// Variables declaration - do not modify
private javax.swing.JButton exitButton;
private javax.swing.JLabel minutesLabel;
private javax.swing.JSlider minutesSlider;
private javax.swing.JTextField minutesTextField;
private javax.swing.JLabel remainingLabel;
private javax.swing.JLabel remainingMinLabel;
private javax.swing.JButton startButton;
private javax.swing.JLabel timeToMsgLabel;
// End of variables declaration
```

14

Da wir im Konstruktor auf die Elemente Zugriff haben, wollen wir nach dem Aufruf von `initComponents()` die Schiebereglerposition auf 1 setzen und das Textfeld ebenfalls mit 1 vorbelegen:

```
/** Creates new form MainFrame */
public MainFrame() {
 initComponents();
 minutesSlider.setValue(1);
 minutesTextField.setText("1");
}
```

Die Methode `setValue()` erwartet einen numerischen Wert für den Slider, und `setText()` erwartet einen String für das Textfeld.

#### 14.2.6 Funktionalität geben

Nachdem die Variablen gut benannt sind, soll es an die Implementierung der Funktionalität gehen. Folgendes gilt es zu realisieren:

1. Das Aktivieren der Schaltfläche BEENDEN beendet das Programm.
2. Das Bewegen des Sliders aktualisiert das Textfeld.
3. Das Verändern des Textfeldes aktualisiert den SLIDER.
4. Nach dem Start läuft das Programm, und die verbleibende Zeit wird aktualisiert. Ist die Zeit um, erscheint eine Dialogbox.

Der erste Punkt ist am einfachsten: Beginnen wir dort.

#### Applikation beenden

Nötig für Interaktionen sind die sogenannten Listener, die auf Benutzerinteraktionen reagieren. Wenn etwa auf die Schaltfläche BEENDEN geklickt wird, muss es einen Listener geben, der das Klick-Ereignis mitbekommt und reagiert.

Für Schaltflächen gibt es eine einfache Möglichkeit, einen Listener hinzuzufügen: Wir doppelklicken im Designer schnell auf die Schaltfläche. Machen wir das für BEENDEN: NetBeans wechselt dann von der Design-Ansicht in den Quellcode und hat eine neue Methode hinzugefügt – in ihr setzen wir `System.exit(0);` ein, sodass sich Folgendes ergibt:

```
private void exitButtonActionPerformed(java.awt.event.ActionEvent evt) {
 System.exit(0);
}
```

Nur der Rumpf der Methode ist editierbar. Wer versehentlich einen Doppelklick gesetzt hat, kann ein Undo in der Design-Ansicht durchführen.

Ein Programmstart über `[F6]` zeigt schließlich, dass die Applikation mit einem Klick auf BEENDEN auch tatsächlich beendet wird.

## Sliderwert und Textfeld synchronisieren

Als Nächstes halten wir den Sliderwert und den Wert im Textfeld synchron. Zurück in der Design-Ansicht selektieren wir den SLIDER und finden im Kontextmenü den Menüpunkt EVENTS. Das sind alle Ereignisse, die der Slider auslösen kann. Wir interessieren uns für CHANGE.

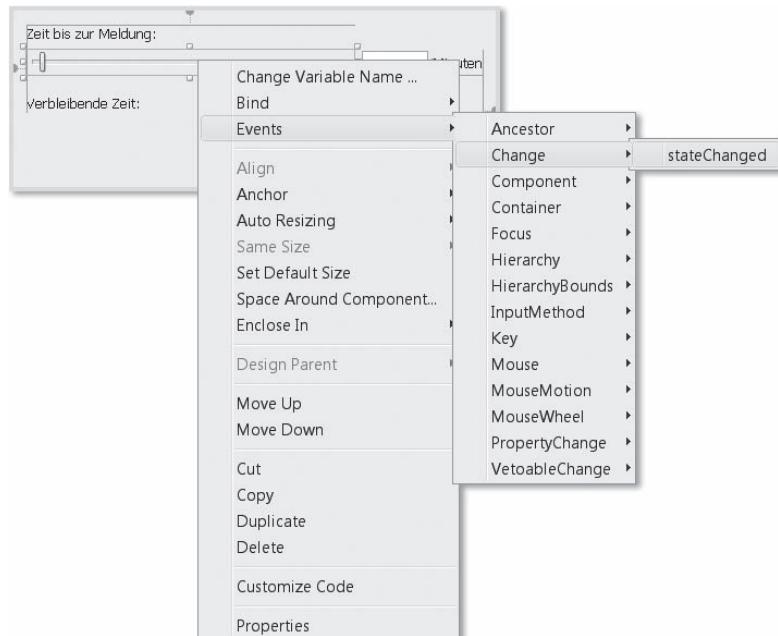


Abbildung 14.15: Im Formulardesigner einen Ereignisbehandler hinzufügen

Nach dem Aktivieren des Menüpunkts bekommen wir von NetBeans wieder Quellcode generiert. Die Listener-Methode wird immer dann aufgerufen, wenn der Slider vom Benutzer bewegt wird. Lesen wir einfach den aktuellen Wert aus, und schreiben wir ihn in das Textfeld:

```
private void minutesSliderStateChanged(javax.swing.event.ChangeEvent evt) {
 minutesTextField.setText("" + minutesSlider.getValue());
}
```

Die `JSlider`-Methode `getValue()` liefert also den aktuell eingestellten Wert, und `setText()` vom `JTextField` setzt einen String in die Textzeile.

Nach dem Start des Programms können wir den Slider bewegen, und im Textfeld steht die ausgewählte Zahl.

Jetzt der umgekehrte Fall: Wenn das Textfeld mit bestätigt wird, soll der Wert ausgelesen und damit die JSlider-Position gesetzt werden. Doppelklicken wir im Designer auf das Textfeld, dann wird wieder der passende Listener in den Quellcode eingefügt. Füllen wir ihn wie folgt:

```
private void minutesTextFieldActionPerformed(java.awt.event.ActionEvent evt) {
 try {
 minutesSlider.setValue(Integer.parseInt(minutesTextField.getText()));
 }
 catch (NumberFormatException e) { }
}
```

Da im Textfeld ja fälschlicherweise Nicht-Zahlen stehen können, fangen wir den Fehler ab, ignorieren ihn aber.

Nachdem jetzt Änderungen im Textfeld und Slider synchron gehalten werden, ist es an der Zeit, die Implementierung mit dem Start eines Timers abzuschließen.

### Timer starten

In der Ansicht DESIGNER doppelklicken wir auf die Schaltfläche START. Jetzt muss die aktuelle Wartezeit ausgelesen werden, nach deren Ende eine Dialogbox erscheint. Die konstante Abarbeitung übernimmt ein Swing-Timer (javax.swing.Timer), der alle 100 Millisekunden die Oberfläche aktualisiert und dann beendet wird, wenn die Wartezeit abgelaufen ist:

```
private void startButtonActionPerformed(java.awt.event.ActionEvent evt) {

 startButton.setEnabled(false);

 final long start = System.currentTimeMillis();
 final long end = start + minutesSlider.getValue() * 60 * 1000;

 final javax.swing.Timer timer = new javax.swing.Timer(100, null);
 timer.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
 long now = System.currentTimeMillis();
 if (now >= end)
 {

```

```

 remainingMinLabel.setText("");
 startButton.setEnabled(true);
 JOptionPane.showMessageDialog(null, "BING!");
 timer.stop();
 }
else
 remainingMinLabel.setText((end - now) / 1000 + " Sekunden");
}
});
timer.start();
}

```

Und das Programm ist fertig.

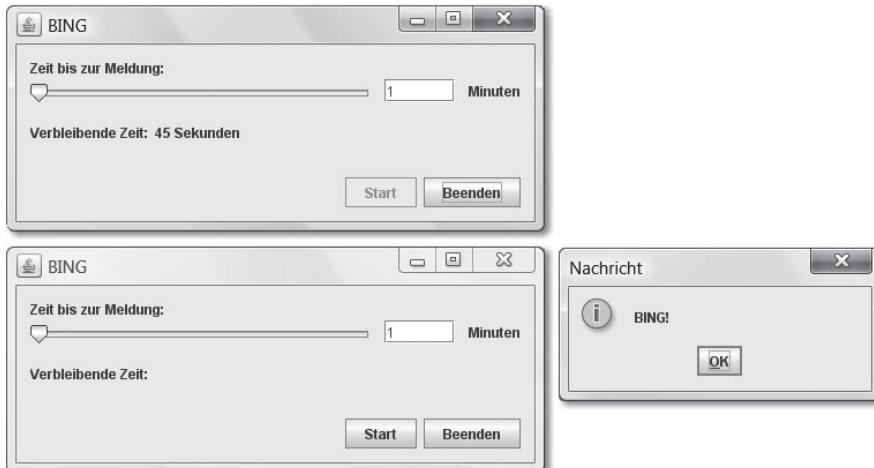


Abbildung 14.16: Die Anwendung in Aktion

### 14.3 Fenster zur Welt

Der Anfang aller GUI-Programme ist das Fenster (engl. *frame*), das einen sogenannten *Top-Level-Container* bildet. Wir müssen uns daher erst mit den Fenstern beschäftigen, bevor wir auf den Fensterinhalt näher eingehen können. Das Fenster dient auch als Grundlage von Dialogen: speziellen Fenstern, die entweder modal oder nicht modal arbeiten können. Wobei ein modaler Dialog erst bedient werden möchte, bis es mit dem Gesamtsystem weitergehen kann.

### 14.3.1 Swing-Fenster mit javax.swing.JFrame darstellen

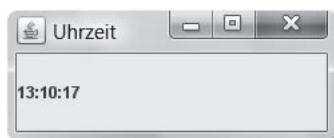
Um unter Swing ein Fenster zu öffnen, müssen wir die zentrale Klasse `JFrame` über das Paket `javax.swing` einbinden. Die allermeisten Swing-Komponenten befinden sich in diesem Paket, und nur ausgewählte komplexe Klassen wie Textkomponenten sind in Unterpaketen untergebracht. Viele Methoden der `JFrame`-Klasse stammen von den Oberklassen `java.awt.Frame` bzw. `java.awt.Window`.

**Listing 14.1:** com/tutego/insel/ui/swing/ClockApplication.java

```
package com.tutego.insel.ui.swing;

import java.util.Date;
import javax.swing.*;

public class ClockApplication
{
 public static void main(String[] args)
 {
 JFrame f = new JFrame("Uhrzeit");
 f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 f.setSize(250, 100);
 f.add(new JLabel(String.format("%tT", new Date())));
 f.setVisible(true);
 }
}
```



**Abbildung 14.17:** Swing-Fenster mit Datum

Aus dem Programm lassen sich unterschiedliche Elemente ablesen:

- Der parametrisierte Konstruktor von `JFrame` setzt automatisch einen Titel für das Fenster. Der Titel eines Fensters lässt sich aber auch später mit `setTitle()` wieder ändern. Der Standardkonstruktor lässt den Titel leer.

- Mit `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` setzen wir einen Zustand, sodass die Anwendung mit einem Klick auf das X sofort beendet wird. Das ist zum Testen praktisch, aber für echte GUI-Anwendungen natürlich keine Lösung.
- Die Methode `setSize()` setzt die Fenstergröße in Pixel.
- Abschließend zeigt `setVisible(true)` das Fenster an.

### Mit add() auf den Container

Das Programm erzeugt ein `JLabel`-Objekt und setzt es mit `add()` auf den `JFrame`. Der `JFrame` referenziert einen eigenen Kind-Container, der *Content-Pane* genannt wird und unser `JLabel` aufnimmt.

Vor Java 5 konnte nicht direkt mit `add()` gearbeitet werden, da der `JFrame` genau genommen nicht nur einen Container verwaltet, sondern viele, und wir mussten uns die Content-Pane mit `getContentPane()` erfragen und dann `add()` auf diesem Container-Objekt ausführen:

```
f.getContentPane().add(component);
```

Die beiden interessanten Konstruktoren sind:

```
class javax.swing.JFrame
extends Frame
implements WindowConstants, Accessible, RootPaneContainer
```

- `JFrame()`  
Erzeugt ein neues `JFrame`-Objekt, das am Anfang unsichtbar ist.
- `JFrame(String title)`  
Erzeugt ein neues `JFrame`-Objekt mit einem Fenster-Titel, das am Anfang unsichtbar ist.

Der Titel eines AWT- und Swing-Fensters lässt sich später mit `setTitle()` wieder ändern.

14

### 14.3.2 Fenster schließbar machen – `setDefaultCloseOperation()`

Die `JFrame`-Methode `setDefaultCloseOperation()` mit dem Argument `JFrame.EXIT_ON_CLOSE` beendet die Applikation über `System.exit()`, wenn der Benutzer über das  $\times$  in der Fensterleiste das Fenster schließt. Ohne die Anweisung verschwindet lediglich das Fenster in den Hintergrund: Es wird also geschlossen, die Applikation wird jedoch nicht beendet. Neben `EXIT_ON_CLOSE` gibt es weitere Konstanten. Mit `DO NOTHING ON CLOSE` bekom-

men wir das Standardverhalten eines AWT-Frames: Beim Schließen passiert nichts. Weder geht das Fenster zu, noch beendet die JVM das Programm.

```
class javax.swing.JFrame
extends Frame
implements WindowConstants, Accessible, RootPaneContainer
```

- void setDefaultCloseOperation(int operation)

Bestimmt, was passieren soll, wenn der Benutzer das Fenster schließt. Gültig sind die Konstanten `WindowConstants.DO NOTHING ON CLOSE`, `WindowConstants.HIDE ON CLOSE`, `WindowConstants.DISPOSE ON CLOSE`, `JFrame.EXIT ON CLOSE`. Eine weitere Erklärung findet sich bei der Ereignisbehandlung.

- int getDefaultCloseOperation()

Liefert die eingestellte Eigenschaft beim Schließen des Fensters.



#### Hinweis

Ein AWT-Fenster (also `java.awt.Frame`) kann nicht mit `×` in der Titelleiste geschlossen werden, da noch keine Ereignisbehandlung implementiert ist – der Frame bietet auch keine Methode `setDefaultCloseOperation()` an. Wir müssten selbst Fensterereignisse abfangen. Unter Swing horcht der `JFrame` selbstständig auf ein `WindowEvent`, reagiert in der `protected`-Methode `processWindowEvent()` auf das `WINDOW_CLOSING` und kann das Fenster nach Wunsch auch ohne hinzugefügten Ereignisbehandler schließen.

### 14.3.3 Sichtbarkeit des Fensters

Nach der Konstruktion ist das Fenster vorbereitet, aber erst der Aufruf von `setVisible(true)` macht es sichtbar. `setVisible()` stammt, wie auch weitere Methoden, die für `JFrame` und `Frame` interessant sind, von der Oberklasse `Window`.

```
class java.awt.Window
extends Container
implements Accessible
```

- void setVisible(boolean b)

Der Aufruf von `setVisible(true)` zeigt das Fenster an. Liegt es im Hintergrund, holt der Aufruf es wieder in den Vordergrund.

- `boolean isShowing()`  
Liefert true, wenn sich das Fenster auf dem Bildschirm befindet.
- `void toBack()`  
Reiht das Fenster als hinterstes in die Fensterreihenfolge ein. Ein anderes Fenster wird somit sichtbar.
- `void toFront()`  
Platziert das Fenster als vorderstes in der Darstellung aller Fenster auf dem Schirm.

**Hinweis**

In der Java-Steinzeit wurden die Methoden `show()` und `hide()` genutzt. Sie sind heute veraltet (*deprecated*).



## 14.4 Beschriftungen (JLabel)

Die erste Komponente, die wir kennenlernen wollen, ist das `javax.swing.JLabel`. Es repräsentiert eine Zeichenkette oder ein Icon, die bzw. das der Benutzer nicht editieren kann. Zum Einsatz kommt die Beschriftung zum Beispiel in einer Dialogbox.

Wie jede andere Komponente wird auch `JLabel` mit der `add()`-Methode auf den Bildschirm gebracht. Labels lösen keine eigenen Events aus. Da aber `JLabel` eine Unterklasse von `Component` und `JComponent` ist, reagiert es auf Ereignisse wie das Erzeugen und auch auf Maus-Operationen.

**Listing 14.2:** com/tutego/insel/ui/swing/JLabelDemo.java

```
package com.tutego.insel.ui.swing;

import java.awt.Color;
import javax.swing.*;

public class JLabelDemo
{
 public static void main(String[] args)
 {
 JFrame frame = new JFrame();
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 }
}
```

```

JLabel l = new JLabel("Lebe immer First-Class, sonst tun es deine Erben!");
l.setForeground(Color.BLUE);
frame.add(l);

frame.pack();
frame.setVisible(true);
}
}

```



Abbildung 14.18: Ein Swing-Label

Im Nachhinein lässt sich der Text mit `setText(String)` ändern. Der Text wird sofort angezeigt, da das `JLabel` (das Gleiche gilt auch für andere Komponenten) einen Auftrag zur Neuzeichnung vergibt, sodass kurze Zeit später der neue Text – inklusive nötiger Neuausrichtung durch Größenänderungen – erscheint. Mit `getText()` lässt sich der aktuelle Text auslesen.

- ```

class javax.swing.JLabel
extends JComponent
implements SwingConstants, Accessible

```

 - `JLabel()`
Erzeugt ein leeres Label mit links angeordnetem Text.
 - `JLabel(String text)`
Erzeugt ein Label mit gegebenem Text.
 - `void setText(String text)`
Ändert die Aufschrift des Labels im laufenden Betrieb.
 - `String getText()`
Liefert den Text des Labels.

Grafik und Beschriftung

Anders als das AWT-Label kann Swings `JLabel` ein Bild (Icon) anzeigen. Hinzu kommt, dass sich Icon und Text auch gemeinsam verwenden lassen. Über verschiedene Mög-

lichkeiten können horizontale und vertikale Positionen vom Text relativ zum Icon gesetzt werden. Auch die relative Position des Inhalts innerhalb der Komponente lässt sich spezifizieren. Die Voreinstellung für Labels ist eine zentrierte vertikale Darstellung im angezeigten Bereich. Enthalten die Labels nur Text, so ist dieser standardmäßig linksbündig angeordnet, und Bilder sind horizontal zentriert. Ist keine relative Position des Textes zum Bild angegeben, befindet sich der Text standardmäßig auf der rechten Seite des Bilds, und beide sind auf der Vertikalen angeordnet. Der Abstand von Bild und Text lässt sich beliebig ändern und ist im Standard-Look and Feel mit 4 Pixeln vordefiniert.

```
class javax.swing.JLabel  
extends JComponent  
implements SwingConstants, Accessible
```

- `JLabel(Icon icon)`
Erzeugt ein Label mit links angeordnetem Icon.
- `String getIcon()`
Liefert das Icon.
- `void setIcon(Icon icon)`
Ändert das Icon.

14

14.5 Es tut sich was – Ereignisse beim AWT

Beim Arbeiten mit grafischen Oberflächen interagiert der Benutzer mit Komponenten. Er bewegt die Maus im Fenster, klickt eine Schaltfläche an oder verschiebt einen Rollbalken. Das grafische System beobachtet die Aktionen des Benutzers und informiert die Applikation über die anfallenden Ereignisse. Dann kann das laufende Programm entsprechend reagieren.

14.5.1 Die Ereignisquellen und Horcher (Listener) von Swing

Im Ereignismodell von Java gibt es eine Reihe von *Ereignisauslösern* (*Ereignisquellen*, engl. *event sources*), wie zum Beispiel Schaltflächen oder Schieberegler. Die Ereignisse können vom Benutzer der grafischen Oberfläche kommen, etwa wenn er eine Schaltfläche anklickt, aber auch auf eigene Auslöser zurückzuführen sein, zum Beispiel wenn im Hintergrund eine Tabelle geladen wird.

Neben den Ereignisauslösern gibt es eine Reihe von Interessenten (*Listener* oder *Horcher* genannt), die gern informiert werden wollen, wenn ein Ereignis aufgetreten ist. Da der Interessent in der Regel nicht an allen ausgelösten Oberflächen-Ereignissen interessiert ist, sagt er einfach, welche Ereignisse er empfangen möchte. Dies funktioniert so, dass er sich bei einer Ereignisquelle anmeldet, und diese informiert ihn, wenn sie ein Ereignis aussendet. Auf diese Weise leidet die Systemeffizienz nicht, da nur diejenigen informiert werden, die auch Verwendung für das Ereignis haben. Für jedes Ereignis gibt es einen eigenen Listener, an den das Ereignis weitergeleitet wird – darum der Name für das Modell: *Delegation Model*.³ Die folgende Tabelle gibt eine Übersicht über einige Listener und zeigt, was für Ereignisse sie melden.

Listener	Ereignisse
ActionListener	Der Benutzer aktiviert eine Schaltfläche bzw. ein Menü oder drückt  auf einem Textfeld.
WindowListener	Der Benutzer schließt ein Fenster oder möchte es verkleinern.
MouseListener	Der Benutzer drückt auf eine Maustaste.
MouseMotionListener	Der Benutzer bewegt die Maus.

Tabelle 14.1: Einige Listener und die von ihnen gemeldeten Ereignisse

Dem Listener über gibt das Swing-Grafiksystem jeweils ein Ereignis-Objekt, etwa dem ActionListener ein ActionEvent-Objekt, dem WindowListener ein WindowEvent-Objekt usw. Die Einzigsten, die etwas aus der Reihe tanzen, sind MouseListener und MouseMotionListener, denn beide melden MouseEvent-Objekte.

Schritte zum richtigen Horchen

Damit die Quelle Ereignisse sendet und der Client darauf reagiert, sind zwei Schritte durchzuführen:

1. Implementieren des Listeners
2. Anmelden des Listeners

Um dieses Konzept kennenzulernen, wollen wir einen Fenster-Horcher programmieren, der registriert, ob ein Fenster geschlossen werden soll. Wenn ja, bietet er einen Dialog; bestätigt der Benutzer den Dialog, wird die Applikation beendet.

³ Die Entwickler hatten vorher den Namen »Command Model« vergeben, doch drückte dies die Arbeitsweise nicht richtig aus.

14.5.2 Listener implementieren

Der Listener selbst ist eine Schnittstelle, die von den Interessenten implementiert wird. Da die Ereignis-Schnittstelle Callback-Methoden vorschreibt, muss der Interessent diese Operation implementieren. Wird im nächsten Schritt ein Horcher mit dem Ereignisauslöser verbunden, kann die Ereignisquelle davon ausgehen, dass der Horcher die entsprechende Methode besitzt. Diese ruft die Ereignisquelle bei einem Ereignis später auf.

Die Schnittstelle WindowListener

Alle Meldungen rund um Fenster werden über einen WindowListener abgewickelt. Implementierungen können so etwa erfahren, ob das Fenster vergrößert oder verkleinert wurde. Die Schnittstelle deklariert folgende Operationen:

```
interface java.awt.event.WindowListener  
extends EventListener  
  
■ void windowOpened(WindowEvent e)  
    Wird aufgerufen, wenn das Fenster geöffnet wurde.  
■ void windowClosing(WindowEvent e)  
    Wird aufgerufen, wenn das Fenster geschlossen wird.  
■ void windowClosed(WindowEvent e)  
    Wird aufgerufen, wenn das Fenster mit dispose() geschlossen wurde.  
■ void windowIconified(WindowEvent e)  
    Wird aufgerufen, wenn das Fenster zum Icon verkleinert wird.  
■ void windowDeiconified(WindowEvent e)  
    Wird aufgerufen, wenn das Fenster wieder hochgeholt wird.  
■ void windowActivated(WindowEvent e)  
    Wird aufgerufen, wenn das Fenster aktiviert wird.  
■ void windowDeactivated(WindowEvent e)  
    Wird aufgerufen, wenn das Fenster deaktiviert wird.
```

Der nächste Schritt ist es, die Schnittstelle zu implementieren und in `windowClosing()` Programmcode einzubetten, der den Benutzer fragt, ob die Anwendung wirklich geschlossen werden soll.

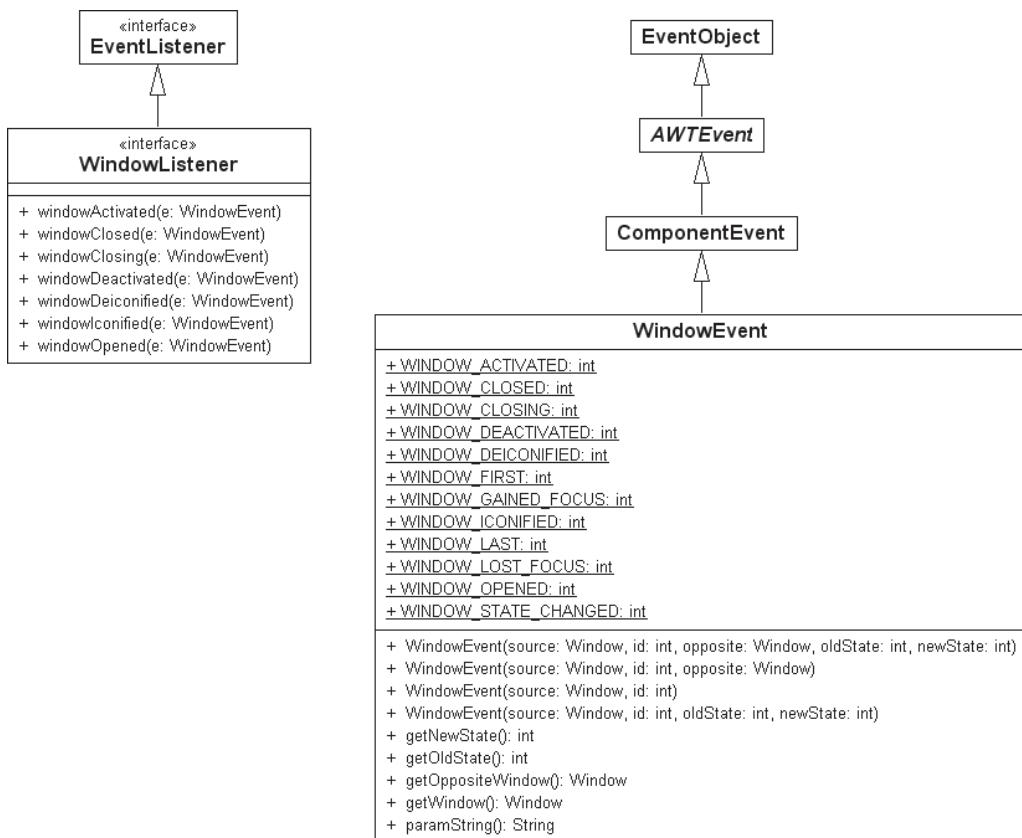


Abbildung 14.19: UML-Diagramm von WindowListener

Wer implementiert die Schnittstelle WindowListener?

Bei der Implementierung der Schnittstelle WindowListener gibt es unterschiedliche Varianten:

- Eine Klasse, die zum Beispiel JFrame erweitert, implementiert gleichzeitig WindowListener.
- Eine externe Klasse implementiert die Listener-Schnittstelle.
- Eine innere anonyme Klasse implementiert den Listener.

Die Lösungen haben Vor- und Nachteile. Wenn die eigene Klasse den Listener implementiert, ist das Programm schön kurz und der Listener hat einfachen Zugriff auf alle Zustände oder Variablen. Den Listener von einer anderen Klasse implementieren zu las-

sen hat dagegen den Vorteil, dass die Implementierung wiederverwendbar ist. Das ist bei einem allgemeinen Horcher für Fenterschließereignisse sicherlich angebracht.

Fenster-Schließ-Horcher

Die Implementierung fragt mit `JOptionPane.showConfirmDialog()`, ob das Fenster wirklich geschlossen werden soll. Die nicht genutzten Callback-Methoden implementieren wir leer.

Listing 14.3: com/tutego/insel/ui/event/DialogWindowClosingListener.java

```
package com.tutego.insel.ui.event;

import java.awt.event.*;
import javax.swing.JOptionPane;

public class DialogWindowClosingListener implements WindowListener
{
    @Override public void windowClosing( WindowEvent event )
    {
        int option = JOptionPane.showConfirmDialog( null, "Applikation beenden?" );
        if ( option == JOptionPane.OK_OPTION )
            System.exit( 0 );
    }

    @Override public void windowClosed( WindowEvent event ) { /*Empty*/ }
    @Override public void windowDeiconified( WindowEvent event ) { /*Empty*/ }
    @Override public void windowIconified( WindowEvent event ) { /*Empty*/ }
    @Override public void windowActivated( WindowEvent event ) { /*Empty*/ }
    @Override public void windowDeactivated( WindowEvent event ) { /*Empty*/ }
    @Override public void windowOpened( WindowEvent event ) { /*Empty*/ }
}
```

14

An diesem Beispiel ist abzulesen, dass jeder, der ein `WindowListener` sein möchte, die vorgeschriebenen Methoden implementieren muss. Damit zeigt er Interesse an dem `WindowEvent`. Bis auf `windowClosing()` haben wir die anderen Operationen nicht implementiert, da sie uns nicht interessieren. Die Implementierung ist so, dass die Anwendung beendet wird, wenn der Anwender auf das `x` klickt und den Dialog mit `OK` bestätigt.

14.5.3 Listener bei dem Ereignisauslöser anmelden/abmelden

Hat der Listener die Schnittstelle implementiert, wird er mit dem Ereignisauslöser verbunden. Dafür gibt es eine Reihe von Hinzufügen- und Entfernen-Methoden, die einer Namenskonvention folgen.

- `addEreignisListener(EreignisListener)`
- `removeEreignisListener(EreignisListener)`

Üblicherweise lassen sich beliebig viele Listener an einen Ereignisauslöser hängen.

addXXXListener() nur dann möglich, wenn es auch Ereignisse gibt

Nicht jede Komponente kann jedes Ereignis auslösen. Daher gibt es **nur addXXXListener()** für Ereignisse, die die Komponenten tatsächlich auslösen, und unterschiedliche Komponenten bieten unterschiedliche Hinzufügemethoden. Die `JFrame`-Klasse bietet zum Beispiel `addWindowListener()` für Fensterereignisse, aber kein `addActionListener()` – das wiederum hat bietet `JButton`, damit er die Aktivierung der Schaltfläche melden kann. Eine Schaltfläche löst eben keine Fenster-Ereignisse aus, und daher gibt es die Methode `addWindowListener()` bei Schaltflächen nicht. So lassen sich über die angebotenen `addXXXListener()`-Methoden gut die Ereignisse ablesen, die eine Komponente auslösen kann, denn das `XXX` wird dann nach der Namenskonvention der Ereignis-Typ sein.

Fensterschließer registrieren

Den Listener für Fenster-Ereignisse, unseren `DialogWindowClosingListener`, binden wir mit der Methode `addWindowListener()` an ein Fenster. Wird genau dieses Fenster geschlossen, bekommen wir das mit und können reagieren.

Listing 14.4: com/tutego/insel/ui/event/CloseWithDialogFrame.java

```
package com.tutego.insel.ui.event;

import javax.swing.*;

public class CloseWithDialogFrame
{
    public static void main( String[] args )
    {
        JFrame f = new JFrame();
        f.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );
```

```

f.add( new JLabel( "Zyklose bringen Regen" ) );
f.pack();
f.addWindowListener( new DialogWindowClosingListener() );
f.setVisible( true );
}
}

```

Wir tragen mit `addWindowListener()` den Listener ein. Immer wenn ein Window-Event ausgelöst wird, kümmert sich die jeweilige Methode um dessen Abarbeitung.

Normalerweise schließt Swing ein `JFrame` automatisch, wenn das \times gedrückt wird – anders als beim AWT-Frame. Die Anweisung `setDefaultCloseOperation(JFrame.DO NOTHING_ON_CLOSE)` verändert dieses Standardverhalten, damit nichts passiert und wir selbst zu 100 % die Kontrolle behalten. Denn ist die Zeile nicht im Programm, so merkt Swing, dass das \times gedrückt wurde, und versteckt das Fenster, egal was der Benutzer im Dialog angibt.

setDefaultCloseOperation() und der WindowListener *

Die ersten Swing-Programme, die wir geschrieben haben, nutzen `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`, damit Swing automatisch die Anwendung beendet, wenn der Anwender auf \times klickt. Diese Belegung ist für kleine Testprogramme in Ordnung, in ausgewachsenen GUI-Anwendungen sollte sich die Applikation jedoch nicht einfach schließen, sondern mit einem Dialog über das Ende informieren. Die Voreinstellung für die default-Close-Operation von `JFrame` ist `HIDE_ON_CLOSE`, was das Fenster automatisch verdeckt (das Fenster kann später wieder aktiviert werden), nachdem die `WindowListener` aufgerufen wurden. Eine weitere Konstante ist `DISPOSE_ON_CLOSE`, was ebenfalls die Listener abarbeitet, aber das Fenster automatisch schließt und alle Ressourcen freigibt. Übernehmen wir selbst die komplette Kontrolle, ist die Belegung mit `JFrame.DO NOTHING_ON_CLOSE` angebracht.

14

Der Unterschied zwischen windowClosing() und windowClosed() *

Die Schnittstelle `WindowListener` schreibt zwei Methoden vor, deren Namen sich ziemlich ähnlich anhören: `windowClosing()` und `windowClosed()`. Betrachten wir den Unterschied zwischen beiden und sehen wir uns an, wie ein Programm beide Methoden nutzen oder meiden kann.

In den einfachen Programmen setzen wir in die `windowClosing()`-Methode einen Aufruf von `System.exit()`, um die Applikation zu beenden, da `windowClosing()` immer bei Been-

digung der Applikation mit dem × am Fenster aufgerufen wird. Was allerdings leicht vergessen wird, ist die Tatsache, dass nicht nur der Benutzer über das × das Fenster schließen kann, sondern auch die Applikation über die spezielle Methode `dispose()`. Sie gibt alle Ressourcen frei und schließt das Fenster. Die Applikation ist dann allerdings noch nicht beendet. Damit wir das Schließen mit dem × und durch `dispose()` unterscheiden können, kümmert sich `windowClosing()` um das × und `windowClosed()` um das `dispose()`. Wenn wir lediglich mit dem × das Fenster schließen und die Applikation beendet werden soll, muss nicht noch extra `dispose()` schön brav die Ressourcen freigeben. Daher reicht oft ein `System.exit()`. Soll das Fenster jedoch mit × und `dispose()` einfach nur geschlossen werden oder ist eine gemeinsame Behandlung gewünscht, so ist es sinnvoll, in `windowClosing()` mit `dispose()` indirekt `windowClosed()` aufzurufen, etwa so:

```
public void windowClosing( WindowEvent e )
{
    event.getWindow().dispose();
}

public void windowClosed( WindowEvent e )
{
    // Das Fenster ist geschlossen, und jetzt können wir hier
    // weitermachen, etwa mit System.exit(), wenn alles
    // vorbei sein soll.
}
```

14.5.4 Adapterklassen nutzen

Hat eine Schnittstelle wie `WindowListener` viele Methoden, so hat eine Implementierung alle Methoden zu realisieren, auch wenn die Mehrzahl einen leeren Rumpf hat. Das ist lästig. Hier helfen *Adapterklassen* – Klassen, die die Schnittstellen mit leeren Rümpfen implementieren. Hat beispielsweise die Schnittstelle `WindowListener` sieben Methoden, so steht in der Adapterklasse folgende Implementierung:

Listing 14.5: `java.awt.event.WindowAdapter`

```
public abstract class WindowAdapter
    implements WindowListener, WindowStateListener, WindowFocusListener
{
    public void windowOpened( WindowEvent e ) { }

    public void windowClosing( WindowEvent e ) { }
```

```

public void windowClosed( WindowEvent e ) { }
public void windowIconified( WindowEvent e ) { }
public void windowDeiconified( WindowEvent e ) { }
public void windowActivated( WindowEvent e ) { }
public void windowDeactivated( WindowEvent e ) { }
public void windowStateChanged( WindowEvent e ) { }
public void windowGainedFocus( WindowEvent e ) { }
public void windowLostFocus( WindowEvent e ) { }
}

```

Zusätzlich entdecken wir einige Methoden, die nicht direkt von unserem `WindowListener` stammen, sondern von zwei weiteren Schnittstellen, die jetzt keine Rolle spielen.

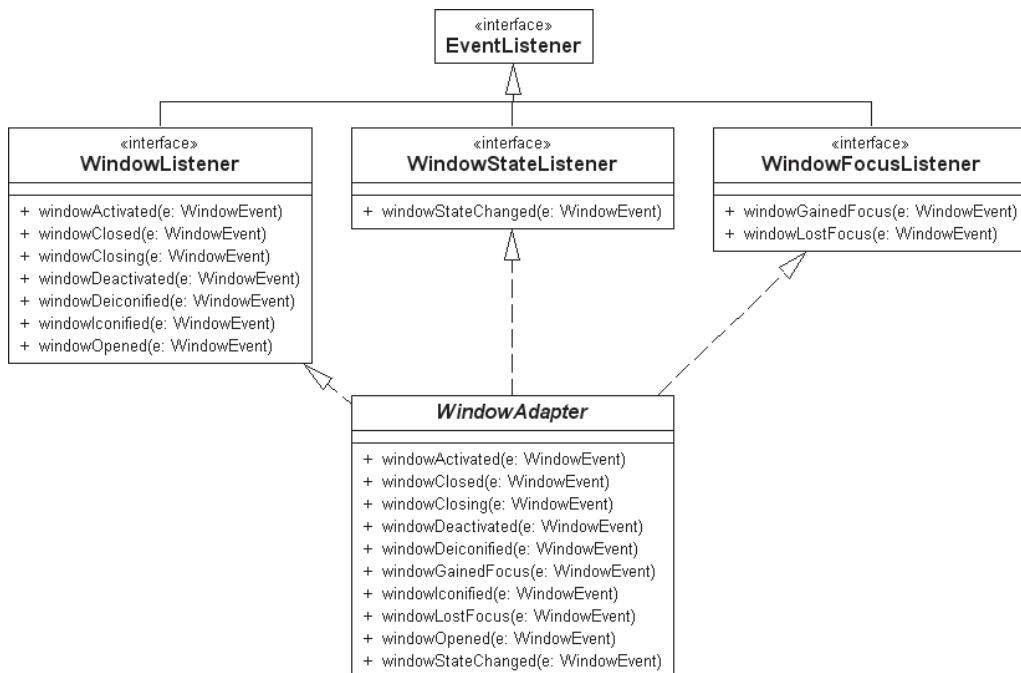


Abbildung 14.20: UML-Diagramm von `WindowAdapter`

Wenn wir jetzt einen Ereignisbehandler implementieren müssen, erweitern wir einfach die Adapterklasse und überschreiben nur die relevanten Methoden. Der Listener zum Schließen des Fensters mit einer Adapterklasse sieht dann wie folgt aus:

Listing 14.6: com/tutego/insel/ui/event/DialogWindowClosingListener2.java

```
package com.tutego.insel.ui.event;

import java.awt.event.*;
import javax.swing.JOptionPane;

public class DialogWindowClosingListener2 extends WindowAdapter
{
    @Override public void windowClosing( WindowEvent event )
    {
        int option = JOptionPane.showConfirmDialog( null, "Applikation beenden?" );
        if ( option == JOptionPane.OK_OPTION )
            System.exit( 0 );
    }
}
```

Beim Client ändert sich nichts viel, es ist immer noch die addWindowListener()-Methode:

Listing 14.7: com/tutego/insel/ui/event/CloseWithDialogFrame.java, main()

```
JFrame f = new JFrame();
f.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );
f.add( new JLabel( "Zyklone bringen Regen" ) );
f.pack();
f.addWindowListener( new DialogWindowClosingListener2() );
f.setVisible( true );
```

14.5.5 Innere Mitgliedsklassen und innere anonyme Klassen

Wir haben für das Fensterschließ-Beispiel eine externe Klasse benutzt, weil die Klasse für unterschiedliche Swing-Programme nützlich ist. Ist der Listener sehr individuell und mit der Komponente verbunden, so sind innere anonyme Klassen nützlich. Dazu zwei Beispiele.

Auf Klicks reagieren

Das JLabel kann Maus-Ereignisse empfangen, was wir nutzen wollen, um bei einem Doppelklick die Applikation zu beenden:

Listing 14.8: com/tutego/insel/ui/event/ClickOnJLabelToClose.java, main()

```
JFrame frame = new JFrame();
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

JLabel label = new JLabel( "Lebe immer First-Class, sonst tun es deine Erben!" );
label.setForeground( Color.BLUE );

frame.add( label );

label.addMouseListener( new MouseAdapter() {
    @Override public void mouseClicked( MouseEvent e ) {
        if ( e.getClickCount() > 1 )
            System.exit( 0 );
    }
} );

frame.pack();
frame.setVisible( true );
```

Die Lösung hat den Vorteil, dass nicht extra eine eigene Klasse mit einem häufig überflüssigen Namen angelegt wird. Die Unterklasse von `MouseAdapter` ist nur hier sinnvoll und wird nur in diesem Kontext benötigt.

14

Mini-Multiplikationsrechner

Innere Klassen sind elegant, denn sie können leicht auf Zustände der äußeren Klasse zugreifen. Das folgende Programm realisiert einen Hexadezimalwandler.

Listing 14.9: com/tutego/insel/ui/event/Hexconverter.java

```
package com.tutego.insel.ui.event;

import java.awt.FlowLayout;
import java.awt.event.*;
import javax.swing.*;

public class Hexconverter
{
```

```

public static void main( String[] args )
{
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setLayout( new FlowLayout() );

    final JTextField decTextField = new JTextField();
    decTextField.setColumns( 5 );
    frame.add( decTextField );
    frame.add( new JLabel(" ist hexadezimal " ) );
    final JTextField hexTextField = new JTextField();
    hexTextField.setColumns( 5 );
    frame.add( hexTextField );
    JButton okButton = new JButton( "Konvertiere" );
    frame.add( okButton );

    okButton.addActionListener( new ActionListener() {
        @Override public void actionPerformed( ActionEvent e ) {
            int dec = Integer.parseInt( decTextField.getText() );
            String hex = Integer.toHexString( dec );
            hexTextField.setText( hex );
        }
    } );
}

frame.pack();
frame.setVisible( true );
}
}

```

Das Programm nutzt einige Dinge, die neu sind, aber gut verständlich: Damit mehrere Komponenten nebeneinander gesetzt werden, konfigurieren wir das Fenster mit einem FlowLayout. Dann platzieren wir ein Textfeld, eine Beschriftung, ein Textfeld und eine Schaltfläche nebeneinander. An die Schaltfläche kommt ein Listener, damit wir mitbekommen, ob sie bestätigt wurde. Wenn sie bestätigt wurde, erfolgt die Konvertierung (Fehler werden nicht abgefangen). Die anonyme Klasse kann dabei sehr gut auf die beiden Textfelder zurückgreifen. Wäre der Listener eine externe Klasse, so müssten ihr – etwa im Konstruktor – die beiden Textfelder mitgegeben werden.

14.5.6 Aufrufen der Listener im AWT-Event-Thread

Nachdem der Listener implementiert und angemeldet wurde, ist das System im Fall eines aufkommenden Ereignisses bereit, es zu verteilen. Aktiviert zum Beispiel der Benutzer eine Schaltfläche, so führt der *AWT-Event-Thread* – auch *Event-Dispatching-Thread* genannt – den Programmcode im Listener selbstständig aus. Sehr wichtig ist Folgendes: Der Programmcode im Listener sollte nicht zu lange laufen, da sich sonst Ereignisse in der Queue sammeln, die der AWT-Thread nicht mehr verarbeiten kann. Diese Eigenschaft fällt dann schnell auf, wenn sich Aufforderungen zum Neuzeigen (Repaint-Ereignisse) aufstauen, da auf diese Weise leicht ein »stehendes System« entsteht.

Die Reihenfolge, in der die Listener abgearbeitet werden, ist im Prinzip undefiniert. Zwar reiht das JDK sie in eine Liste ein, sodass es dadurch eine Reihenfolge gibt, doch sollte diesem Implementierungsdetail keine Beachtung geschenkt werden.

14.6 Schaltflächen

14

Eine *Schaltfläche* (engl. *button*) ermöglicht es dem Anwender, eine Aktion auszulösen. Schaltflächen sind meistens beschriftet und stellen eine Zeichenkette dar oder tragen eine Grafik, etwa im Fall eines Symbols in der Symbolleiste. Unter dem AWT kann eine Schaltfläche nur Text, aber keine Icons darstellen.

Swing kennt unterschiedliche Schaltflächen. Dazu zählen `JButton` für einfache Schaltflächen, aber auch Schaltflächen zum Ankreuzen.

14.6.1 Normale Schaltflächen (`JButton`)

Eine *Schaltfläche* (engl. *button*) ermöglicht es dem Anwender, eine Aktion auszulösen. Schaltflächen sind meistens beschriftet und stellen eine Zeichenkette dar oder tragen eine Grafik, etwa im Fall eines Symbols in der Symbolleiste. Unter dem AWT kann eine Schaltfläche nur Text, aber keine Icons darstellen. Die Schaltfläche `JButton` reagiert auf Aktivierung und erzeugt ein `ActionEvent`, das ein angehänger `ActionListener` meldet.



Abbildung 14.21: JButton mit einer einfachen Schaltfläche zum Schließen

Listing 14.10: com/tutego/insel/ui/swing/JButtonDemo.java, main()

```
JFrame frame = new JFrame();
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
frame.setLayout( new FlowLayout() );

// Button 1

final Icon icon1 = new ImageIcon(
    JButtonDemo.class.getResource( "/images/user-trash-full.png" ) );
final Icon icon2 = new ImageIcon(
    JButtonDemo.class.getResource( "/images/user-trash.png" ) );

final JButton button1 = new JButton( icon1 );
frame.add( button1 );

ActionListener al = new ActionListener() {
    @Override public void actionPerformed( ActionEvent e ) {
        button1.setIcon( icon2 );
    }
};

button1.addActionListener( al );

// Button 2

final JButton button2 = new JButton( "Ende" );
frame.add( button2 );

button2.addActionListener( new ActionListener() {
    @Override public void actionPerformed( ActionEvent e ) {
        System.exit( 0 );
    }
} );

frame.pack();
frame.setVisible( true );
```

Die JButton-API

Es gibt mehrere Konstruktoren für JButton-Objekte. Die parameterlose Variante erzeugt eine Schaltfläche ohne Text. Der Text lässt sich mit `setText()` nachträglich ändern. In der Regel nutzen wir den Konstruktor, dem ein String mitgegeben wird.

```
class javax.swing.JButton  
extends AbstractButton  
implements Accessible
```

- `JButton()`
Erzeugt eine neue Schaltfläche ohne Aufschrift.
- `JButton(String text)`
Erzeugt eine neue Schaltfläche mit Aufschrift.
- `JButton(Icon icon)`
Erzeugt eine neue Schaltfläche mit Icon.
- `JButton(String text, Icon icon)`
Erzeugt eine neue Schaltfläche mit Aufschrift und Icon.
- `void setText(String text)`
Ändert die Aufschrift der Schaltfläche auch im laufenden Betrieb.⁴
- `String getText()`
Liefert die Aufschrift der Schaltfläche.
- `void addActionListener(ActionListener l)`
Fügt dem Button einen ActionListener hinzu, der die Ereignisse abgreift, die durch die Schaltfläche ausgelöst werden.
- `void removeActionListener(ActionListener l)`
Entfernt den ActionListener wieder. Somit kann er keine weiteren Ereignisse mehr abgreifen.

⁴ Vergleichen wir das mit den Methoden der Klasse `java.awt.Label`. Hier heißen die Methoden zum Lesen und Ändern des Textes `setLabel()` und `getLabel()`. Dies ist für mich wieder eines der AWT-Rätsel. Warum heißt es bei einem `java.awt.Label`-Objekt `setText()/getText()` und bei einem `java.awt.Button`-Objekt `setLabel()/getLabel()`? Immerhin heißt es bei den Swing-Komponenten `JLabel` und `JButton` konsequent `setText()/getText()`.

**Hinweis**

Wörter mit einer starken emotionalen Bindung sollten vermieden werden. In englischen Programmen müssen Wörter wie »kill« oder »abort« umgangen werden.⁵

14.6.2 Der aufmerksame ActionListener

Klicken wir auf die Schaltfläche, so sollte die Aktion gemeldet werden. Diese wird in Form eines ActionEvent-Objekts an den Zuhörer (einen ActionListener) gesendet. Ein ActionListener wird mit der Methode add ActionListener() an die Objekte angeheftet, die Aktionen auslösen können. ActionListener ist eine Schnittstelle mit der Methode actionPerformed(). Die Schnittstelle ActionListener wiederum erweitert die Schnittstelle EventListener, die von allen Listener-Interfaces implementiert werden muss.

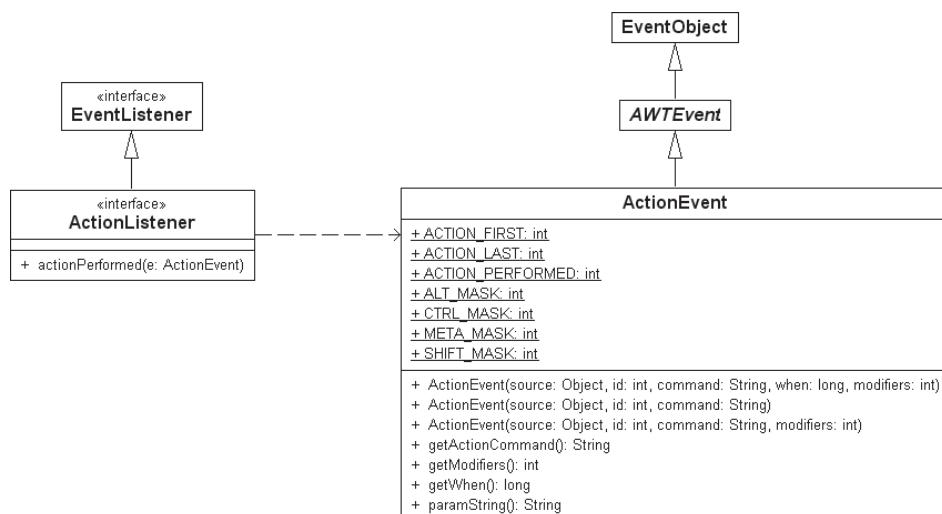


Abbildung 14.22: Die Beziehung zwischen ActionListener und ActionEvent

```

interface java.awt.event.ActionListener
extends EventListener
  
```

- void actionPerformed(ActionEvent e)
- Wird aufgerufen, wenn eine Aktion ausgelöst wird.

⁵ Siehe dazu das Buch »Tog on Interface« von Bruce Tognazzini, auszugsweise unter <http://www.asktog.com/TOI/toi06KeyboardVMouse1.html>.

Werfen wir noch einmal einen Blick auf die Implementierung unserer beiden ActionListener-Klassen, die als anonyme innere Klasse realisiert sind:

```
ActionListener al = new ActionListener() {  
    public void actionPerformed( ActionEvent e ) {  
        button1.setIcon( icon2 );  
    }  
};  
  
button2.addActionListener( new ActionListener() {  
    public void actionPerformed( ActionEvent e ) {  
        System.exit( 0 );  
    }  
} );
```

14.7 Alles Auslegungssache: die Layoutmanager

14

Alle Komponenten müssen auf einen Container platziert werden. Container sind besondere Swing-Elemente, die dazu dienen, andere Kinder aufzunehmen und zu verwalten. Ein Container ist zum Beispiel JPanel. Er ist im Wesentlichen eine JComponent mit der Möglichkeit, Kinder nach einem bestimmten Layoutverfahren anzurufen.

Ein Layoutmanager ist dafür verantwortlich, Elemente eines Containers nach einem bestimmten Verfahren anzurufen, zum Beispiel zentriert oder von links nach rechts. Ein Container fragt bei einer Neudarstellung immer seinen Layoutmanager, wie er seine Kinder anordnen soll. Jeder Layoutmanager implementiert eine unterschiedliche Strategie zur Anordnung.

14.7.1 Übersicht über Layoutmanager

Java bietet bisher folgende Layoutmanager:

- **FlowLayout:** Ordnet Komponenten von links nach rechts an.
- **BoxLayout:** Ordnet Komponenten horizontal oder vertikal an.
- **GridLayout:** Setzt Komponenten in ein Raster, wobei jedes Element die gleichen Maße besitzt.

- BorderLayout: Setzt Komponenten in vier Himmelsrichtungen oder in der Mitte.
- GridBagLayout: Sehr flexibler Manager als Erweiterung von GridLayout.⁶
- CardLayout: Verwaltet Komponenten wie auf einem Stapel, sodass nur eine Komponente sichtbar ist.
- SpringLayout: Berücksichtigt Abhängigkeiten der Kanten von Komponenten.⁷
- GroupLayout: Für GUI-Builder im Allgemeinen die beste Wahl.

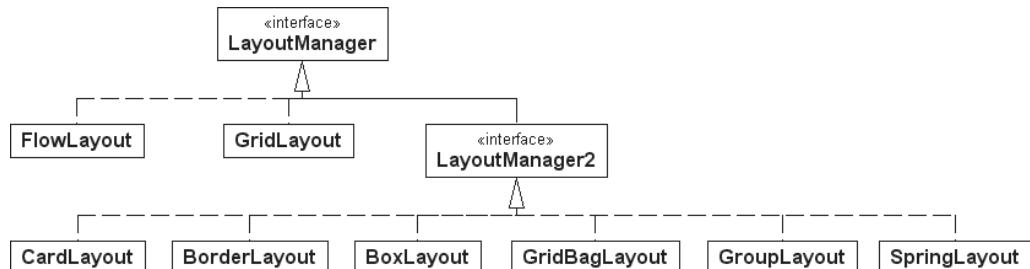


Abbildung 14.23: Typbeziehungen zwischen den Layout-Managern

14.7.2 Zuweisen eines Layoutmanagers

Die Methode `setLayout(LayoutManager)` weist einem Container eine Ausrichtungsstrategie zu.

```

class java.awt.Container
extends Component
  
```

- `void setLayout(LayoutManager mgr)`
Setzt einen neuen Layoutmanager für den Container.
- `LayoutManager getLayout()`
Liefert den aktuellen Layoutmanager.

LayoutManager ist eine Schnittstelle, die von unterschiedlichsten konkreten Layoutmanagern implementiert wird. Die zentrale Operation ist `layoutContainer()`, die dafür verantwortlich ist, die absoluten Positionen via `setBounds()` zu setzen.

6 Packer (<https://packer.dev.java.net/>) ist eine alternative API für die GridBagLayout-Funktionalität.

7 SpringLayout wird selten verwendet – wenn, dann von GUI-Buildern. Mehr Informationen dazu gibt die Webseite <http://java.sun.com/docs/books/tutorial/uiswing/layout/spring.html>.

JPanel mit einem Layoutmanager verbinden

Erinnern wir uns, dass die Klasse JPanel ein Container ist und daher auch ein eigenes Layout besitzen kann. Praktisch ist der Konstruktor, der gleich einen Layoutmanager annimmt (das JPanel ist bisher die einzige Klasse, der ein Layoutmanager gleich im Konstruktor übergeben werden kann).

```
class javax.swing.JPanel  
extends JPanel  
implements Accessible
```

- JPanel(LayoutManager layout)
Erzeugt ein JPanel mit Doppelpufferung und dem angegebenen Layoutmanager.
- JPanel(LayoutManager layout, boolean isDoubleBuffered)
Erzeugt ein neues JPanel mit dem angegebenen Layoutmanager und der Puffer-Strategie.

Tipp

Fitts's Law beschreibt die Zeit, die benötigt wird, um von einem Anfangspunkt zu einem Endpunkt zu kommen. Diese Zeit ist abhängig vom Logarithmus der Strecke zwischen dem Start- und dem Endpunkt und der Größe des Ziels. Daher gilt: Platziere die Elemente einer Oberfläche so, dass sie leicht zu erreichen sind. Je weiter das Ziel entfernt und je kleiner der Button ist, desto länger dauert die Operation.

14

14.7.3 Im Fluss mit FlowLayout

Der FlowLayout-Manager setzt seine Elemente von links nach rechts in eine Zeile. Die Komponenten behalten ihre Größe, das heißt, der Layoutmanager gibt keine neue Größe vor. Passen nicht alle Elemente in eine Zeile, so werden sie untereinander angeordnet. Ein zusätzlicher Parameter bestimmt, wie die Elemente im Container positioniert werden: zentriert, rechts- oder linksbündig. Ohne Einstellung ist die Anzeige zentriert. Standardmäßig besitzt jedes neue JPanel-Objekt ein FlowLayout als Layoutmanager.



Listing 14.11: com/tutego/insel/ui/layout/FlowLayoutDemo.java, main()

```
JFrame f = new JFrame();
f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
f.setLayout( new FlowLayout() );

JComboBox<String> choice = new JComboBox<String>();
choice.addItem( "Mike: Mein Gott Walter" );
choice.addItem( "Sweet: Co Co" );

f.add( choice );
f.add( new JButton(
    new ImageIcon(FlowLayoutDemo.class.getResource("/images/play.png") ) ) );

f.pack();
f.setVisible( true );
```



Abbildung 14.24: FlowLayout mit Mike

Den Elementen kann zusätzlich mehr Freiraum (engl. *gap*) gegeben werden. Voreingestellt sind 5 Pixel. Die Ausrichtung (engl. *alignment*), die beim Umbruch angegeben werden kann, ist eine ganzzahlige Konstante aus `FlowLayout`. Es stehen drei Klassen-Konstanten zur Verfügung: `FlowLayout.LEFT`, `FlowLayout.CENTER` und `FlowLayout.RIGHT`.

```
class java.awt.FlowLayout  
    implements LayoutManager, Serializable
```

- `FlowLayout()`
Erzeugt ein Flow-Layout mit 5 Pixeln horizontalem und vertikalem Freiraum.
- `FlowLayout(int align)`
Erzeugt ein Flow-Layout mit 5 Pixeln Freiraum und der angegebenen Ausrichtung.
- `FlowLayout(int align, int hgap, int vgap)`
Erzeugt ein Flow-Layout mit der angegebenen Ausrichtung und einem horizontalen beziehungsweise vertikalen Freiraum.
- `int getAlignment()`
Liefert das Alignment des Layoutmanagers. Möglich sind `FlowLayout.LEFT`, `FlowLayout.RIGHT` oder `FlowLayout.CENTER`.
- `void setAlignment(int align)`
Setzt das Alignment mithilfe der Konstanten `FlowLayout.LEFT`, `FlowLayout.RIGHT` oder `FlowLayout.CENTER`.
- `int getHgap()`
- `int getVgap()`
Liefert den horizontalen/vertikalen Abstand der Komponenten.
- `void setHgap(int hgap)`
- `void setVgap(int vgap)`
Setzt den horizontalen/vertikalen Abstand zwischen den Komponenten.

14.7.4 BoxLayout

BoxLayout ist vergleichbar mit `FlowLayout`, nur ordnet dieses entlang der x- oder y-Achse an und umbricht nicht. Das Layoutmanagement ist etwas seltsam, da `setLayout()` nicht allein genügt, um den Layoutmanager zuzuweisen. Vielmehr bekommt ein Exemplar von `BoxLayout` zusätzlich eine Referenz auf den Container.

zB Beispiel

Erzeuge ein JPanel, und füge zwei untereinander angeordnete JButton-Objekte hinzu:

```
JPanel p = new JPanel();
p.setLayout( new BoxLayout(p, BoxLayout.Y_AXIS) );
p.add( new JButton("<") );
p.add( new JButton(">") );
```

Swing bringt für das BoxLayout noch eine Abkürzung mit: Die Klasse heißt `javax.swing.Box` und verhält sich wie ein Container. Dem Box-Objekt ist automatisch der Layoutmanager `BoxLayout` zugewiesen.

zB Beispiel

Füge in eine Box eine Schaltfläche und ein Textfeld ein:

```
Box box = new Box( BoxLayout.Y_AXIS );
box.add( new JButton("Knopf") );
box.add( new JTextField() );
```

14.7.5 Mit BorderLayout in alle Himmelsrichtungen

Ein BorderLayout unterteilt seine Zeichenfläche in fünf Bereiche: Norden, Osten, Süden, Westen und Mitte (»Center«). Die Elemente im Norden und Süden erstrecken sich immer über die gesamte Länge des Containers. Die Höhe des Nordens und Südens ergibt sich aus der Wunschhöhe der Kinder, und die Breite wird angepasst. Die Elemente rechts und links bekommen ihre gewünschte Breite, werden aber in der Höhe gestreckt. Das Element in der Mitte wird in Höhe und Breite angepasst.

Für jeden dieser Bereiche (Richtungen) sieht die Klasse BorderLayout eine Konstante vor: `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.EAST`, `BorderLayout.SOUTH` und `BorderLayout.WEST`. Dem Container fügen wir mit der Methode `add(Komponente, Richtung)` eine Komponente hinzu, wobei das zweite Argument die Angabe der Himmelsrichtung ist. Diese Angabe ist jedoch ungünstig für bidirektionale Anwendungen wie Arabisch oder Hebräisch, da eine Komponente, die für uns links liegt, dort rechts liegen soll. Eine Angabe wie `BorderLayout.WEST` ist aber statisch. Seit Java 1.4 bietet BorderLayout die Konstanten `LINE_START` und `LINE_END`, was im Fall der Links-nach-rechts-Anordnung

bedeutet: LINE_START ist WEST und LINE_END ist EAST. Eigentlich gibt es auch PAGE_START und PAGE_END, die jedoch nicht beachtet werden, da Java bisher keine Verdrehung der Nord-Südachse unterstützt. Also ist PAGE_START immer oben und PAGE_END immer unten (also so wir das verstehen).

Beispiel

zB

Setze die Schaltfläche button in den Westen:

```
container.add( button, BorderLayout.LINE_START );
```

Wird die Methode add() mit nur einem Argument aufgerufen, so wird die Komponente automatisch in die Mitte (Center) gesetzt:

Listing 14.12: com/tutego/insel/ui/layout/BorderLayoutDemo.java, main()

```
JFrame f = new JFrame();
f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
// applyComponentOrientation( ComponentOrientation.RIGHT_TO_LEFT );

f.setLayout( new BorderLayout(5, 5) );

f.add( new JButton("Nie"), BorderLayout.PAGE_START );
f.add( new JButton("ohne"), BorderLayout.LINE_END );
f.add( new JButton("Seife"), BorderLayout.PAGE_END );
f.add( new JButton("waschen"), BorderLayout.LINE_START );
f.add( new JButton("Center") );

f.setSize( 400, 150 );
f.setVisible( true );
```

14



Abbildung 14.25: Der Layoutmanager BorderLayout



Hinweis

Beim AWT gilt, dass der Container `java.awt.Frame` automatisch mit einem `BorderLayout` verbunden ist. Das gilt bei `JFrame` beziehungsweise seinen Content-Panes ebenso. Allerdings verwendet die `JRootPane`, das heißt der Container für die Content-Panes, den internen Manager `RootLayout`, der nicht mit `BorderLayout` verwandt ist.

Wer das Vertauschen der Seiten ausprobieren möchte, der muss nur den Kommentar aus der Zeile mit der Methode `applyComponentOrientation()` herausnehmen.

```
class java.awt.BorderLayout
    implements LayoutManager, Serializable
```

- `BorderLayout()`
Erzeugt ein neues `BorderLayout`, wobei die Komponenten ohne Abstand aneinanderliegen.
- `BorderLayout(int hgap, int vgap)`
Erzeugt ein `BorderLayout`, wobei zwischen den Komponenten ein Freiraum eingefügt wird. `hgap` spezifiziert den Freiraum in der Horizontalen und `vgap` den in der Vertikalen. Die Freiräume werden in Pixeln gemessen.
- `int getHgap()`
- `int getVgap()`
Gibt den horizontalen/vertikalen Raum zwischen den Komponenten zurück.
- `void setHgap(int hgap)`
- `void setVgap(int vgap)`
Setzt den horizontalen/vertikalen Zwischenraum.



Hinweis

Anstelle von `add(Komponente, BorderLayout.Orientierung)` lässt sich eine Komponente auch mit der Variante `add(Orientierungszeichenkette, Komponente)` hinzufügen. Diese Angabe ist jedoch veraltet und sollte nicht mehr verwendet werden.

```
class java.awt.Container
    extends Component
```

- `void add(Component comp, Object constraints)`
Fügt die Komponente in den Container ein. Die Variable `constraints` wird im Fall von `BorderLayout` etwa mit den Konstanten `PAGE_START`, `LINE_END`, `PAGE_END`, `LINE_START` oder `CENTER` belegt.



Hinweis

Ein einfaches `add(comp)` auf einem Container mit `BorderLayout` hat den gleichen Effekt wie `add(comp, BorderLayout.CENTER)`. Werden mehrmals hintereinander Komponenten einfach mit `add(comp)` dem Container hinzugefügt, so werden sie alle im Zentrum übereinander gestapelt, sodass nur noch die letzte hinzugefügte Komponente sichtbar ist.

14.7.6 Rasteranordnung mit GridLayout

Das `GridLayout` ordnet seine Komponenten in Zellen an, wobei die Zeichenfläche rechteckig ist. Jeder Komponente in der Zelle wird dieselbe Größe zugeordnet, also bei drei Elementen in der Breite ein Drittel des Containers. Wird der Container vergrößert, so werden die Elemente gleichmäßig vergrößert. Sie bekommen so viel Platz wie möglich.

Listing 14.13: com/tutego/insel/ui/layout/GridLayoutDemo.java, main()

```
JFrame f = new JFrame();
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

f.setLayout( new GridLayout(/*3*/ 0, 2, 6, 3) );

f.add( new JLabel(" Wie heißt du denn, mein Kleiner?") );
f.add( new JTextField() );
f.add( new JLabel(" Na, wie alt bist du denn?") );
f.add( new JTextField(NumberFormat.getIntegerInstance()) );
f.add( new JLabel(" Dann mal das Passwort eingeben:") );
f.add( new JPasswordField() );

f.pack();
f.setVisible( true );
```

14

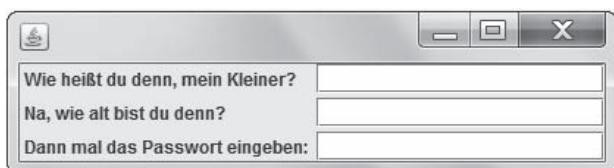


Abbildung 14.26: Beispiel für GridLayout

API-Dokumentation

```
class java.awt.GridLayout
    implements LayoutManager, Serializable
```

- `GridLayout()`
Erzeugt ein GridLayout mit einer Spalte pro Komponente in einer Zeile.
- `GridLayout(int rows, int cols)`
Erzeugt ein GridLayout mit rows Zeilen oder cols Spalten. Die zu berechnende Anzahl sollte auf 0 gesetzt werden.
- `GridLayout(int rows, int cols, int hgap, int vgap)`
Erzeugt ein GridLayout mit rows Zeilen oder cols Spalten. Horizontale Freiräume werden an die rechten und linken Ecken jeder Zeile sowie zwischen den Spalten gesetzt. Vertikale Freiräume werden an die unteren und oberen Ecken gesetzt, zudem zwischen den Reihen.

Beim Konstruktor, der Zeilen oder Spalten angibt, reicht es, lediglich die Anzahl der Elemente in der Zeile oder Spalte anzugeben; der Layoutmanager nutzt ohnehin nur eine Angabe und berechnet daraus die verbleibende Anzahl. Ein mit Zeilen oder Spalten parametrisierter Konstruktor erlaubt es – so wie beim BorderLayout –, Zwischenraum einzufügen.

zB Beispiel

Setze ein Layout mit drei Zeilen:

```
container.setLayout( new GridLayout(3, 0xcafebabe) );
```

Bei nur vier Elementen können wir auf diese Anzahl von fiktiven Spalten gar nicht kommen. Bei gegebener Zeilenanzahl wird sie nicht genutzt.

GridLayout berechnet die Anzahl der passenden Spalten für die Anzahl der Komponenten. Das zeigt die Implementierung in den Methoden `preferredLayoutSize()`, `minimumLayoutSize()` und `layoutContainer()`.

```
if ( nrows > 0 )
    ncols = (ncomponents + nrows - 1) / nrows;
else
    nrows = (ncomponents + ncols - 1) / ncols;
```

Ist die Anzahl der Zeilen gleich 0, so berechnet der Layoutmanager den Wert aus der Anzahl der Spalten.

Tipp

Existiert eine Anzahl Zeilen, so ist die Angabe für die Spalten völlig uninteressant. Der Wert sollte daher der Übersichtlichkeit halber auf 0 gesetzt werden.

14.8 Textkomponenten

Swing bietet eine Reihe von Textkomponenten:

- JTextField: einzeiliges Textfeld
- JFormattedTextField: einzeiliges Textfeld mit Formatierungsvorgaben
- JPasswordField: einzeilige Eingabe mit verdeckten Zeichen
- JTextArea: mehrzeiliges Textfeld
- JEditorPane: Editor-Komponente
- JTextPane: Spezialisierung der Editor-Komponente

14

Die JEditorPane ist die leistungsfähigste Komponente, die über sogenannte Editor-Kits reinen Text, HTML oder RTF darstellen und verwalten kann.

Hinweis

Swings Textkomponenten sind sehr leistungsfähig und allemal beeindruckend, da Swing ja allerlei Dinge von Hand erledigt, weil es nicht auf die nativen Textkomponenten vom grafischen Teil des Betriebssystems zurückgreift. Der aufwändigste Teil ist die korrekte Darstellung des Textes (die Java 2D übernimmt), aber Selektion, effektive Verwaltung von großen Textmengen, schnelles Scrolling, Tastaturkommandos, einfache Programmier-API sind weitere Anforderungen.

Viele wichtige Methoden sind in der Oberklasse javax.swing.text.JTextComponent zu finden. Zwar liegt diese Klasse im Paket javax.swing.text, doch liegen alle anderen Klassen »klassischerweise« unter javax.swing.

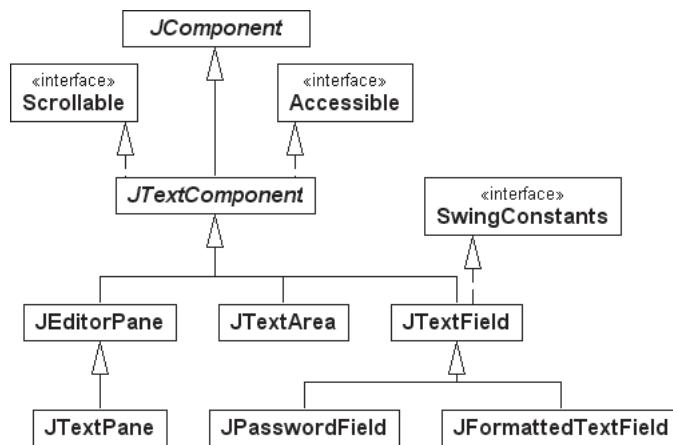


Abbildung 14.27: Vererbungsbeziehung der Swing-Textkomponenten

14.8.1 Text in einer Eingabezeile

Einzelne Textfelder werden mit der Klasse `JTextField` erstellt. Unterschiedliche Konstruktoren legen einen Start-String oder die Anzahl der Zeichen fest, die ein Textfeld anzeigen kann. Ein `JTextField` löst mit der `[←]`-Taste ein `ActionEvent` auf diesen Ereignistyp aus – das kennen wir schon von `JButton`.

Ein kleiner Rechner soll über eine Textzeile mit einer Länge von 20 Zeichen verfügen. Bei Aktivierung der `[←]`-Taste soll der Ausdruck berechnet werden und in der Textzeile erscheinen:

Listing 14.14: com/tutego/insel/ui/text/JTextFieldDemo.java, main()

```

 JFrame frame = new JFrame();
 frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

 final JTextField input = new JTextField( "12 * 3 + 2", 20 );
 input.addActionListener( new ActionListener() {
     @Override public void actionPerformed( ActionEvent e ) {
         try {
             input.setText( "" +
                 new ScriptEngineManager().getEngineByName("JavaScript")
                     .eval(input.getText()) );
         }
     }
 });
  
```

```

        catch ( ScriptException ex ) {
            ex.printStackTrace();
        }
    }
} );
frame.add( input );
frame.pack();
frame.setVisible( true );

```

```

class javax.swing.JTextField
extends JTextComponent

```

- JTextField()

Erzeugt ein leeres Textfeld.
- JTextField(int columns)

Erzeugt ein Textfeld mit einer gegebenen Anzahl von Spalten.
- JTextField(String text)

Erzeugt ein mit text initialisiertes Textfeld.
- JTextField(String text, int columns)

Erzeugt ein mit text initialisiertes Textfeld mit columns Spalten.

14

14.8.2 Die Oberklasse der Text-Komponenten (JTextComponent)

Alle Texteingabefelder unter Swing sind von der abstrakten Oberklasse `JTextComponent` abgeleitet. Die wichtigsten Methoden sind `setText(String)` und `getText()`, mit denen sich Zeichenketten setzen und erfragen lassen.

```

class javax.swing.text.JTextComponent
extends JComponent
implements Scrollable, Accessible

```

- String `getText()`

Liefert den Inhalt des Textfelds.
- String `getText(int offs, int len)`

Liefert den Inhalt des Textfelds von `offs` bis `offs + len`. Stimmen die Bereiche nicht, wird eine `BadLocationException` ausgelöst.

- `String getSelectedText()`
Liefert den selektierten Text. Keine Selektion ergibt die Rückgabe `null`.
- `void setText(String t)`
Setzt den Text neu.
- `void read(Reader in, Object desc) throws IOException`
Liest den Inhalt aus dem Reader in das Textfeld. `desc` beschreibt den Datenstrom näher, kann aber `null` sein. Die `read()`-Methode erzeugt intern ein neues Document-Objekt und verwirft das alte.
- `void write(Writer out) throws IOException`
Schreibt den Inhalt des Textfelds in den Writer.

14.9 Zeichnen von grafischen Primitiven

Ist das Fenster geöffnet, lässt sich etwas in dem Fenster zeichnen. Da sich die Wege zwischen AWT und Swing trennen, wollen wir mit dem AWT beginnen und dann alle weiteren Beispiele mit Swing bestreiten.

14.9.1 Die `paint()`-Methode für das AWT-Frame

Als einleitendes Beispiel soll uns genügen, einen Text zu platzieren. Dafür überschreiben wir die Methode `paint()` der Klasse `Frame` und setzen dort alles hinein, was gezeichnet werden soll, etwa Linien, Texte oder gefüllte Polygone. Der gewünschte Inhalt wird immer dann gezeichnet, wenn das Fenster neu aufgebaut wird oder wir von außen `repaint()` aufrufen, denn genau in diesem Fall wird das Grafiksystem `paint()` aufrufen und das Zeichnen anstoßen:

Listing 14.15: com/tutego/insel/ui/graphics/Bee.java

```
package com.tutego.insel.ui.graphics;

import java.awt.*;
import java.awt.event.*;

public class Bee extends Frame
{
    private static final long serialVersionUID = -3800165321162121122L;
```

```
public Bee()
{
    setSize( 500, 100 );

    addWindowListener( new WindowAdapter() {
        @Override
        public void windowClosing ( WindowEvent e ) { System.exit( 0 ); }
    } );
}

@Override
public void paint( Graphics g )
{
    g.drawString( "\"Maja, wo bist du?\" (Mittermeier)", 120, 60 );
}

public static void main( String[] args )
{
    new Bee().setVisible( true );
}
```

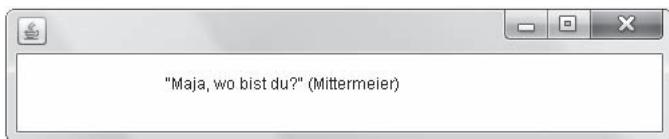


Abbildung 14.28: Ein Fenster mit gezeichnetem Inhalt

Der Grafikkontext `Graphics`

Das Grafiksystem ruft von unserem Programm die `paint()`-Methode auf und übergibt ein Objekt vom Typ `Graphics` – beziehungsweise `Graphics2D`, wie wir später sehen werden. Dieser Grafikkontext bietet verschiedene Methoden zum Setzen von Zeichenzusänden und zum Zeichnen selbst, etwa von Linien, Kreisen, Ovalen, Rechtecken, Zeichenfolgen oder Bildern. Dies funktioniert auch dann, wenn die Zeichenfläche nicht direkt sichtbar ist, wie bei Hintergrundgrafiken.

Das `Graphics`-Objekt führt Buch über mehrere Dinge:

- die Komponente, auf der gezeichnet wird (hier ist das erst einmal das rohe Fenster)
- Koordinaten des Bildbereichs und des Clipping-Bereichs. Die Zeichenoperationen außerhalb des Clipping-Bereichs werden nicht angezeigt. Daher wird ein Clipping-Bereich auch Beschnitt-Bereich genannt.
- den aktuellen Zeichensatz (`java.awt.Font`) und die aktuelle Farbe (`java.awt.Color`)
- die Pixeloperation (Xor⁸ oder Paint)
- die Translation – eine Verschiebung vom Nullpunkt

Wir können nur in der `paint()`-Methode auf das `Graphics`-Objekt zugreifen. Diese wird immer dann aufgerufen, wenn die Komponente neu gezeichnet werden muss. Dies nutzen wir, um einen Text zu schreiben. Dem Bee-Beispiel ist zu entnehmen, dass die Methode `drawString(String text, int x, int y)` einen Text in den Zeichenbereich des Grafikkontexts schreibt. Im Folgenden werden wir noch weitere Methoden kennenlernen.



Hinweis

Etwas ungewöhnlich ist die Tatsache, dass der Nullpunkt nicht oben links in den sichtbaren Bereich fällt, sondern dass die Titelleiste den Nullpunkt überdeckt. Um an die Höhe der Titelleiste zu kommen und die Zeichenoperationen so zu verschieben, dass sie in den sichtbaren Bereich fallen, wird ein `java.awt.Insets`-Objekt benötigt. Ist `f` ein `Frame`-Objekt, liefert `f.getInsets().top` die Höhe der Titelleiste.

14.9.2 Die ereignisorientierte Programmierung ändert Fensterinhalte

Der Einstieg in die Welt der Grafikprogrammierung mag etwas seltsam erscheinen, weil in der prozeduralen, nicht ereignisgesteuerten Welt die Programmierung anders verlief. Es gab einige Funktionen, mit denen sich direkt sichtbar auf dem Bildschirm operieren ließ. Ein Beispiel⁹ aus der C128-Zeit:

10 COLOR 0,1	:REM SELECT BACKGROUND COLOR
20 COLOR 1,3	:REM SELECT FOREGROUND COLOR

⁸ Zur Bewegung des Grafik-Cursors wird gern eine Xor-Operation eingesetzt. Obwohl dies absolut einfach erscheint, ist die Realisierungsidee patentiert.

⁹ Commodore 128 System Guide, Commodore Business Machines, Inc. 1985, online zugänglich unter <http://members.tripod.com/~rvbelzen/c128sg/toc.htm>.

```

30 COLOR 4,1 :REM SELECT BORDER COLOR
40 GRAPHIC 1,1 :REM SELECT BIT MAP MODE
60 CIRCLE 1,160,100,40,40 :REM DRAW A CIRCLE
70 COLOR 1,6 :REM CHANGE FOREGROUND COLOR
80 BOX 1,20,60,100,140,0,1 :REM DRAW A BLOCK
90 COLOR 1,9 :REM CHANGE FOREGROUND COLOR
100 BOX 1,220,62,300,140,0,0 :REM DRAW A BOX
110 COLOR 1,9 :REM CHANGE FOREGROUND COLOR
120 DRAW 1,20,180 TO 300,180 :REM DRAW A LINE
130 DRAW 1,250,0 TO 30,0 TO 40,40 TO 250,0 :REM DRAW A TRIANGLE
140 COLOR 1,15 :REM CHANGE FOREGROUND COLOR
150 DRAW 1,160,160 :REM DRAW A POINT
160 PAINT 1,150,97 :REM PAINT IN CIRCLE
170 COLOR 1,5 :REM CHANGE FOREGROUND COLOR
180 PAINT 1,50,25 :REM PAINT IN TRIANGLE
190 COLOR 1,7 :REM CHANGE FOREGROUND COLOR
200 PAINT 1,225,125 :REM PAINT IN EMPTY BOX
210 COLOR 1,11 :REM CHANGE FOREGROUND COLOR
220 CHAR 1,11,24,"GRAPHIC EXAMPLE" :REM DISPLAY TEXT
230 FOR I=1 TO 5000:NEXT:GRAPHIC 0,1:COLOR 1,2

```

Diese Vorgehensweise funktioniert in Java (und auch in vielen modernen Systemen) nicht mehr. Auch mit Objektorientierung hat sie nicht viel zu tun!

In Java führt ein Repaint-Ereignis zum Aufruf der `paint()`-Methode. Dieses Ereignis kann ausgelöst werden, wenn der Bildschirm zum ersten Mal gezeichnet wird, aber auch, wenn Teile des Bildschirms verdeckt werden. Falls das Repaint-Ereignis eintritt, springt das Java-System in die `paint()`-Methode, in der der Bildschirm aufgebaut werden kann. Nur dort finden die Zeichenoperationen statt. Wenn wir nun selbst etwas zeichnen wollen, kann das nur in der `paint()`-Methode geschehen, beziehungsweise in Methoden, die von `paint()` aufgerufen werden. Wenn wir aber selbst etwas zeichnen wollen, wie lässt sich `paint()` dann parametrisieren?

Um mit diesem Problem umzugehen, müssen wir der `paint()`-Methode Informationen mitgeben. Diese Informationen kann `paint()` nur aus den Objektattributen beziehen. Daher implementieren wir eine Unterklasse einer Komponente, die eine `paint()`-Methode besitzt. Anschließend können wir Objektzustände ändern, sodass `paint()` neue Werte bekommt und somit gewünschte Inhalte zeichnen kann.

14.9.3 Zeichnen von Inhalten auf ein JFrame

Wenn Swing eine Komponente zeichnet, ruft es automatisch die Methode `paint()` auf. Um eine Grafik selbst in ein Fenster zu zeichnen, ließe sich von `JFrame` eine Unterklasse bilden und `paint()` überschreiben – das ist jedoch nicht der übliche Weg.

Stattdessen wählen wir einen anderen Ansatz, der sogar unter AWT eine gute Lösung ist. Wir bilden eine eigene Komponente, eine Unterklasse von `JPanel` (unter AWT `Panel`, was wir aber nicht mehr weiter verfolgen wollen), und setzen diese auf das Fenster. Wird das Fenster neu gezeichnet, gibt das Grafiksystem den Zeichenauftrag an die Kinder weiter, also an unser spezielles `JPanel`, und ruft die überschriebene `paint()`-Methode auf. Allerdings überschreiben eigene Unterklassen von Swing-Komponenten im Regelfall nicht `paint()`, sondern `paintComponent()`. Das liegt daran, dass Swing in `paint()` zum Beispiel noch Rahmen zeichnet und sich um eine Pufferung des Bildschirminhalts zur Optimierung kümmert. So ruft `paint()` die drei Methoden `paintComponent()`, `paintBorder()` und `paintChildren()` auf, und bei einer Neudarstellung kümmert sich ein `RepaintManager` um eine zügige Darstellung mithilfe der gepufferten Inhalte, was bei normalen Swing-Interaktionskomponenten wie Schaltflächen wichtig ist.

Damit ist die Darstellung von Inhalten in einem `JFrame` einfach. Wir importieren drei Klassen, `JPanel` und `JFrame` aus `javax.swing` sowie `Graphics` aus `java.awt`. Dann bilden wir eine Unterklasse von `JPanel` und überschreiben `paintComponent()`:

Listing 14.16: com/tutego/insel/ui/graphics/DrawFirstLine.java, Teil 1

```
package com.tutego.insel.ui.graphics;

import java.awt.Graphics;
import javax.swing.*;

class DrawPanel extends JPanel
{
    @Override
    protected void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        g.drawLine( 10, 10, 100, 50 );
    }
}
```

Die Methode `paintComponent()` besitzt in der Oberklasse die Sichtbarkeit `protected`, was wir beibehalten sollten; die Methode wird nicht von uns von anderer Stelle aufgerufen, daher muss eine Unterklasse die Sichtbarkeit nicht zu `public` erweitern. Der Aufruf von `super.paintComponent()` ist immer dann angebracht, wenn die Oberklasse ihre Inhalte zeichnen soll. Bei vollständig eigenem Inhalt ist das nicht notwendig.

Der letzte Schritt ist ein Testprogramm, das ein Exemplar des spezialisierten `JPanel` bildet und auf den `JFrame` setzt:

Listing 14.17: com/tutego/insel/ui/graphics/DrawFirstLine.java, Teil 2

```
public class DrawFirstLine
{
    public static void main( String[] args )
    {
        JFrame f = new JFrame();
        f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        f.setSize( 100, 100 );
        f.add( new DrawPanel() );
        f.setVisible( true );
    }
}
```

Die Lösung mit dem `JPanel` muss nicht die Höhe der Titelleiste berücksichtigen; die Komponente `JPanel`, die auf das Fenster gesetzt wird, befindet sich korrekt unterhalb der Titelleiste, und die Zeichenfläche liegt nicht verdeckt unter der Titelleiste.

14.9.4 Linien

Bei Linien müssen wir uns von der Vorstellung trennen, die uns die analytische Geometrie nahelegt. Laut Euklid ist dort eine Linie als kürzeste Verbindung zwischen zwei Punkten definiert. Da Linien eindimensional sind, besitzen sie eine Länge aus unendlich vielen Punkten, doch keine wirkliche Breite. Auf dem Bildschirm besteht eine Linie nur aus endlich vielen Punkten, und wenn eine Linie gezeichnet wird, werden Pixel gesetzt, die nahe an der wirklichen Linie sind. Die Punkte müssen passend in ein Raster gesetzt werden, und so kommt es vor, dass die Linie in Stücke zerbrochen wird. Dieses Problem gibt es bei allen grafischen Operationen, da von Fließkommawerten eine Abbildung auf Ganzzahlen, in unserem Fall absolute Koordinaten des Bildschirms, durchgeführt werden muss. Eine bessere Darstellung der Linien und Kurven ist durch *Antialiasing* zu erreichen. Dies ist eine Art Weichzeichnung mit nicht nur einer Farbe, sondern mit

Abstufungen, sodass die Qualität auf dem Bildschirm wesentlich besser ist. Auch bei Zeichensätzen ist dadurch eine merkliche Verbesserung der Lesbarkeit auf dem Bildschirm zu erzielen.

Die Methode `drawLine()` wurde schon im ersten Beispiel vorgestellt.

```
abstract class java.awt.Graphics
```

- `abstract void drawLine(int x1, int y1, int x2, int y2)`
Zeichnet eine Linie zwischen den Koordinaten (x_1, y_1) und (x_2, y_2) in der Vordergrundfarbe.

zB Beispiel

Setze einen Punkt an die Stelle (x, y) :

```
g.drawLine( x, y, x, y );
```

14.9.5 Rechtecke

Als Nächstes werfen wir einen Blick auf die Methoden, die uns Rechtecke zeichnen lassen. Die Rückgabe ist – wie auch bei den anderen Zeichenmethoden – immer `void`. Es ist nicht so, dass die Methoden durch einen Wahrheitswert mitteilen, ob ein tatsächlicher Zeichenbereich gefüllt werden konnte. Liegen die Koordinaten des zu zeichnenden Objekts nicht im Sichtfenster, geschieht einfach gar nichts. Die Zeichenmethode ist nicht in der Lage, dies dem Aufrufer in irgendeiner Form mitzuteilen.

```
abstract class java.awt.Graphics
```

- `void drawRect(int x, int y, int width, int height)`
Zeichnet ein Rechteck in der Vordergrundfarbe. Das Rechteck ist `width + 1` Pixel breit und `height + 1` Pixel hoch.
- `void abstract fillRect(int x, int y, int width, int height)`
Zeichnet ein gefülltes Rechteck in der Vordergrundfarbe. Das Rechteck ist `width` Pixel breit und `height` Pixel hoch.
- `void abstract drawRoundRect(int x, y, int width, height, int arcWidth, arcHeight)`
Zeichnet ein abgerundetes Rechteck in der Vordergrundfarbe. Das Rechteck ist `width + 1` Pixel breit und `height + 1` Pixel hoch. `arcWidth` gibt den horizontalen und `arcHeight` den vertikalen Durchmesser der Kreisbögen der Ränder an.

- void abstract fillRoundRect(int x, int y, int width, height, int arcWidth, arcHeight)
Zeichnet wie drawRoundRect(), nur gefüllt.
- void draw3DRect(int x, int y, int width, int height, boolean raised)
Zeichnet ein dreidimensional angedeutetes Rechteck in der Vordergrundfarbe. Der Parameter raised gibt an, ob das Rechteck optisch erhöht oder vertieft wirken soll. Die Farben für den Effekt werden aus den Vordergrundfarben gewonnen.
- void fill3DRect(int x, int y, int width, int height, boolean raised)
Zeichnet wie draw3DRect(), nur gefüllt.

Hinweis

Die Breiten der Rechtecke bei den Methoden drawRect() und fillRect() unterscheiden sich. drawRect(0, 0, 10, 10) zeichnet ein 11×11 Pixel breites Rechteck, und fillRect(0, 0, 10, 10) zeichnet ein 10×10 Pixel breites Rechteck.

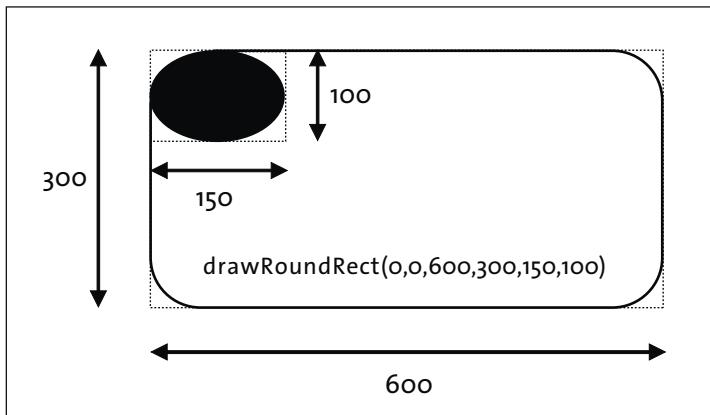


Abbildung 14.29: Beispiel für drawRoundRect()

14.9.6 Zeichenfolgen schreiben

Die Methode `drawString()` bringt eine Unicode-Zeichenkette mit sogenannten *Glyphen*, das sind konkrete grafische Darstellungen eines Zeichens, auf den Bildschirm – die Darstellung von Zeichen übernimmt in Java der Font-Renderer.

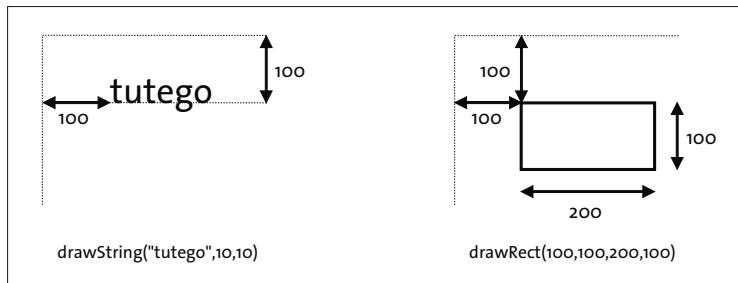


Abbildung 14.30: Koordinatenangaben von `drawString()` und `drawRect()` im Vergleich

Die Parameter von `drawString()` beschreiben die zu schreibende Zeichenkette sowie die x- und y-Koordinaten. Die Koordinaten bestimmen die Position der *Schriftlinie* – auch *Grundlinie* genannt (engl. *baseline*) –, auf der die Buchstaben stehen.

```
abstract class java.awt.Graphics
```

- `abstract void drawString(String s, int x, int y)`

Schreibt einen String in der aktuellen Farbe und dem aktuellen Zeichensatz.

- `abstract void drawString(AttributedCharacterIterator iterator, int x, int y)`

Schreibt einen String, der durch den Attribut-Iterator gegeben ist.

- `abstract void drawChars(char[] data, int offset, int length, int x, int y)`

- `abstract void drawBytes(byte[] data, int offset, int length, int x, int y)`

Schreibt die Zeichenkette und bezieht die Daten aus einem Char- bzw. Byte-Feld.



Hinweis

Die Klasse `TextLayout` bietet weitergehende Möglichkeiten, etwa Farbhervorhebung, Cursor-Funktionalität oder über `TextHitInfo` Tests, welches Zeichen sich an welcher Koordinate befindet, was bei Mausklicks nützlich ist. Ein `draw()` vom `TextLayout` bringt den Textbaustein auf den Schirm. Die Dokumentation listet etwas Quellcode zur Verdeutlichung auf.

14.9.7 Die Font-Klasse

Die Methode `drawString()` verwendet immer den aktuellen Zeichensatz. Um diesen zu ändern, benutzen wir auf dem aktuellen `Graphics`-Objekt die Methode `setFont()`. Der Überabeparameter ist ein `Font`-Objekt, das wir woanders erfragen oder vorher erzeugen müssen.



Abbildung 14.31: UML-Diagramm der Klasse Font

```
class java.awt.Font
    implements Serializable
```

- `Font(String name, int style, int size)`

Erzeugt ein `Font`-Objekt mit einem gegebenen Namen, einem gegebenen Stil und einer gegebenen Größe.

Für den Stil deklariert die `Font`-Klasse drei symbolische Konstanten: `Font.PLAIN` sowie `Font.ITALIC` und `Font.BOLD`, sodass keine Ganzzahlen als Argumente zu übertragen sind. Die Stil-Attribute können mit dem binären Oder oder dem arithmetischen Plus verbunden werden; ein fetter und kursiver Zeichensatz erreicht `Font.BOLD | Font.ITALIC` (beziehungsweise `Font.BOLD + Font.ITALIC`). Die Größe ist in Punkt angegeben, wobei ein Punkt 1/72 Zoll (in etwa 0,376 mm) entspricht. Der Name des Zeichensatzes ist entweder physikalisch (zum Beispiel »Verdana« oder »Geneva«) oder logisch mit `Font`-Konstanten `DIALOG`, `DIALOG_INPUT`, `SANS_SERIF`, `SERIF` und `MONOSPACED`, die später auf die physikalischen `Font`-Namen übertragen werden.¹⁰

zB Beispiel

Ein `Font`-Objekt erzeugen:

```
Font f = new Font( Font.SERIF, Font.PLAIN, 14 );
```

Häufig wird dieses Zeichensatz-Objekt sofort in `setFont()` genutzt, so wie:

```
setFont( new Font( "Verdana", Font.BOLD, 20 ) );
```



Hinweis

Die Dokumentation spricht zwar von »Punkt«, in Java sind aber Punkt und Pixel bisher identisch. Würde das Grafiksystem wirklich in Punkt arbeiten, müsste es die Bildschirmauflösung und den Monitor mit berücksichtigen.

Ist im Programm der aktuell verwendete Zeichensatz nötig, können wir `getFont()` von der `Graphics`-Klasse verwenden:

¹⁰ Die Webseite <http://java.sun.com/javase/7/docs/technotes/guides/intl/fontconfig.html> beschreibt diese Umsetzung.

```
abstract class java.awt.Graphics
```

- abstract Font getFont()
Liefert den aktuellen Zeichensatz.
- abstract Font setFont(Font f)
Setzt den Zeichensatz für das Graphics-Objekt.

14.9.8 Farben mit der Klasse Color

Der Einsatz von Farben und Transparenzen ist in Java-Programmen dank der Klasse `java.awt.Color` einfach. Ein `Color`-Objekt repräsentiert üblicherweise einen Wert aus dem sRGB-Farbraum (Standard-RGB), kann aber auch andere Farträume über den Basis-typ `java.awt.color.ColorSpace` darstellen (wir werden das nicht weiter verfolgen).

Die Klasse `Color` stellt Konstanten für Standard-Farben und einige Konstruktoren sowie Anfragemethoden bereit. Außerdem gibt es Methoden, die abgewandelte `Color`-Objekte liefern – das ist nötig, da `Color`-Objekte wie `String` oder `File` immutable sind.

```
class java.awt.Color
    implements Paint, Serializable
```

- `Color(float r, float g, float b)`
Erzeugt ein `Color`-Objekt mit den Grundfarben Rot, Grün und Blau. Die Werte müssen im Bereich 0.0 bis 1.0 liegen, sonst folgt eine `IllegalArgumentException`.
- `Color(int r, int g, int b)`
Erzeugt ein `Color`-Objekt mit den Grundfarben Rot, Grün und Blau. Die Werte müssen im Bereich 0 bis 255 liegen, sonst folgt eine `IllegalArgumentException`.
- `Color(int rgb)`
Erzeugt ein `Color`-Objekt aus dem `rgb`-Wert, der die Farben Rot, Grün und Blau kodiert. Der Rotanteil befindet sich unter den Bits 16 bis 23, der Grünanteil in 8 bis 15 und der Blauanteil in 0 bis 7. Da ein Integer immer 32 Bit breit ist, ist jede Farbe durch 1 Byte (8 Bit) repräsentiert. Die Farbinformationen werden nur aus den 24 Bit genommen. Sonstige Werte werden einfach nicht betrachtet und mit einem Alpha-Wert gleich 255 überschrieben.
- `Color(int r, int g, int b, int a)`
- `Color(float r, float g, float b, float a)`
Erzeugt ein `Color`-Objekt mit Alpha-Wert für Transparenz.

- static Color decode(String nm) throws NumberFormatException
Liefert die Farbe von nm. Die Zeichenkette ist hexadezimal als 24-Bit-Integer kodiert, etwa #00AAFF. Eine Alternative ist new Color(Integer.parseInt(colorHexString, 16));.



Hinweis

Menschen unterscheiden Farben an den drei Eigenschaften Farbton, Helligkeit und Sättigung. Die menschliche Farbwahrnehmung kann etwa zweihundert Farbtöne unterscheiden. Diese werden durch die Wellenlänge des Lichts bestimmt. Die Lichtintensität und Empfindlichkeit unserer Rezeptoren lässt uns etwa fünfhundert Helligkeitsstufen unterscheiden. Bei der Sättigung handelt es sich um eine Mischung mit weißem Licht. Hier erkennen wir etwa zwanzig Stufen. Unser visuelles System kann somit ungefähr zwei Millionen ($200 \times 500 \times 20$) Farbnuancen unterscheiden.

Zufällige Farbblöcke zeichnen

Um die Möglichkeiten der Farbgestaltung einmal zu beobachten, betrachten wir die Ausgabe eines Programms, das Rechtecke mit wahllosen Farben anzeigt:

Listing 14.18: com/tutego/insel/ui/graphics/ColorBox.java

```
package com.tutego.insel.ui.graphics;

import java.awt.*;
import java.util.Random;
import javax.swing.*;

public class ColorBox extends JPanel
{
    private static final long serialVersionUID = -2294685016438617741L;
    private static final Random r = new Random();

    @Override
    protected void paintComponent( Graphics g )
    {
        super.paintComponent( g );

        for ( int y = 12; y < getHeight() - 25; y += 30 )
            for ( int x = 12; x < getWidth() - 25; x += 30 )
```

```
{  
    g.setColor( new Color( r.nextInt(256), r.nextInt(256), r.nextInt(256) ) );  
    g.fillRect( x, y, 25, 25 );  
    g.setColor( Color.BLACK );  
    g.drawRect( x - 1, y - 1, 25, 25 );  
}  
}  
  
public static void main( String[] args )  
{  
    JFrame f = new JFrame( "Neoplastizismus" );  
    f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );  
    f.setSize( 300, 300 );  
    f.add( new ColorBox() );  
    f.setVisible( true );  
}  
}
```

14



Abbildung 14.32: Programmierter Neoplastizismus

Das Fenster der Applikation hat eine gewisse Größe, die wir mit `size()` in der Höhe und Breite abfragen. Anschließend erzeugen wir Blöcke, die mit einer zufälligen Farbe gefüllt sind. `fillRect()` übernimmt diese Aufgabe. Da die gefüllten Rechtecke immer in der Vordergrundfarbe gezeichnet werden, setzen wir den Zeichenstift durch die Methode `setColor()`, die natürlich eine Objektmethode von `Graphics` ist. Entsprechend gibt es eine

korrespondierende Methode `getColor()`, die die aktuelle Vordergrundfarbe als `Color`-Objekt zurückgibt (diese Methode darf nicht mit den beiden Methoden aus der `Color`-Klasse `getColor(String)` beziehungsweise `getColor(String, Color)` verwechselt werden).

```
abstract class java.awt.Graphics
```

- `abstract void setColor(Color c)`

Setzt die aktuelle Farbe, die dann von den Zeichenmethoden berücksichtigt wird.

- `abstract Color getColor()`

Liefert die aktuelle Farbe.

Vordefinierte Farben

Wenn wir Farben benutzen wollen, sind schon viele Werte vordefiniert, wie `Color.WHITE`. In der Klasse `jawa.awt.Color` sind dazu viele Zeilen der Form

```
/**  
 * The color white.  
 */  
  
public static final Color WHITE = new Color( 255, 255, 255 );
```

platziert.

Die folgende Tabelle zeigt die deklarierten Konstanten inklusive Werteberechnung für die Farbtupel:

Color-Konstante	Rot	Grün	Blau
WHITE	255	255	255
BLACK	0	0	0
RED	255	0	0
GREEN	0	255	0
BLUE	0	0	255
YELLOW	255	255	0
MAGENTA	255	0	255
CYAN	0	255	255
PINK	255	175	175

Tabelle 14.2: Farbanteile für die vordefinierten Standardfarben

Color-Konstante	Rot	Grün	Blau
ORANGE	255	200	0
LIGHT_GRAY	192	192	192
DARK_GRAY	64	64	64

Tabelle 14.2: Farbanteile für die vordefinierten Standardfarben (Forts.)

Alle Farbnamen gibt es auch kleingeschrieben. Zwar stammt von Sun die Namenskonvention, dass Konstanten großgeschrieben werden, aber bei den Farbnamen wurde das erst in Java 1.4 nachgeholt – sieben Jahre später.

14.10 Zum Weiterlesen

»Java 7 – Mehr als eine Insel« bietet drei Extrakapitel für grafische Programmierung, die Swing für grafische Oberflächen, das AWT und Java 2D für die Zeichenfunktionalität sowie JavaFX vorstellen.



Kapitel 15

Einführung in Dateien und Datenströme

»Schlagfertigkeit ist jede Antwort, die so klug ist, dass der Zuhörer wünscht, er hätte sie gegeben.«

– Elbert Green Hubbard (1856–1915)

Computer sind für uns so nützlich, weil sie Daten bearbeiten. Der Bearbeitungszyklus beginnt mit dem Einlesen der Daten, umfasst das Verarbeiten und endet mit der Ausgabe der Daten. In der deutschsprachigen Informatikliteratur wird deswegen auch vom EVA-Prinzip der Datenverarbeitungsanlagen gesprochen. In frühen EDV-Zeiten wurde die Eingabe vom Systemoperator auf Lochkarten gestanzt. Glücklicherweise sind diese Zeiten vorbei. Heutzutage speichern wir unsere Daten in Dateien (engl. *files*¹) und Datenbanken ab. Es ist wichtig, anzumerken, dass eine Datei nur in ihrem Kontext interessant ist, andernfalls beinhaltet sie für uns keine Information – die Sichtweise auf eine Datei ist demnach wichtig. Auch ein Programm besteht aus Daten und wird oft in Form einer Datei repräsentiert.

Das `java.io`-Paket bietet zahlreiche dateiorientierte Operationen. Dazu zählen Möglichkeiten, Dateien anzulegen, zu löschen, umzubenennen und sie in Verzeichnissen zu strukturieren. Diese gehören schon seit Java 1.0 zur API, und wir wollen uns zuerst damit beschäftigen. Java 7 krempelt diese API um und macht insbesondere die zentrale `File`-Klasse überflüssig. Die neue API liegt getrennt in einem neuen Paket namens `java.nio.file` und wird in dem Buch intensiv nach der traditionellen API vorgestellt.

15

15.1 Datei und Verzeichnis

Die Klasse `File` hat die Aufgabe, Dateioperationen plattformunabhängig durchzuführen. Plattformunabhängigkeit ist aber auch eine Einschränkung, denn wie sollen Rechte ver-

¹ Das englische Wort »file« geht auf das lateinische Wort »filum« zurück. Dies bezeichnete früher eine auf Draht aufgereihte Sammlung von Schriftstücken.

geben werden, wenn dies etwa der Macintosh mit Mac OS X oder ein Android Smartphone nicht unterstützt? Unix und Windows haben zwei völlig verschiedene Ansätze zur Rechteverwaltung. Die `File`-Klasse ignoriert diese Frage und bietet überhaupt keine Lösung. In NIO.2 jedoch geht Java einen anderen Weg und neigt wieder deutlich mehr zu plattformspezifischen Details.

15.1.1 Dateien und Verzeichnisse mit der Klasse File

Ein `File`-Objekt repräsentiert einen Datei- oder Verzeichnisnamen im Dateisystem. Die Datei oder das Verzeichnis, das das `File`-Objekt beschreibt, muss nicht physisch existieren. Der Verweis wird durch einen Pfadnamen spezifiziert. Dieser kann absolut oder relativ zum aktuellen Verzeichnis angegeben werden.

zB Beispiel

Erzeuge ein `File`-Objekt für das Laufwerk C:/:

```
File f = new File( "C:/" );
System.out.println( f ); // C:\
```

Folgende Konstruktoren erzeugen ein `File`-Objekt:

```
class java.io.File
    implements Serializable, Comparable<File>
```

- `File(String pathname)`
Erzeugt ein `File`-Objekt aus einem Dateinamen.
- `File(String parent, String child)`
Setzt ein neues `File`-Objekt aus einem Basisverzeichnis und einem weiteren Teil zusammen, der auch wieder ein Verzeichnis oder ein Dateiname sein kann.
- `File(URI uri)`
Erfragt von `uri` den Pfadnamen (`uri.getPath()`) und erzeugt ein neues `File`-Objekt. Ist `uri` gleich `null`, folgt eine `NullPointerException`. Ist die `URI` falsch formuliert, gibt es eine `IllegalArgumentException`.

Die Pfadangabe kann in allen Fällen absolut sein, muss es aber nicht.

Pfadtrenner

Die Angabe des Pfades ist wegen der Pfadtrenner plattformabhängig. Auf Windows-Rechnern trennt ein Backslash »\« die Pfade, auf Unix-Maschinen ein normaler Slash »/« und unter dem älteren MAC OS 9 ein Doppelpunkt.

Glücklicherweise speichert die Klasse `File` den Pfadtrenner in zwei öffentlichen Konstanten: `File.separatorChar`² ist ein `char`, und `File.separator` stellt den Pfadtrenner als `String` bereit (dies ist wiederum auf `System.getProperty("file.separator")` zurückzuführen).

Hinweis

Wie bei den Dateitrennern gibt es einen Unterschied bei der Darstellung des Wurzelverzeichnisses. Unter Unix ist dies ein einzelnes Divis »/«, und unter Windows steht die Laufwerksbezeichnung vor dem Doppelpunkt und dem Backslash-Zeichen (»Z:\«).

Namen erfragen und auflösen

Mit einem `File`-Objekt erfragen ganz unterschiedliche Methoden den Dateinamen, den kompletten Pfad, das vorangehende Verzeichnis und ob eine Angabe absolut oder relativ ist. Bei einigen Methoden lässt sich wählen, ob die Rückgabe ein `String`-Objekt mit dem Dateinamen sein soll oder ein `File`-Objekt.

Beispiel

zB

Liefere einen Dateinamen, bei dem die relativen Bezüge aufgelöst sind:

```
try
{
    File f = new File("C:/./WasNDas//..\\Programme/").getCanonicalFile();
    System.out.println( f );                                // C:\\Programme
}
catch ( IOException e ) { e.printStackTrace(); }
```

Nicht viele Methoden der `File`-Klasse lösen eine `IOException` aus, diese ist eher eine Ausnahme.

² Eigentlich sollte der Variablenname großgeschrieben werden, da die Variable als `public static final char` eine Konstante ist.



Abbildung 15.1: Klassendiagramm für File

```
class java.io.File  
implements Serializable, Comparable<File>
```

- `String getName()`
Gibt den Dateinamen zurück.
- `String getPath()`
Gibt den Pfadnamen zurück.
- `String getAbsolutePath()`
- `File getAbsoluteFile()`
Liefert den absoluten Pfad. Ist das Objekt kein absoluter Pfadname, so wird ein Objekt aus dem aktuellen Verzeichnis, einem Separator-Zeichen und dem Dateinamen aufgebaut.
- `String getCanonicalPath () throws IOException`
- `File getCanonicalFile() throws IOException`
Gibt den Pfadnamen des Dateiobjekts zurück, der keine relativen Pfadangaben mehr enthält. Kann im Gegensatz zu den anderen Pfadmethoden eine `IOException` aufrufen, da mitunter verbotene Dateizugriffe erfolgen.
- `String getParent(), File getParentFile()`
Gibt den Pfad des Vorgängers als `String`- oder `File`-Objekt zurück. Die Rückgabe ist `null`, wenn es keinen Vater gibt, etwa beim Wurzelverzeichnis.
- `boolean isAbsolute()`
Liefert `true`, wenn der Pfad in der systemabhängigen Notation absolut ist.

15.1.2 Verzeichnis oder Datei? Existiert es?

Das `File`-Objekt muss nicht unbedingt eine existierende Datei oder ein existierendes Verzeichnis repräsentieren. Für Dateioperationen mit `File`-Objekten und nachfolgendem Zugriff testet `exists()`, ob die Datei oder das Verzeichnis tatsächlich vorhanden ist. Da nun aber ein `File`-Objekt Dateien sowie Verzeichnisse gleichzeitig repräsentiert, ermöglichen `isDirectory()` und `isFile()` eine genauere Aussage über den `File`-Typ. Es kann gut sein, dass für eine `File` weder `isDirectory()` noch `isFile()` die Rückgabe `true` liefert (in Java können nur normale Dateien erzeugt werden).

```
class java.io.File
    implements Serializable, Comparable<File>
```

- `boolean exists()`
Liefert true, wenn das File-Objekt eine existierende Datei oder einen existierenden Ordner repräsentiert.
- `boolean isDirectory()`
Gibt true zurück, wenn es sich um ein Verzeichnis handelt.
- `boolean isFile()`
Liefert true, wenn es sich um eine »normale« Datei handelt (kein Verzeichnis und keine Datei, die vom zugrunde liegenden Betriebssystem als besonders markiert wird, keine Blockdateien, Verknüpfungen).

15.1.3 Verzeichnis- und Dateieigenschaften/-attribute

Eine Datei oder ein Verzeichnis besitzt zahlreiche Eigenschaften, die sich mit Anfragemethoden auslesen lassen. In einigen wenigen Fällen lassen sich die Attribute auch ändern.

```
class java.io.File
    implements Serializable, Comparable<File>
```

- `boolean canExecute()`
- `boolean canRead()`
- `boolean canWrite()`
- Liefert true, wenn die Ausführungsrechte/Leserechte/Schreibrechte gesetzt sind.
- `long length()`
Gibt die Länge der Datei in Byte zurück oder 0L, wenn die Datei nicht existiert oder es sich um ein Verzeichnis handelt.

Änderungsdatum einer Datei

Eine Datei verfügt unter jedem Dateisystem nicht nur über Attribute wie Größe und Rechte, sondern verwaltet auch das Datum der letzten Änderung. Letzteres nennt sich *Zeitstempel*. Die File-Klasse verfügt zum Abfragen dieser Zeit über die Methode `lastModified()` und zum Setzen über `setLastModified()`.

Die Methode `setLastModified()` ändert (wenn möglich) den Zeitstempel, und ein anschließender Aufruf von `lastModified()` liefert die gesetzte Zeit (womöglich gerundet) zurück. Die Methode ist von vielfachem Nutzen, aber in Hinblick auf die Sicherheit bedenklich, denn ein Programm kann den Dateiinhalt einschließlich des Zeitstempels ändern. Auf den ersten Blick ist nicht mehr erkennbar, dass eine Veränderung der Datei vorgenommen wurde. Doch die Methode ist von größerem Nutzen bei der Programmierung, wo Quellcodedateien etwa mit Objektdateien verbunden sind. Nur über einen Zeitstempel ist eine einigermaßen intelligente Projektdateiverwaltung möglich.

Dabei bleibt es verwunderlich, warum `lastModified()` nicht als veraltet ausgezeichnet ist und zu `getLastModified()` wurde, wo doch nun die passende Methode zum Setzen der Namensgebung genügt.

```
class java.io.File
    implements Serializable, Comparable<File>
```

- `long lastModified()`

Liefert den Zeitpunkt, zu dem die Datei zum letzten Mal geändert wurde. Die Zeit wird in Millisekunden ab dem 1. Januar 1970, 00:00:00 UTC, gemessen. Die Methode liefert 0, wenn die Datei nicht existiert oder ein Ein-/Ausgabefehler auftritt.

- `boolean setLastModified(long time)`

Setzt die Zeit (wann die Datei zuletzt geändert wurde). Die Zeit ist wiederum in Millisekunden seit dem 1. Januar 1970 angegeben. Ist das Argument negativ, dann wird eine `IllegalArgumentException` ausgelöst.

15

Hinweis

Zwar lässt Java die Ermittlung der Zeit der letzten Änderung zu, doch gilt dies nicht für die Erzeugungszeit. Das Standard-Dateisystem von Unix-Systemen speichert diese Zeit nicht. Windows speichert sie hingegen schon, sodass hier grundsätzlich der Zugriff, etwa über JNI, möglich wäre. Legt ein Java-Programm die Dateien an, deren Anlegezeiten später wichtig sind, müssen die Zeiten beim Anlegen gemessen und gespeichert werden. Falls die Datei nicht verändert wird, stimmt `lastModified()` mit der Anlegezeit überein.

15.1.4 Umbenennen und Verzeichnisse anlegen

Mit `mkdir()` lassen sich Verzeichnisse anlegen und mit `renameTo()` Dateien oder Verzeichnisse umbenennen.

```
class java.io.File
    implements Serializable, Comparable<File>
```

- boolean mkdir()
 Legt das Unterverzeichnis an.
- boolean mkdirs()
 Legt das Unterverzeichnis inklusive weiterer Verzeichnisse an.
- boolean renameTo(File d)
 Benennt die Datei in den Namen um, der durch das File-Objekt d gegeben ist. Ging alles gut, wird true zurückgegeben. Bei zwei Dateinamen alt und neu benennt new File(alt).renameTo(new File(neu)); die Datei um. Die Methode muss vom Betriebssystem nicht atomar ausgeführt werden, und die tatsächliche Implementierung ist von der JVM und vom Betriebssystem abhängig.

Über renameTo() sollte noch ein Wort verloren werden: File-Objekte sind immutable, stehen also immer nur für genau eine Datei. Ändert sich der Dateiname, ist das File-Objekt ungültig, und es ist kein Zugriff mehr über dieses File-Objekt erlaubt. Auch wenn eine Laufzeitumgebung keine Exception auslöst, sind alle folgenden Ergebnisse von Anfragen unsinnig.

15.1.5 Verzeichnisse auflisten und Dateien filtern

Um eine Verzeichnisanzeige oder einen Dateiauswahldialog zu programmieren, benötigen wir eine Liste von Dateien, die in einem Verzeichnis liegen. Ein Verzeichnis kann reine Dateien oder auch wieder Unterverzeichnisse besitzen. Die list()- und listFiles()-Methoden der Klasse File geben ein Feld von Zeichenketten mit Dateien und Verzeichnissen beziehungsweise ein Feld von File-Objekten mit den enthaltenen Elementen zurück.

```
class java.io.File
    implements Serializable, Comparable<File>
```

- File[] listFiles()
- String[] list()
 Gibt eine Liste der Dateien in einem Verzeichnis als File-Array oder String-Array zurück. Das Feld enthält weder »..« noch »..«.

zB

Beispiel

Ein einfacher Directory-Befehl ist leicht mittels einiger Zeilen programmiert:

```
String[] entries = new File( "." ).list();
System.out.println( Arrays.toString(entries) );
```

Die einfache Methode `list()` liefert dabei nur relative Pfade, also einfach den Dateinamen oder den Verzeichnisnamen. Den absoluten Namen zu einer Dateiquelle müssen wir also erst zusammensetzen. Praktischer ist da schon die Methode `listFiles()`, da wir hier komplette `File`-Objekte bekommen, die ihre ganze Pfadangabe schon kennen. Wir können den Pfad mit `getName()` erfragen.

15.1.6 Dateien und Verzeichnisse löschen

Mithilfe der Methode `delete()` auf einem `File`-Objekt lässt sich eine Datei oder ein Verzeichnis entfernen. Diese Methode löscht wirklich! Sie ist nicht so zu verstehen, dass sie `true` liefert, falls die Datei potenziell gelöscht werden kann. Konnte die Laufzeitumgebung `delete()` nicht ausführen, so sollte die Rückgabe `false` sein. Ein zu lösches Verzeichnis muss leer sein, andernfalls kann das Verzeichnis nicht gelöscht werden. Unsere unten stehende Implementierung geht dieses Problem so an, dass sie rekursiv die Unterverzeichnisse löscht.

```
class java.io.File
    implements Serializable, Comparable<File>
```

- `boolean delete()`

Löscht die Datei oder das leere Verzeichnis. Falls die Datei nicht gelöscht werden konnte, gibt es keine Ausnahme, sondern den Rückgabewert `false`.

- `void deleteOnExit()`

Löscht die Datei bzw. das Verzeichnis, wenn die virtuelle Maschine korrekt beendet wird. Einmal vorgeschlagen, kann das Löschen nicht mehr rückgängig gemacht werden. Falls die JVM vorzeitig die Grätsche macht – Reinigungsfachkraft stolpert über Kabel –, kann natürlich die Datei möglicherweise noch immer nicht gelöscht sein, was insbesondere für temporäre Dateien, die über `createTempFile()` angelegt wurden, eventuell lästig wäre.



Hinweis

Auf manchen Systemen liefert `delete()` die Rückgabe `true`, die Datei ist aber nicht gelöscht. Der Grund kann eine noch geöffnete Datei sein, mit der zum Beispiel ein Eingangsstrom verbunden ist und die dadurch gelockt ist.

15.2 Dateien mit wahlfreiem Zugriff

Dateien können auf zwei unterschiedliche Arten gelesen und modifiziert werden: zum einen über einen Datenstrom, der Bytes wie in einem Medien-Stream verarbeitet, zum anderen über *wahlfreien Zugriff* (engl. *random access*). Während der Datenstrom eine strenge Sequenz erzwingt, ist dies beim wahlfreien Zugriff egal, da innerhalb der Datei beliebig hin und her gesprungen werden kann und ein Dateizeiger verwaltet wird, den wir setzen können. Da wir es mit Dateien zu tun haben, heißt das Ganze dann *Random Access File*, und die Klasse, die wahlfreien Zugriff anbietet, ist `java.io.RandomAccessFile`.

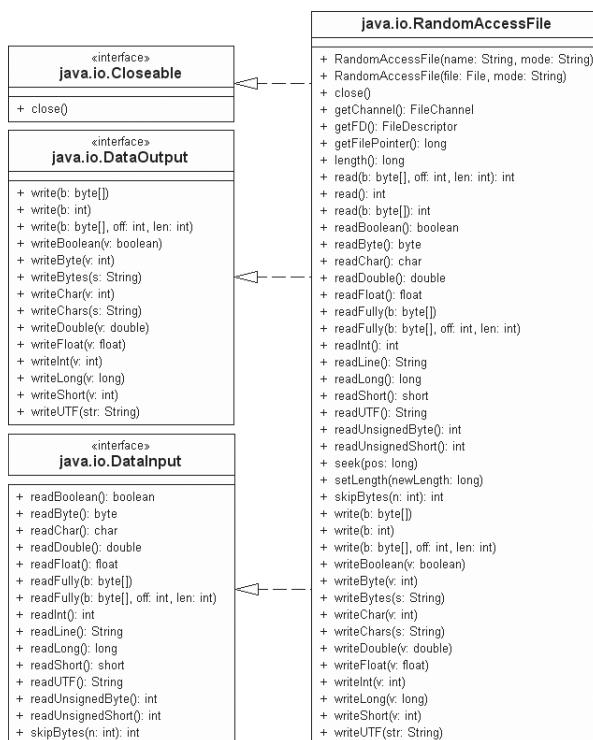


Abbildung 15.2: UML-Diagramm der Klasse RandomAccessFile

15.2.1 Ein RandomAccessFile zum Lesen und Schreiben öffnen

Die Klasse deklariert zwei Konstruktoren, um mit einem Dateinamen oder File-Objekt ein RandomAccessFile-Objekt anzulegen. Im Konstruktor bestimmt der zweite Parameter eine Zeichenkette für den Zugriffsmodus; damit lässt sich eine Datei lesend oder schreibend öffnen. Die Angabe vermeidet Fehler, da eine zum Lesen geöffnete Datei nicht versehentlich überschrieben werden kann.

Modus	Funktion
r	Die Datei wird zum Lesen geöffnet. Wenn sie nicht vorhanden ist, wird ein Fehler ausgelöst. Der Versuch, auf diese Datei schreibend zuzugreifen, wird mit einer Exception bestraft.
rw	Die Datei wird zum Lesen oder Schreiben geöffnet. Eine existierende Datei wird dabei geöffnet, und hinten können die Daten angehängt werden, ohne dass die Datei gelöscht wird. Existiert die Datei nicht, wird sie neu angelegt, und ihre Startgröße ist null. Soll die Datei gelöscht werden, so müssen wir dies ausdrücklich über delete() der File-Klasse selbst tun.

Tabelle 15.1: Zwei Modi für den Konstruktor von RandomAccessFile

Zusätzlich lässt sich bei rw noch ein s oder d anhängen; sie stehen für Möglichkeiten, beim Schreiben die Daten mit dem Dateisystem zu synchronisieren.

```
class java.io.RandomAccessFile
    implements DataOutput, DataInput, Closeable
```

- RandomAccessFile(String name, String mode) throws FileNotFoundException
- RandomAccessFile(File file, String mode) throws FileNotFoundException
Öffnet die Datei. Ob die Datei zum Lesen oder Schreiben vorbereitet ist, bestimmt der String mode mit gültigen Belegungen »r« oder »rw«. Ist der Modus falsch gesetzt, zeigt eine IllegalArgumentException dies an. Löst eine FileNotFoundException³ aus, falls die Datei nicht geöffnet werden kann.
- void close()
Schließt eine geöffnete Datei wieder.

³ Eingedeutscht »DöösIschNetDoo«.

15.2.2 Aus dem RandomAccessFile lesen

Um Daten aus einer mit einem RandomAccessFile verwalteten Datei zu bekommen, nutzen wir eine der `readXXX()`-Methoden. Sie lesen direkt das Byte-Feld aus der Datei oder mehrere Bytes, die zu einem primitiven Datentyp zusammengesetzt sind. `readChar()` etwa liest hintereinander 2 Byte und verknüpft diese zu einem char.

```
class java.io.RandomAccessFile  
    implements DataOutput, DataInput, Closeable
```

- `int read() throws IOException`
Liest genau ein Byte und liefert es als int zurück.
- `int read(byte[] b) throws IOException`
Liest `b.length()` viele Bytes und speichert sie im Feld `b`.
- `int read(byte[] b, int off, int len) throws IOException`
Liest `len` Bytes aus der Datei und schreibt sie in das Feld `b` ab der Position `off`. Wurden mehr als ein, aber weniger als `len` Bytes gelesen, wird die gelesene Größe als Rückgabewert zurückgegeben.
- `final boolean readBoolean() throws IOException`
- `final byte readByte() throws IOException`
- `final short readShort() throws IOException`
- `final int readInt() throws IOException`
- `final long readLong() throws IOException`
- `final char readChar() throws IOException`
- `final double readDouble() throws IOException`
- `final float readFloat() throws IOException`
Liest einen primitiven Datentyp.
- `final int readUnsignedByte() throws IOException`
Liest ein als vorzeichenlos interpretiertes Byte.
- `final int readUnsignedShort() throws IOException`
Liest zwei als vorzeichenlos interpretierte Bytes.
- `final void readFully(byte[] b) throws IOException`
Versucht, den gesamten Puffer `b` zu füllen.
- `final void readFully(byte[] b, int off, int len) throws IOException`
Liest `len` Bytes und speichert sie im Puffer `b` ab dem Index `off`.

Zum Schluss bleiben zwei Methoden, die eine Zeichenkette liefern:

- `final String readLine() throws IOException`
Liest eine Textzeile, die das Zeilenendezeichen `\r` oder `\n` beziehungsweise eine Kombination `\r\n` abschließt. Die letzte Zeile muss nicht so abgeschlossen sein, denn ein Dateiende zählt als Zeilenende. `readLine()` interpretiert die Zeichen nicht als Unicode, sondern übernimmt die Zeichen einfach als ASCII-Bytes. (Ohne die Konvertierung verschiedener Codepages, etwa von einer Datei in einem ungewohnten IBM-Format, liest `readLine()` nicht die korrekten entsprechenden Unicode-Zeilen heraus. Diese Byte-in-Char-Umwandlung müsste manuell vorgenommen werden.) Auch weil `RandomAccessFile` nicht puffert, bietet sich aus Geschwindigkeitsgründen eine zeilenweise Verarbeitung von ASCII-Dateien über `readLine()` nicht an, und die passende Klasse `Scanner` oder `BufferedReader` sollte Verwendung finden.
- `final String readUTF()`
Liest einen modifizierten UTF-kodierten String und gibt einen Unicode-String zurück. Ein UTF-String fasst entweder 1, 2 oder 3 Byte zu einem Unicode-Zeichen zusammen. Der übernächste Abschnitt erklärt die Kodierung genauer.

Rückgabe -1 und EOFException *

Die Methoden liefern nicht alle einen Fehler, wenn die Datei schon fertig abgearbeitet wurde und keine Daten mehr anliegen. Im Fall von `int read()`, `int read(byte[])` oder `int read(byte[], int, int)` gibt es einfach den Rückgabewert `-1` und keine Exception. Ähnliches gilt für `readLine()`. Die Methode liefert `null` am Dateiende. Für die anderen Lese-Methoden gilt, dass sie eine bestimmte Anzahl Bytes erzwingen, etwa `readLong()` 8 – oder auch nur 1 Byte für `readByte()` –, sodass im Fall eines Dateiendes eine `EOFException` folgt. Bis auf wenige Ausnahmen gibt es kaum weitere Einsatzgebiete von `EOFException` in der Java-Bibliothek.

15.2.3 Schreiben mit RandomAccessFile

Da `RandomAccessFile` die Schnittstellen `DataOutput` und `DataInput` implementiert, werden zum einen die `readXXX()`-Methoden wie bisher vorgestellt implementiert und zum anderen eine Reihe von Schreibmethoden der Form `writeXXX()`. Diese sind analog zu den Lesemethoden:

- `write(byte[] b)`
- `write(int b)`
- `write(byte[] b, int off, int len)`

- `writeBoolean(boolean v)`
- `writeByte(int v)`
- `writeBytes(String s)`
- `writeChar(int v)`
- `writeChars(String s)`
- `writeDouble(double v)`
- `writeFloat(float v)`
- `writeInt(int v)`
- `writeLong(long v)`
- `writeShort(int v)`
- `writeUTF(String str)`

Der Rückgabetyp ist `void`, und die Methoden können eine `IOException` auslösen.

15.2.4 Die Länge des RandomAccessFile

Mit zwei Methoden greifen wir auf die Länge der Datei zu: einmal schreibend (verändernd) und einmal lesend.

```
class java.io.RandomAccessFile
    implements DataOutput, DataInput, Closeable
```

- `void setLength(long newLength) throws IOException`
Setzt die Größe der Datei auf `newLength`. Ist die Datei kleiner als `newLength`, wird sie mit unbestimmten Daten vergrößert; wenn die Datei größer war als die zu setzende Länge, wird die Datei abgeschnitten. Dies bedeutet, dass der Dateiinhalt mit `setLength(0)` leicht zu löschen ist.
- `long length() throws IOException`
Liefert die Länge der Datei. Schreibzugriffe erhöhen den Wert, und `setLength()` modifiziert ebenfalls die Länge.

15.2.5 Hin und her in der Datei

Die bisherigen Lesemethoden setzen den Datenzeiger automatisch eine Position weiter. Wir können den Datenzeiger jedoch auch manuell an eine selbst gewählte Stelle setzen und damit durch die Datei navigieren.

zB

Beispiel

Erzeuge eine Datei, und setze an die Stelle 1.000 das Byte 0xFF:

Listing 15.1: com/tutego/insel/io/raf/CreateBigFile.java, main()

```
RandomAccessFile file = new RandomAccessFile("c:/test.bin", "rw" );
file.seek( 1000 );
file.write( -1 );
file.close();
```

Da skipBytes() den Dateizeiger nicht »hinter« die Datei stellen kann, funktioniert die Lösung nur mit seek().

Die nachfolgenden Lese- oder Schreibzugriffe setzen dann dort an. Die im Folgenden beschriebenen Methoden haben etwas mit diesem Dateizeiger und seiner Position zu tun:

```
class java.io.RandomAccessFile
    implements DataOutput, DataInput, Closeable
```

- long getFilePointer() throws IOException

Liefert die momentane Position des Dateizeigers. Das erste Byte steht an der Stelle null.

- void seek(long pos) throws IOException

Setzt die Position des Dateizeigers auf pos. Diese Angabe ist absolut und kann daher nicht negativ sein. Falls doch, wird eine Ausnahme ausgelöst. file.seek(file.length()); setzt den Zeiger auf das Ende der Datei.

- int skipBytes(int n) throws IOException

Im Gegensatz zu seek() positioniert skipBytes() relativ. n ist die Anzahl, um die der Dateizeiger bewegt wird. Ist n negativ, werden keine Bytes übersprungen. Eine relative Positionierung mit positivem und negativem n für ein RandomAccessFile raf erreicht raf.seek(raf.getFilePointer() + n). Die Summe darf aber nicht negativ sein, sonst gibt es von seek() eine IOException. Die Rückgabe gibt die tatsächlich gesprungenen Bytes zurück, was nicht mit n identisch sein muss!

Setzt seek() den Zeiger weiter, als es möglich ist, wird die Datei dadurch nicht automatisch größer. Sie verändert jedoch ihre Größe, wenn Daten geschrieben werden.

15.3 Dateisysteme unter NIO.2

Die bisher vorgestellten Konzepte gibt es im Wesentlichen schon seit den Urzeiten von Java, also seit Java 1.0. In den letzten Jahren ist rund um die `File`-Klasse wenig passiert. Doch Entwickler quälten sich immer wieder mit ganz zentralen Fragen, die die bisherigen Implementierungen nicht wirklich lösten:

- Wie lässt sich eine Datei einfach und schnell kopieren?
- Wie lässt sich eine Datei verschieben, wobei die Semantik auf unterschiedlichen Plattformen immer gleich ist.
- Wie lässt sich auf eine Änderung im Dateisystem reagieren, sodass ein Callback uns informiert, dass sich eine Datei verändert hat?
- Wie lässt sich einfach ein Verzeichnis rekursiv ablaufen?
- Wie lässt sich eine symbolische Verknüpfung anlegen und verfolgen?
- Wie lässt sich realisieren, dass die `File`-Operationen abstrahiert werden und nicht nur auf dem lokalen Dateisystem basieren? Wünschenswert ist eine Abstraktion, so dass die gleiche API auch ein virtuelles Dateisystem im Hauptspeicher, entfernte Dateisysteme wie FTP oder ein Repository anspricht.

Diese Probleme wurden für Java 7 angegangen und in der JSR-203, »More New I/O APIs for the JavaTM Platform ("NIO.2")«, spezifiziert. Die JSR began schon 2003, und so waren die Erwartungen der Java-Community groß, dass sie nicht so lange warten müssten. Aber erst in Java 7 kam es zum großen Wurf. Das macht die »alte« `File`-Klasse eigentlich überflüssig, aber vermutlich scheut sich Oracle davor, ein `@Deprecated` an die Klasse zu setzen, denn sonst würden plötzlich riesige Mengen Quellcode in vielen Programmen markiert.

15.3.1 FileSystem und Path

Im Zentrum der in Java 7 und NIO.2 eingeführten neuen Klassen stehen `FileSystem` und `Path`. Die neuen Typen befinden sich im Gegensatz zu `File`, das im `java.io`-Paket liegt, im Paket `java.nio.file`. Es gibt zwar einige Überlappungen, doch NIO.2 ist mehr oder weniger komplett vom »alten« Modell getrennt:

- `FileSystem` beschreibt ein Datensystem und ist eine abstrakte Klasse. Es wird von konkreten Dateisystemen, wie dem lokalen Dateisystem oder einem Zip-Archiv, realisiert. Um an das aktuelle Dateisystem zu kommen, deklariert die Klasse `FileSystems` eine statische Methode: `FileSystems.getDefault()`.

- Path repräsentiert einen Pfad zu einer Datei oder einem Verzeichnis, wobei die Pfadangaben relativ oder absolut sein können. Die Methoden erinnern ein wenig an File, doch der große Unterschied ist, dass File selbst die Datei oder das Verzeichnis repräsentiert und Anfragemethoden wie `isDirectory()` oder `lastModified()` deklariert, während Path nur den Pfad repräsentiert und nur pfad-bezogene Methoden anbietet. Modifikationsmethoden gehören nicht dazu; dazu dienen extra Typen wie `BasicFileAttributes` für Attribute.

Ein Path-Objekt aufbauen

Ein Path-Objekt lässt sich nicht wie File über einen Konstruktor aufbauen, da die Klasse abstrakt ist. File und Path haben aber dennoch einiges gemeinsam, etwa dass sie immutable sind. Das FileSystem-Objekt bietet die entsprechende Methode `getPath()`, und ein FileSystem wird über eine Fabrikmethode von `FileSystems` erfragt.

Beispiel

Baue ein Path-Objekt auf:

```
FileSystem fs = FileSystems.getDefault();
Path p = fs.getPath( "C:/Windows/Fonts/" );
```

zB

15

Da der Ausdruck `FileSystems.getDefault().getPath()` etwas unhandlich ist, existiert die Methode `get()` in der Utility-Klasse `Paths`. Auch aus einem File-Objekt lässt sich mit `toPath()` ein Path ableiten, was bedeutet, dass Oracle für Java 7 noch einmal die File-Klasse angefasst und auf die neuen NIO.2-Klassen angepasst hat. Wir werden die Vereinfachung mit `Paths.get()` im Folgenden nutzen.

```
final class java.nio.file.Paths
```

- static Path get(String first, String... more)
Erzeuge einen Pfad aus Segmenten. Wenn etwa »\« der Separator ist, dann ist `Paths.get("a", "b", "c")` gleich `Paths.get("a\\b\\c")`.
- static Path get(URI uri)
Erzeugt einen Pfad aus einer URI.

Jedes Path-Objekt hat auch eine Methode `getFileSystem()`, um wieder an das FileSystem zu kommen.

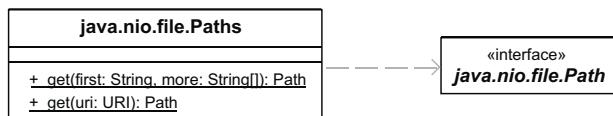


Abbildung 15.3: Abhängigkeiten der Klassen Paths und Path

15.3.2 Die Utility-Klasse Files

Da die Klasse Path nur Pfade, aber keine Dateiinformationen wie die Länge oder Änderungszeit repräsentiert und Path auch keine Möglichkeit bietet, Dateien anzulegen und zu löschen, übernimmt die Klasse Files diese Aufgaben.

Einfaches Einlesen und Schreiben von Dateien

Nach 15 Jahren des Wartens gibt es nun auch Methoden, die den Dateiinhalt einlesen oder Strings bzw. ein Byte-Feld schreiben.

Listing 15.2: com/tutego/insel/nio2/ListAllLines.java

```

package com.tutego.insel.nio2;

import java.io.IOException;
import java.net.*;
import java.nio.charset.Charset;
import java.nio.file.*;

public class ListAllLines
{
    public static void main( String[] args ) throws IOException, URISyntaxException
    {
        URI uri = ListAllLines.class.getResource( "/lyrics.txt" ).toURI();
        Path path = Paths.get( uri );
        System.out.printf( "Datei '%s' mit Länge %d Byte(s) hat folgendes Zeilen:%n",
                           path.getFileName(), Files.size( path ) );
        int lineCnt = 1;
        for ( String line : Files.readAllLines( path, StandardCharsets.UTF_8 ) )
            System.out.println( lineCnt++ + ": " + line );
    }
}
  
```

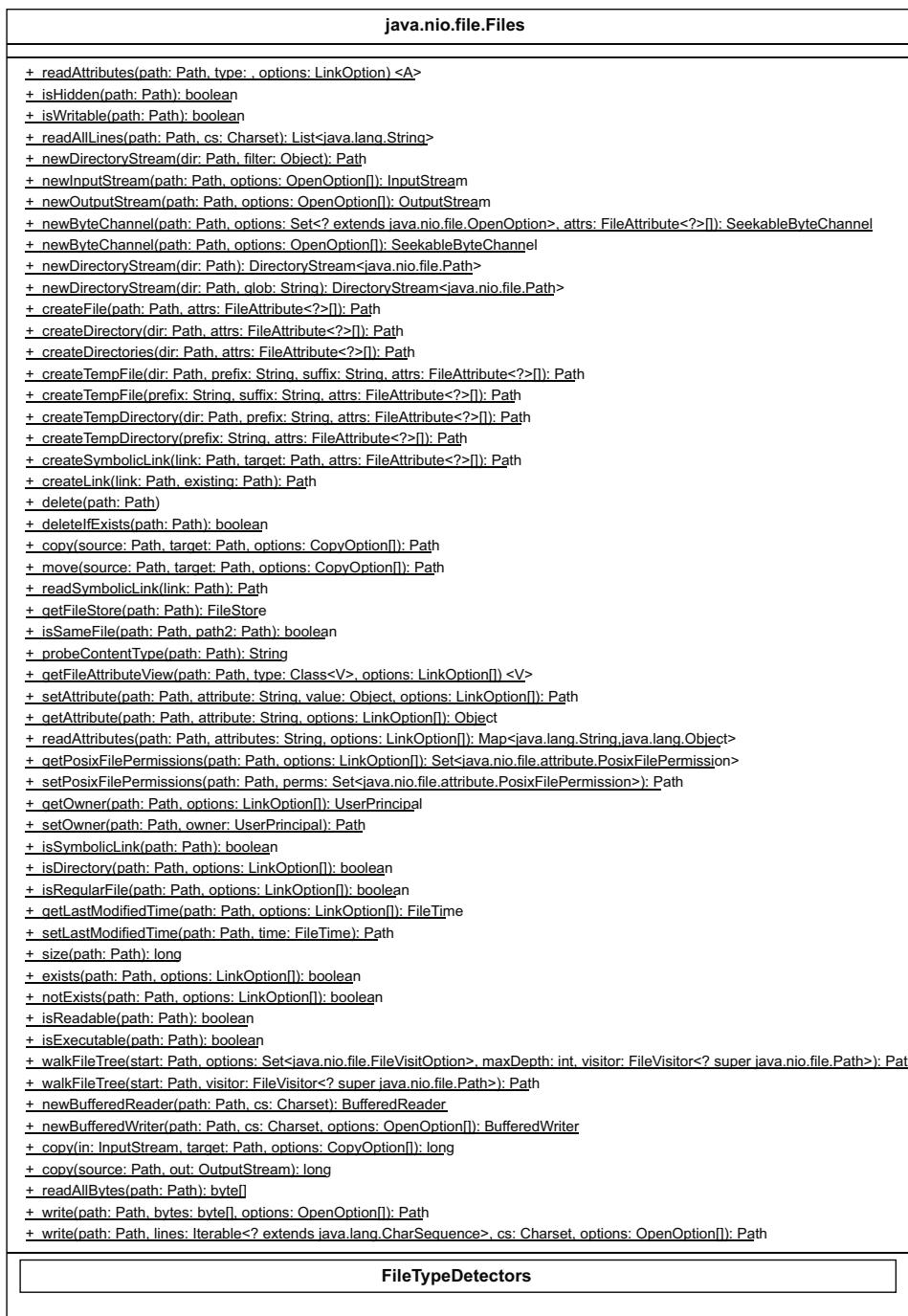


Abbildung 15.4: UML-Diagramm von Files



Hinweis

Auch wenn es naheliegt, die `Files`-Methode zum Einlesen mit einem `Path`-Objekt zu füttern, das ein HTTP-URI repräsentiert, funktioniert dies nicht. So liefert schon die erste Zeile des Programms eine Ausnahme des Typs »`java.nio.file.FileSystemNotFoundException`: Provider "http" not installed«.

```
Path path = Paths.get( new URI( "http://tutego.de/aufgaben/bond.txt" ) ); // ☹
String content = new String( Files.readAllBytes( path ),
                            StandardCharsets.UTF_8 );
System.out.println( content );
```

Vielleicht kommt in der Zukunft ein Standard-Provider von Oracle, doch es ist davon auszugehen, dass quelloffene Lösungen diese Lücke schließen werden. Schwer zu programmieren sind Dateisystem-Provider nämlich nicht.

```
final class java.nio.file.Files
```

- static long size(Path path) throws IOException
- static byte[] readAllBytes(Path path) throws IOException
- static List<String> readAllLines(Path path, Charset cs) throws IOException
- static Path write(Path path, byte[] bytes, OpenOption... options) throws IOException
- static Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options) throws IOException

15.4 Stream-Klassen und Reader/Writer am Beispiel von Dateien

Unterschiedliche Klassen zum Lesen und Schreiben von Binär- und Zeichendaten sammeln sich im Paket `java.io`. Für die byte-orientierte Verarbeitung, etwa von PDF- oder MP3-Dateien, gibt es andere Klassen als für Textdokumente, zum Beispiel HTML oder Konfigurationsdateien. Binär- von Zeichendaten zu trennen ist sinnvoll, da zum Beispiel beim Einlesen von Textdateien diese immer in Unicode konvertiert werden müssen, da Java intern alle Zeichen in Unicode kodiert.

Die vier Basisklassen sind

- die zeichenorientierten Klassen `Reader`, `Writer` und
- die byte-orientierten Klassen `InputStream` und `OutputStream`.

FileInputStream, FileReader, FileOutputStream, FileWriter

Dieser Abschnitt stellt die vier Klassen zum Lesen und Schreiben aus Dateien vor, und zwar jeweils die zeichen- und byte-orientierten Klassen.

	Bytes (oder Byte-Arrays)	Zeichen (oder Zeichen-Arrays, Strings)
Aus Dateien lesen	<code>InputStream</code>	<code>Reader</code>
In Dateien schreiben	<code>OutputStream</code>	<code>Writer</code>

Tabelle 15.2: Lese- und Schreibklassen für Dateien

Hinweis

Lies den ganzen Dateiinhalt in ein Byte-Feld:

```
File f = new File( dateiname );
byte[] buffer = new byte[ (int) f.length() ];
InputStream in = new FileInputStream( f );
in.read( buffer );
in.close();
```

Sinnvoller als das gesamte Einlesen ist aber im Allgemeinen das Lesen in Blöcken. Eine korrekte Fehlerbehandlung ist immer notwendig, wird aber hier erst einmal ausgeblendet.

15

15.4.1 Mit dem `FileWriter` Texte in Dateien schreiben

Der `FileWriter` ist ein spezieller `Writer`, der Ausgaben in eine Datei erlaubt.

Das folgende Programm erstellt die Datei `fileWriter.txt` und schreibt eine Textzeile mit Zeilenvorschubzeichen hinein:

Listing 15.3: com/tutego/insel/io/stream/FileWriterDemo.java, main()

```

Writer fw = null;

try
{
    fw = new FileWriter( "fileWriter.txt" );
    fw.write( "Zwei Jäger treffen sich..." );
    fw.append( System.getProperty("line.separator") ); // e.g. "\n"
}
catch ( IOException e ) {
    System.err.println( "Konnte Datei nicht erstellen" );
}
finally {
    if ( fw != null )
        try { fw.close(); } catch ( IOException e ) { e.printStackTrace(); }
}

```

Da der Konstruktor und die `write()/append()`-Methoden eine `IOException` in dem Fall auslösen, wenn ein Öffnen beziehungsweise Schreiben nicht möglich ist, müssen wir einen `try`-Block um die Anweisungen setzen oder mit `throws` den Fehler nach oben weitergeben.

```

class java.io.FileWriter
extends OutputStreamWriter

```

- `FileWriter(File file) throws IOException`
- `FileWriter(String filename) throws IOException`
- `FileWriter(File file, boolean append) throws IOException`
- `FileWriter(String filename, boolean append) throws IOException`
Erzeugt einen Ausgabestrom und hängt die Daten an eine existierende Datei an, wenn `append` gleich `true` ist. Eine weitere Möglichkeit, Daten hinten anzuhängen, bietet die Klasse `RandomAccessFile` oder `FileOutputStream`.
- `FileWriter(FileDescriptor fd)`
Erzeugt einen Ausgabestrom zum Schreiben in eine Datei. Existiert die Datei bereits, deren Namen wir übergeben, wird die Datei gelöscht.

Auf den ersten Blick scheinen der Klasse `FileWriter` die versprochenen `write()`-Methoden zu fehlen. Fakt ist aber, dass diese von `OutputStreamWriter` geerbt werden, und die Klasse erbt und überschreibt wiederum die Methoden aus `Writer`. Mit den Oberklassen verfügt der `FileWriter` insgesamt über folgende Methoden, deren Ausnahme `IOException` hier nicht genannt ist:

- `Writer append(char c)`
- `Writer append(CharSequence csq)`
- `Writer append(CharSequence csq, int start, int end)`
- `void write(int c)`
- `void write(String str)`
- `void write(String str, int off, int len)`
- `void write(char[] cbuf)`
- `void write(char[] cbuf, int off, int len)`
- `void close()`
- `void flush()`
- `String getEncoding()`

Bis auf `getEncoding()` lösen alle verbleibenden Methoden im Fehlerfall eine `IOException` aus, die als geprüfte Ausnahme behandelt werden muss. Die Methoden stellt Abschnitt 15.5.7, »Die abstrakte Basisklasse `Writer`«, genauer vor.

15.4.2 Zeichen mit der Klasse `FileReader` lesen

Der `FileReader` liest aus Dateien entweder einzelne Zeichen, Strings oder Zeichenfelder. Wie beim `Writer` deklariert die Klasse Konstruktoren zur Annahme des Dateinamens. So zeigt folgendes Beispiel eine Anwendung der `FileReader`-Klasse:

Listing 15.4: com/tutego/insel/io/stream/FileReaderDemo.java, main()

```
Reader reader = null;
try
{
    reader = new FileReader( "bin/lyrics.txt" );

    for ( int c; ( c = reader.read() ) != -1; )
        System.out.print( (char) c );
}
```

```

    catch ( IOException e ) {
        System.err.println( "Fehler beim Lesen der Datei!" );
    }
    finally {
        try { reader.close(); } catch ( Exception e ) { e.printStackTrace(); }
    }
}

```

```

class java.io.FileReader
extends InputStreamReader

```

- public FileReader(String fileName) throws FileNotFoundException
Öffnet die Datei über einen Dateinamen zum Lesen. Falls sie nicht vorhanden ist, löst der Konstruktor eine FileNotFoundException aus.
- public FileReader(File file) throws FileNotFoundException
Öffnet die Datei zum Lesen über ein File-Objekt. Falls sie nicht verfügbar ist, löst der Konstruktor eine FileNotFoundException aus.
- public FileReader(FileDescriptor fd)
Nutzt die schon vorhandene offene Datei über ein FileDescriptor-Objekt.

Die Methoden zum Lesen stammen aus den Oberklassen InputStreamReader und Reader. Aus InputStreamReader kommen int read(), int read(char[], int, int), close(), getEncoding() und ready(). Da InputStreamReader wiederum Reader erweitert, kommen die Methoden int read(char[]), int read(CharBuffer), mark(int), markSupported(), reset(), skip(long) hinzu. Abschnitt 15.5.8, »Die abstrakte Basisklasse Reader«, beschreibt die Methoden genauer.

15.4.3 Kopieren mit FileOutputStream und FileInputStream

Die Klasse FileOutputStream bietet grundlegende Methoden, um in Dateien zu schreiben. FileOutputStream implementiert alle nötigen Methoden, die java.io.OutputStream vorschreibt, also etwa write(int), write(byte[]).

```

class java.io.FileOutputStream
extends OutputStream

```

- FileOutputStream(String name) throws FileNotFoundException
Erzeugt einen FileOutputStream mit einem gegebenen Dateinamen.

- `FileOutputStream(File file) throws FileNotFoundException`
Erzeugt einen `FileOutputStream` aus einem `File`-Objekt.
- `FileOutputStream(String name, boolean append) throws FileNotFoundException`
Wie `FileOutputStream(name)`, hängt jedoch bei `append=true` Daten an.
- `FileOutputStream(File file, boolean append) throws FileNotFoundException`
Wie `FileOutputStream(file)`, hängt jedoch bei `append=true` Daten an.
- `FileOutputStream(FileDescriptor fdObj)`
Erzeugt einen `FileOutputStream` aus einem `FileDescriptor`-Objekt.

Ist der Parameter `append` nicht mit `true` belegt, wird der alte Inhalt überschrieben. Die `FileNotFoundException` wirkt vielleicht etwas komisch, wird aber dann ausgelöst, wenn zum Beispiel die Dateiangabe ein Verzeichnis repräsentiert oder die Datei gelockt ist.

`FileInputStream` ist der Gegenspieler und dient zum Lesen der Binärdaten. Um ein Objekt anzulegen, haben wir die Auswahl zwischen drei Konstruktoren. Sie binden eine Datei (etwa repräsentiert als ein Objekt vom Typ `File`) an einen Datenstrom.

```
class java.io.FileInputStream
extends InputStream
```

- `FileInputStream(String name) throws FileNotFoundException`
Erzeugt einen `FileInputStream` mit einem gegebenen Dateinamen.
- `FileInputStream(File file) throws FileNotFoundException`
Erzeugt `FileInputStream` aus einem `File`-Objekt.
- `FileInputStream(FileDescriptor fdObj)`
Erzeugt `FileInputStream` aus einem `FileDescriptor`-Objekt.

Der `FileInputStream` ist ein spezieller `InputStream` und besitzt daher Methoden wie `int read()`, `int read(byte[])` zum Lesen.

Zur Veranschaulichung dient die folgende Grafik.

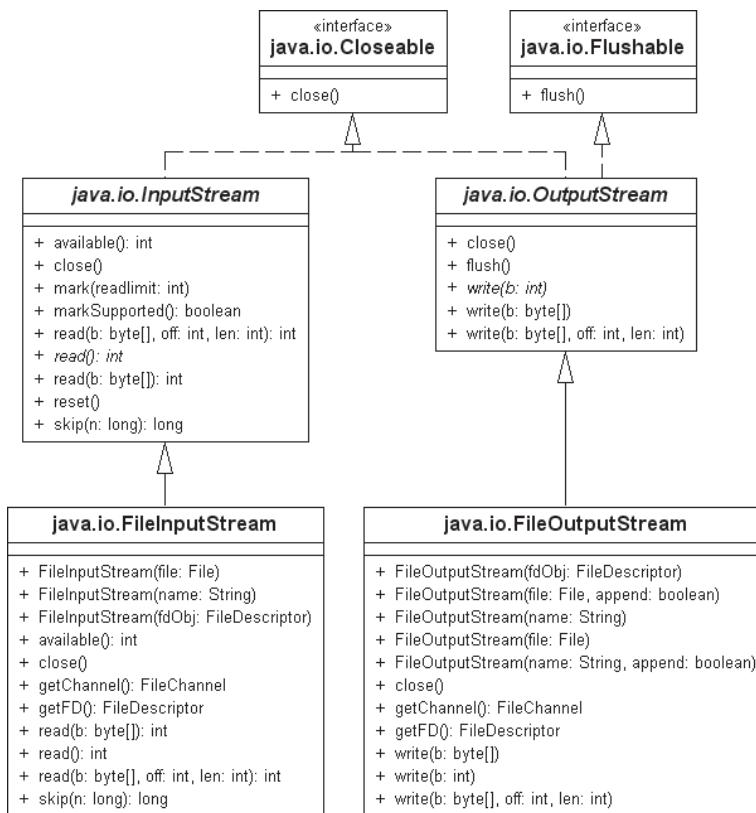


Abbildung 15.5: UML-Diagramm der Klassen FileInputStream und FileOutputStream

15.4.4 Datenströme über Files mit NIO.2 beziehen

Anstatt einen `FileInputStream` oder einen `FileOutputStream` mit einem Dateinamen oder `File`-Objekt anzulegen, bietet die `Files`-Klasse zwei Methoden, die direkt den Eingabe-/Ausgabestrom liefern:

```
final abstract java.nio.file.Files
```

- `static OutputStream newOutputStream(Path path, OpenOption... options)`
Legt eine Datei an und liefert den Ausgabestrom auf die Datei.
- `static InputStream newInputStream(Path path, OpenOption... options)`
Öffnet die Datei und liefert einen Eingabestrom zum Lesen.

Da die `OpenOption` ein Vararg ist und somit weggelassen werden können, ist der Programmcode kurz. (Er wäre noch kürzer ohne die korrekte Fehlerbehandlung...)

Beispiel: Eine kleine PPM-Grafikdatei schreiben

Das PPM-Format ist ein einfaches Grafikformat. Es beginnt mit einem Identifizierer, dann folgenden die Ausmaße und schließlich die ARGB-Werte für die Pixelfarben.

Listing 15.5: com/tutego/insel/nio2/WriteTinyPPM.java, main()

```
try ( OutputStream out = Files.newOutputStream( Paths.get( "littlepic.ppm" ) ) )
{
    out.write( "P3 1 1 255 255 0 0".getBytes() );
}
catch ( IOException e )
{
    e.printStackTrace();
}
```

Falls die Datei nicht existiert, wird sie überschrieben; existiert sie nicht, wird sie neu angelegt. Diese Standardoption ist aber ein wenig zu einschränkend, und daher beschreibt `OpenOption` Zusatzoptionen. `OpenOption` ist eine Schnittstelle, die von den Aufzählungen `LinkOption`, `StandardOpenOption` realisiert wird.

OpenOption	Beschreibung
java.nio.file.StandardOpenOption	
READ	Öffnen für Lesezugriff
WRITE	Öffnen für Schreibzugriff
APPEND	Neue Daten kommen an das Ende. Atomar bei parallelen Schreiboperationen.
TRUNCATE_EXISTING	Für Schreiber: Existiert die Datei, wird die Länge vorher auf 0 gesetzt.
CREATE	Legt Datei an, falls sie noch nicht existiert.
CREATE_NEW	Legt Datei nur an, falls sie vorher noch nicht existierte.
DELETE_ON_CLOSE	Die Java-Bibliothek versucht, die Datei zu löschen, wenn sie geschlossen wird.
SPARSE	Hinweis für das Dateisystem, die Datei kompakt zu speichern, da sie aus vielen Null-Bytes besteht
SYNC	Jeder Schreibzugriff und jedes Update der Metadaten soll sofort zum Dateisystem.

Tabelle 15.3: Konstanten aus `StandardOpenOption` und `LinkOption`

OpenOption	Beschreibung
DSYNC	Jeder Schreibzugriff soll sofort zum Dateisystem.
java.nio.file.LinkOption	
NOFOLLOW_LINKS	Symbolischen Links wird nicht gefolgt.

Tabelle 15.3: Konstanten aus StandardOpenOption und LinkOption (Forts.)

Die Option CREATE_NEW kann nur funktionieren, wenn die Datei noch nicht vorhanden ist. Das zeigt anschaulich das folgende Beispiel:

Listing 15.6: com/tutego/insel/nio2/StandardOpenOptionCreateNewDemo.java, main()

```
Files.deleteIfExists( Paths.get( "opa.herbert.tmp" ) );
Files.newOutputStream( Paths.get( "opa.herbert.tmp" ) ).close();
Files.newOutputStream( Paths.get( "opa.herbert.tmp" ) ).close();
Files.newOutputStream( Paths.get( "opa.herbert.tmp" ),
    StandardOpenOption.CREATE_NEW ).close();
```

Hier führt die letzte Zeile zu einer »java.nio.file.FileAlreadyExistsException: opa.herbert.tmp«.

Die Option DELETE_ON_CLOSE ist für temporäre Dateien nützlich. Das folgende Beispiel verdeutlicht die Arbeitsweise:

Listing 15.7: com/tutego/insel/nio2/StandardOpenOptionDeleteOnCloseDemo.java, main()

```
Path path = Paths.get( "opa.herbert.tmp" );

Files.deleteIfExists( path );
System.out.println( Files.exists( path ) ); // false

Files.newOutputStream( path ).close();
System.out.println( Files.exists( path ) ); // true

Files.newOutputStream( path, StandardOpenOption.DELETE_ON_CLOSE,
    StandardOpenOption.SYNC ).close();
System.out.println( Files.exists( path ) ); // false
```

Im letzten Fall wird die Datei angelegt, ein Datenstrom geholt und gleich wieder geschlossen. Wegen StandardOpenOption.DELETE_ON_CLOSE wird Java die Datei von sich aus löschen, was Files.exists() belegt.

15.5 Basisklassen für die Ein-/Ausgabe

Die Strom-Klassen aus dem `java.io`-Paket sind um drei zentrale Prinzipien aufgebaut:

1. Es gibt abstrakte Basisklassen, die Operationen für die Ein-/Ausgabe vorschreiben.
2. Die abstrakten Basisklassen gibt es einmal für Unicode-Zeichen und einmal für Bytes.
3. Die Implementierungen der abstrakten Basisklassen realisieren entweder die konkrete Ein-/Ausgabe in eine bestimmte Ressource (etwa eine Datei oder auch ein Bytefeld) oder sind Filter.

15.5.1 Die abstrakten Basisklassen

Die konkreten Eingabe/Ausgabe-Klassen wie `FileInputStream`, `FileOutputStream`, `FileWriter` oder `BufferedWriter` erweitern abstrakte Oberklassen. Im Allgemeinen können wir vier Kategorien bilden: Klassen zur Ein-/Ausgabe von Bytes (oder Byte-Arrays) und Klassen zur Ein-/Ausgabe von Unicode-Zeichen (Arrays oder Strings).

Basisklasse für	Bytes (oder Byte-Arrays)	Zeichen (oder Zeichen-Arrays)
Eingabe	<code>InputStream</code>	<code>Reader</code>
Ausgabe	<code>OutputStream</code>	<code>Writer</code>

Tabelle 15.4: Basisklassen für Ein- und Ausgabe

Die Klassen `InputStream` und `OutputStream` bilden die Basisklassen für alle byte-orientierten Klassen und dienen somit als Bindeglied bei Methoden, die als Parameter ein Eingabe- und Ausgabe-Objekt verlangen. So ist ein `InputStream` nicht nur für Dateien denkbar, sondern auch für Daten, die über das Netzwerk kommen. Das Gleiche gilt für `Reader` und `Writer`; sie sind die abstrakten Basisklassen zum Lesen und Schreiben von Unicode-Zeichen und Unicode-Zeichenfolgen. Die Basisklassen geben abstrakte `read()`- oder `write()`-Methoden vor, die Unterklassen überschreiben, da nur sie wissen, wie etwas tatsächlich gelesen oder geschrieben wird.

15.5.2 Übersicht über Ein-/Ausgabeklassen

Während die abstrakten Basisklassen von keiner konkreten Datenquelle lesen oder in keine Ressource schreiben, implementieren die Unterklassen eine ganz bestimmte Strategie für eine Ressource. So weiß zum Beispiel ein `FileWriter`, wie für Dateien die ab-

strakte Klasse `Writer` zu implementieren ist, also wie Unicode-Zeichen in eine Datei geschrieben werden.

Die folgenden Tabellen vermitteln einen Überblick über die wichtigsten Unterklassen von `InputStream/OutputStream` und `Reader/Writer`. Die erste Tabelle listet die Eingabeklassen auf – und stellt die byte-orientierten und zeichenorientierten Klassen gegenüber –, und die zweite Tabelle zeigt die wesentlichen Ausgabeklassen.

Byte-Stream-Klasse für die Eingabe	Zeichen-Stream-Klasse für die Eingabe	Beschreibung
<code>InputStream</code>	<code>Reader</code>	Abstrakte Klasse für Zeicheneingabe und Byte-Arrays
<code>BufferedInputStream</code>	<code>BufferedReader</code>	Puffert die Eingabe.
<code>LineNumberInputStream</code> [†]	<code>LineNumberReader</code>	Merkt sich Zeilennummern beim Lesen.
<code>ByteArrayInputStream</code>	<code>CharArrayReader</code>	Liest Zeichen-Arrays oder Byte-Arrays.
(keine Entsprechung)	<code>InputStreamReader</code>	Wandelt einen Byte-Stream in einen Zeichen-Stream um. Diese Klasse ist das Bindeglied zwischen Byte und Zeichen.
<code>DataInputStream</code>	(keine Entsprechung)	Liest Primitive und auch UTF-8.
<code>FilterInputStream</code>	<code>FilterReader</code>	Abstrakte Klasse für gefilterte Eingabe
<code>PushbackInputStream</code>	<code>PushbackReader</code>	Erlaubt, gelesene Zeichen wieder in den Stream zu geben.
<code>PipedInputStream</code>	<code>PipedReader</code>	Liest von einem <code>PipedWriter</code> oder <code>PipedOutputStream</code> .
<code>StringBufferInputStream</code> [†]	<code>StringReader</code>	Liest aus Strings.
<code>SequenceInputStream</code>	(keine Entsprechung)	Verbindet mehrere <code>InputStreams</code> .
<code>TelepathicInputStream</code>	<code>TelepathicWriter</code>	Überträgt Daten mittels Telepathie. ⁴

Tabelle 15.5: Wichtige Eingabeklassen. Die mit [†] markierten Klassen sind veraltet (deprecated)

⁴ Noch in der Entwicklung.

Byte-Stream-Klasse für die Ausgabe	Zeichen-Stream-Klasse für die Ausgabe	Beschreibung
OutputStream	Writer	Abstrakte Klasse für Zeichenausgabe oder Byte-Ausgabe
BufferedOutputStream	BufferedWriter	Ausgabe des Puffers. Nutzt passendes Zeilenendezeichen.
ByteArrayOutputStream	CharArrayWriter	Schreibt in Arrays.
DataOutputStream	(keine Entsprechung)	Schreibt Primitive und auch UTF-8.
(keine Entsprechung)	OutputStreamWriter	Übersetzt Zeichen-Streams in Byte-Streams.
FileOutputStream	FileWriter	Schreibt in eine Datei.
PrintStream	PrintWriter	Konvertiert primitive Datentypen in Strings und schreibt sie in einen Ausgabestrom.
PipedOutputStream	PipedWriter	Schreibt in eine Pipe.
(keine Entsprechung)	StringWriter	Schreibt in einen String.

Tabelle 15.6: Wichtige Ausgabeklassen

15

Die beiden vorangehenden Tabellen sind nach Eingabe- und Ausgabeklassen segmentiert. Die Klassen lassen sich aber auch anders sortieren, etwa nach der Ressource:

Ressource	Zeichenorientierte Klasse	Byte-orientierte Klasse
Datei	FileReader FileWriter	FileInputStream FileOutputStream
Speicher	CharArrayReader CharArrayWriter StringReader StringWriter	ByteArrayInputStream ByteArrayOutputStream – –
Pipe	PipeReader PipeWriter	PipeInputStream PipeOutputStream

Tabelle 15.7: Ein-/Ausgabeklassen nach Ressourcenzugehörigkeit

15.5.3 Die abstrakte Basisklasse OutputStream

Der Clou bei allen Datenströmen ist nun, dass spezielle Unterklassen wissen, wie sie genau die vorgeschriebene Funktionalität implementieren. Wenn wir uns den OutputStream anschauen, dann sehen wir auf den ersten Blick, dass hier alle wesentlichen Operationen um das Schreiben versammelt sind. Das heißt, dass ein konkreter Stream, der in Dateien schreibt, nun weiß, wie er Bytes in Dateien schreiben wird. Java ist auf der unteren Ebene mit seiner Plattformunabhängigkeit am Ende, und native Methoden schreiben die Bytes.

```
abstract class java.io.OutputStream
    implements Closeable, Flushable
```

- abstract void write(int b) throws IOException
Schreibt ein einzelnes Byte in den Datenstrom.
- void write(byte[] b) throws IOException
Schreibt die Bytes aus dem Array in den Strom.
- void write(byte[] b, int off, int len) throws IOException
Schreibt Teile des Byte-Feldes, nämlich len Byte ab der Position off, in den Ausgabestrom.
- void close() throws IOException
Schließt den Datenstrom. Einzige Methode aus Closeable.
- void flush() throws IOException
Schreibt noch im Puffer gehaltene Daten. Einzige Methode aus der Schnittstelle Flushable.

Die IOException ist keine RuntimeException, muss also behandelt werden.

Zwei Eigenschaften lassen sich an den Methoden ablesen: zum einen, dass nur Bytes geschrieben werden, und zum anderen, dass nicht wirklich alle Methoden abstract sind. Nicht alle diese Methoden sind wirklich elementar, müssen also nicht von allen Ausgabeströmen überschrieben werden. Wir entdecken, dass nur write(int) abstrakt ist. Das würde aber bedeuten, dass alle anderen Methoden konkret wären. Gleichzeitig stellt sich die Frage, wie ein OutputStream, der die Eigenschaften für alle erdenklichen Ausgabeströme vorschreibt, denn wissen kann, wie ein spezieller Ausgabestrom etwa geschlossen (close()) wird oder seine gepufferten Bytes schreibt (flush()). Das weiß er natürlich nicht, aber die Entwickler haben sich dazu entschlossen, eine leere Implementierung anzugeben. Der Vorteil besteht darin, dass Programmierer von Unterklassen nicht verpflichtet werden, immer die Methoden zu überschreiben, auch wenn sie sie gar nicht nutzen wollen.

15.5.4 Die Schnittstellen Closeable, AutoCloseable und Flushable

Zwei besondere Schnittstellen, Closeable und Flushable, schreiben Methoden vor, die alle Ressourcen implementieren, die geschlossen und/oder Daten aus einem internen Puffer herausschreiben sollen.

Closeable

Closeable wird von allen lesenden und schreibenden Datenstrom-Klassen implementiert, die geschlossen werden können. Das sind alle Reader/Writer- und InputStream/OutputStream-Klassen und weitere Klassen wie Socket.

```
interface java.io.Closeable
extends AutoClosable
```

- void close() throws IOException

Schließt den Datenstrom. Einen geschlossenen Strom noch einmal zu schließen, hat keine Konsequenz.

Die Schnittstelle Closeable erweitert seit Java 7 java.lang.AutoCloseable, sodass alles, was Closeable implementiert, damit vom Typ AutoCloseable ist und als Variable bei einem try-mit-Ressourcen verwendet werden kann.

```
interface java.lang.AutoClosable
```

- void close() throws Exception

Schließt den Datenstrom. Einen geschlossenen Strom noch einmal zu schließen, hat keine Konsequenz.

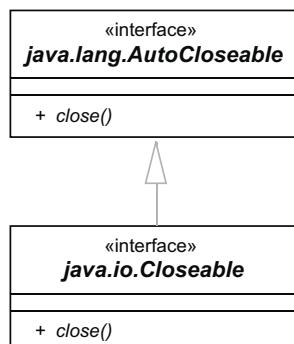


Abbildung 15.6: Das Klassendiagramm zeigt die Verebungsbeziehung zwischen Closeable und AutoCloseable



Hinweis

Jeder `InputStream`, `OutputStream`, `Reader` und `Writer` implementiert `close()` – und mit dem `close()` auch den Zwang, eine geprüfte `IOException` zu behandeln. Bei einem Eingabestrom ist die Exception nahezu wertlos und kann auch tatsächlich ignoriert werden. Bei einem Ausgabestrom ist die Exception schon deutlich wertvoller. Das liegt an der Aufgabe von `close()`, die nicht nur darin besteht, die Ressource zu schließen, sondern vorher noch gepufferte Daten zu schreiben. Somit ist ein `close()` oft ein indirektes `write()`, und hier es ist es sehr wohl wichtig zu wissen, ob alle Restdaten korrekt geschrieben wurden. Die Ausnahme sollte auf keinen Fall ignoriert werden und der catch-Block darf nicht einfach leer bleiben; Logging ist hier das Mindeste.

Flushable

`Flushable` findet sich nur bei schreibenden Klassen und ist insbesondere bei den Klassen wichtig, die Daten puffern.

```
interface java.io.Flushable
```

- `void flush() throws IOException`
Schreibt gepufferte Daten in den Strom.

Die Basisklassen `Reader` und `OutputStream` implementieren diese Schnittstelle, aber auch `Formatter` tut dies.

15.5.5 Die abstrakte Basisklasse `InputStream`

Das Gegenstück zu `OutputStream` ist `InputStream`; jeder binäre Eingabestrom wird durch die abstrakte Klasse `InputStream` repräsentiert. Die Konsoleneingabe `System.in` ist vom Typ `InputStream`.

```
abstract class java.io.InputStream
implements Closeable
```

- `int available() throws IOException`
Gibt die Anzahl der verfügbaren Zeichen im Datenstrom zurück, die sofort ohne Blockierung gelesen werden können.

- `int read() throws IOException`
Liest ein Byte als Integer aus dem Datenstrom. Ist das Ende des Datenstroms erreicht, wird `-1` übergeben. Die Methode ist überladen, wie die nächsten Signaturen zeigen.
- `int read(byte[] b) throws IOException`
Liest mehrere Bytes in ein Feld. Die tatsächliche Länge der gelesenen Bytes wird zurückgegeben und muss nicht `b.length()` sein.
- `int read(byte[] b, int off, int len) throws IOException`
Liest den Datenstrom in ein Byte-Feld, schreibt ihn aber erst an der Stelle `off` in das Byte-Feld. Zudem begrenzt `len` die maximale Anzahl der zu lesenden Zeichen.
- `long skip(long n) throws IOException`
Überspringt eine Anzahl von Zeichen. Die Rückgabe gibt die tatsächlich gesprungenen Bytes zurück, was nicht mit `n` identisch sein muss.
- `void close() throws IOException`
Schließt den Datenstrom. Operation aus der Schnittstelle `Closeable`.
- `boolean markSupported()`
Gibt einen Wahrheitswert zurück, der besagt, ob der Datenstrom das Merken und Zurücksetzen von Positionen gestattet. Diese Markierung ist ein Zeiger, der auf bestimmte Stellen in der Eingabedatei zeigen kann.
- `void mark(int readlimit)`
Markt sich eine Position im Datenstrom.
- `void reset() throws IOException`
Springt wieder zu der Position zurück, die mit `mark()` gesetzt wurde.

Auffällig ist, dass bis auf `mark()` und `markSupported()` alle Methoden im Fehlerfall eine `IOException` auslösen.

Hinweis

`available()` liefert die Anzahl Bytes, die ohne Blockierung gelesen werden können.
(»Blockieren« bedeutet, dass die Methode nicht sofort zurückkehrt, sondern erst wartet, bis neue Daten vorhanden sind.) Die Rückgabe von `available()` sagt nichts darüber aus, wie viele Zeichen der `InputStream` insgesamt hergibt. Während aber bei `FileInputStream` die Methode `available()` üblicherweise doch die Dateilänge liefert, ist dies bei den Netzwerk-Streams im Allgemeinen nicht der Fall.

15.5.6 Ressourcen aus dem Klassenpfad und aus Jar-Archiven laden

Um Ressourcen wie Grafiken oder Konfigurationsdateien aus Jar-Archiven zu laden, ist die Methode `getResourceAsStream()` beziehungsweise `getResource()` ideal. Beide sind Methoden des `Class`-Objekts. `getResource()` gibt ein `URL`-Objekt für die Ressource zurück. Da oft der Inhalt des Datenstroms interessant ist, liefert `getResourceAsStream()` einen `InputStream`. Intern wird aber nichts anderes gemacht, als `getResource()` aufzurufen und mit `openStream()` ein Eingabe-Objekt zu holen. Nur `getResourceAsStream()` fängt eine eventuelle `IOException` ab und liefert dann die Rückgabe `null`.

Da der Klassenlader die Ressource findet, entdeckt er alle Dateien, die im Pfad des Klassenladers eingetragen sind. Das gilt auch für Jar-Archive, weil dort vom Klassenlader alles verfügbar ist. Konnte die Quelle nicht aufgelöst werden, liefern die Methoden `null`. Die Methode `getResourceAsStream()` liefert auch `null`, wenn die Sicherheitsrichtlinien das Lesen verbieten.

zB Beispiel

Besorge einen Eingabestrom `in1` auf die Datei `kullin_fun.txt` und einen zweiten Eingabestrom `in2` auf die Datei `hirse_fun.jpg` innerhalb der eigenen Methode `init()`:

```
class Classi
{
    InputStream in1 = Classi.class.getResourceAsStream( "kullin_fun.txt" );
    void init()
    {
        InputStream in2 = getClass().getResourceAsStream( "hirse_fun.jpg" );
    }
}
```

Da zum Nutzen der `getResourceXXX()`-Methoden ein `Class`-Objekt nötig ist, zeigt das Beispiel zum einen, dass über `Classi.class` das `Class`-Objekt zu bekommen ist, und zum anderen, dass in einer Objektmethode ebenfalls die geerbte `Object`-Methode `getClass()` ein `Class`-Objekt liefert.

15.5.7 Die abstrakte Basisklasse Writer

Die Basis für alle wichtigen Klassen ist die abstrakte Basisklasse `Writer`.

```
abstract class java.io.Writer
    implements Appendable, Closeable, Flushable
```

- `protected Writer(Object lock)`
Erzeugt einen Writer-Stream, der sich mit dem übergebenen Synchronisationsobjekt initialisiert. Ist die Referenz null, so gibt es eine NullPointerException.
- `protected Writer()`
Erzeugt einen Writer-Stream, der sich selbst als Synchronisationsobjekt nutzt. Der Konstruktor ist für die Unterklassen interessant, die kein eigenes Lock-Objekt zuordnen wollen.
- `void write(int c) throws IOException`
Schreibt ein einzelnes Zeichen. Von der 32-Bit-Ganzzahl wird der niedrige Teil (16 Bit des int) geschrieben.
- `void write(char[] cbuf) throws IOException`
Schreibt ein Feld von Zeichen.
- `abstract void write(char[] cbuf, int off, int len) throws IOException`
Schreibt len Zeichen des Felds cbuf ab der Position off.
- `void write(String str) throws IOException`
Schreibt einen String.
- `void write(String str, int off, int len) throws IOException`
Schreibt len Zeichen der Zeichenkette str ab der Position off.
- `Writer append(char c) throws IOException`
Hängt ein Zeichen an. Verhält sich wie `write(c)`, nur liefert es, wie die Schnittstelle Appendable verlangt, ein Appendable zurück. Writer ist ein passendes Appendable.
- `Writer append(CharSequence csq) throws IOException`
Hängt eine Zeichenfolge an. Implementierung aus der Schnittstelle Appendable.
- `abstract void flush() throws IOException`
Schreibt den internen Puffer. Hängt verschiedene `flush()`-Aufrufe zu einer Kette zusammen, die sich aus der Abhängigkeit der Objekte ergibt. So werden alle Puffer geschrieben. Implementierung aus der Schnittstelle Flushable.
- `abstract void close() throws IOException`
Schreibt den gepufferten Strom und schließt ihn. Nach dem Schließen durchgeföhrte `write()`- oder `flush()`-Aufrufe bringen eine IOException mit sich. Ein zusätzliches `close()` löst keine Exception aus. Implementierung aus der Schnittstelle Closeable.

15.5.8 Die abstrakte Basisklasse Reader

Die abstrakte Klasse Reader dient zum Lesen von Zeichen aus einem zeichengebenden Eingabestrom. Die einzigen Methoden, die Unterklassen implementieren müssen, sind `read(char[], int, int)` und `close()`. Dies entspricht dem Vorgehen bei den Writer-Klassen, die auch nur `close()` und `write(char[], int, int)` implementieren müssen. Eine abstrakte `flush()`-Methode, wie sie Writer besitzt, kann Reader nicht haben. Es bleiben demnach für die Reader-Klasse zwei abstrakte Methoden übrig. Die Unterklassen implementieren jedoch auch andere Methoden aus Geschwindigkeitsgründen neu.

```
abstract class java.io.Reader
    implements Readable, Closeable
```

- `protected Reader()`
Erzeugt einen neuen Reader, der sich mit sich selbst synchronisiert.
- `protected Reader(Object lock)`
Erzeugt einen neuen Reader, der mit dem Objekt lock synchronisiert ist.
- `abstract int read(char[] cbuf, int off, int len) throws IOException`
Liest len Zeichen in den Puffer cbuf ab der Stelle off. Wenn len Zeichen nicht vorhanden sind, wartet der Reader. Die Methode gibt die Anzahl gelesener Zeichen zurück oder -1, wenn das Ende des Stroms erreicht wurde.
- `int read(CharBuffer target) throws IOException`
Liest Zeichen in den CharBuffer. Die Methode schreibt die Schnittstelle Readable vor.
- `int read() throws IOException`
Die parameterlose Methode liest das nächste Zeichen aus dem Eingabestrom. Sie wartet, wenn kein Zeichen im Strom bereitliegt. Der Rückgabewert ist ein int im Bereich von 0 bis 65.635 (0x0000–0xFFFF). Warum dann der Rückgabewert aber int und nicht char ist, kann leicht damit erklärt werden, dass die Methode den Rückgabewert -1 (0xFFFFFFFF) kodieren muss, falls keine Daten anliegen.
- `int read(char[] cbuf) throws IOException`
Liest Zeichen aus dem Strom und schreibt sie in ein Feld. Die Methode wartet, bis Eingaben anliegen. Der Rückgabewert ist die Anzahl der gelesenen Zeichen oder -1, wenn das Ende des Datenstroms erreicht wurde.
- `abstract void close() throws IOException`
Schließt den Strom. Folgt anschließend noch ein Aufruf von `read()`, `ready()`, `mark()` oder `reset()`, lösen diese eine `IOException` aus. Ein doppelt geschlossener Stream hat keinen weiteren Effekt.

Weitere Methoden

Zu diesen notwendigen Methoden, die bei der Klasse Reader gegeben sind, kommen noch weitere interessante Methoden hinzu, die den Status abfragen und Positionen setzen lassen. Die Methode `ready()` liefert als Rückgabe `true`, wenn ein `read()` ohne Blockierung der Eingabe möglich ist. Die Standard-Implementierung der abstrakten Klasse Reader gibt immer `false` zurück.

Beispiel

zB

Zum Lesen aller Zeichen muss der Datenstrom so lange ausgesaugt werden, bis keine Daten mehr verfügbar sind. Der Endetest kann auf zwei Arten geschehen: einmal über `ready()` und einmal durch den Test der Rückgabe von `read()` auf `-1`. Die erste Variante:

```
while ( reader.ready() )
    System.out.println( reader.read() );
```

Und die zweite:

```
for ( int c; (c = reader.read()) != -1; )
    System.out.println( (char) c );
```

Die erste Lösung wirkt aufgeräumter.

15

```
abstract class java.io.Reader
    implements Readable, Closeable
```

- `public boolean ready() throws IOException`
Liefert `true`, wenn aus dem Stream direkt gelesen werden kann. Das heißt allerdings nicht, dass `false` immer Blocken bedeutet.

Hinweis



`InputStream` und `Reader` sind sich zwar sehr ähnlich, aber ein `InputStream` deklariert keine Methode `ready()`. Dafür gibt es in `InputStream` eine Methode `available()`, die sagt, wie viele Bytes ohne Blockierung gelesen werden können. Diese Methode gibt es wiederum nicht im `Reader`.

Sprünge und Markierungen

Mit der Methode `mark()` lässt sich eine bestimmte Position innerhalb des Eingabestroms markieren. Die Methode sichert dabei die Position. Mit beliebigen `reset()`-Aufrufen

lässt sich diese konkrete Stelle zu einem späteren Zeitpunkt wieder anspringen. `mark()` besitzt einen Ganzzahl-Parameter, der angibt, wie viele Zeichen gelesen werden dürfen, bevor die Markierung nicht mehr gültig ist. Die Zahl ist wichtig, da sie die interne Größe des Puffers bezeichnet, der für den Strom angelegt werden muss. Nicht jeder Datenstrom unterstützt dieses Hin- und Herspringen. Die Klasse `StringReader` unterstützt etwa die Markierung einer Position, die Klasse `FileReader` dagegen nicht. Daher sollte vorher mit `markSupported()` überprüft werden, ob das Markieren auch unterstützt wird. Wenn der Datenstrom es nicht unterstützt und wir diese Warnung ignorieren, werden wir eine `IOException` bekommen. Denn Reader implementiert `mark()` und `reset()` ganz einfach und muss von uns im Bedarfsfall überschrieben werden:

```
public void mark( int readAheadLimit ) throws IOException {
    throw new IOException("mark() not supported");
}
public void reset() throws IOException {
    throw new IOException("reset() not supported");
}
```

Daher gibt `markSupported()` auch in der Reader-Klasse `false` zurück.

```
abstract class java.io.Reader
implements Readable, Closeable
```

- `long skip(long n) throws IOException`
Überspringt `n` Zeichen. Blockt, bis Zeichen vorhanden sind. Gibt die Anzahl der wirklich übersprungenen Zeichen zurück.
- `boolean markSupported()`
Der Stream unterstützt die `mark()`-Operation.
- `void mark(int readAheadLimit) throws IOException`
Markiert eine Position im Stream. Der Parameter bestimmt, nach wie vielen Zeichen die Markierung ungültig wird, mit anderen Worten: Er gibt die Puffergröße an.
- `void reset() throws IOException`
Falls eine Markierung existiert, setzt der Stream an der Markierung an. Wurde die Position vorher nicht gesetzt, dann wird eine `IOException` mit dem String »Stream not marked« ausgelöst.

Reader implementiert die schon bekannte Schnittstelle `Closeable` mit der Methode `close()`. Und so, wie ein Writer die Schnittstelle `Appendable` implementiert, so implemen-

tiert ein Reader die Schnittstelle `Readable` und damit die Operation `int read(CharBuffer target) throws IOException`.

15.6 Datenströme filtern und verketten

So wie im alltäglichen Leben Filter beim Kaffee oder bei Fotoapparaten eine große Rolle spielen, so sind sie auch bei Datenströmen zu finden. Immer dann, wenn Daten von einer Quelle gelesen oder in eine Senke geschrieben werden, können Filter die Daten auf dem Weg verändern. Die Java-Bibliothek sieht eine ganze Reihe von Filtern vor, die sich zwischen die Kommunikation schalten können.

Eingabe	Ausgabe	Anwendung
<code>BufferedInputStream</code>	<code>BufferedOutputStream</code>	Daten puffern
<code>BufferedReader</code>	<code>BufferedWriter</code>	
<code>CheckedInputStream</code>	<code>CheckedOutputStream</code>	Checksumme berechnen
<code>DataInputStream</code>	<code>DataOutputStream</code>	Primitive Datentypen aus dem Strom holen und in den Strom schreiben
<code>DigestInputStream</code>	<code>DigestOutputStream</code>	Digest (Checksumme) mitberechnen
<code>InflaterInputStream</code>	<code>DeflaterOutputStream</code>	Kompression von Daten
<code>LineNumberInputStream</code>		Mitzählen von Zeilen
<code>LineNumberReader</code>		
<code>PushbackInputStream</code>		Daten in den Lesestrom zurücklegen
<code>PushbackReader</code>		
<code>CipherInputStream</code>	<code>CipherOutputStream</code>	Daten verschlüsseln und entschlüsseln

Tabelle 15.8: Filter zwischen Ein- und Ausgabe

Der `CipherOutputStream` stammt als Einziger aus dem Paket `javax.crypto`; manche Typen sind aus `java.util.zip`, und alle anderen stammen aus `java.io`.

15.6.1 Streams als Filter verketten (verschachteln)

Die Funktionalität der bisher vorgestellten Ein-/Ausgabe-Klassen reicht für den Alltag zwar aus, doch sind Ergänzungen gefordert, die die Fähigkeiten der Klassen erweitern; so zum Beispiel beim Puffern. Da die Programmlogik zur Pufferung mit Daten unabhän-

gig von der Quelle ist, aus der die Daten stammen, findet sich die Pufferung in einer gesonderten Klasse. Java implementiert hier ein bekanntes Muster, das sich *Dekorator* nennt. Zwei Zeilen sollen dieses Prinzip verdeutlichen, um gepufferte Daten in eine Datei zu schreiben:

```
Writer fw = new FileWriter( filename );
Writer bw = new BufferedWriter( fw );
```

Der Konstruktor von `BufferedWriter` nimmt einen beliebigen anderen `Writer` auf, denn der Pufferung ist es egal, ob die Daten in eine Datei oder ins Netzwerk geschrieben werden. Das Prinzip ist also immer, dass der Filter einen anderen Strom annimmt, an den er die Daten weitergibt oder von dem er sie holt.

Schauen wir uns die Klassen im Paket `java.io` genau an, die andere Ströme im Konstruktor entgegennehmen:

- `BufferedWriter`, `PrintWriter`, `FilterWriter` nehmen `Writer`.
- `BufferedReader`, `FilterReader`, `LineNumberReader`, `PushbackReader`, `StreamTokenizer` nehmen `Reader`.
- `BufferedOutputStream`, `DataOutputStream`, `FilterOutputStream`, `ObjectOutputStream`, `OutputStreamWriter`, `PrintStream`, `PrintWriter` nehmen `OutputStream`.
- `BufferedInputStream`, `DataInputStream`, `FilterInputStream`, `InputStreamReader`, `ObjectInputStream`, `PushbackInputStream` nehmen `InputStream`.

15.6.2 Gepufferte Ausgaben mit `BufferedWriter` und `BufferedOutputStream`

Die Klassen `BufferedWriter` und `BufferedOutputStream` haben die Aufgabe, die mittels `write()` in den Ausgabestrom geleiteten Ausgaben zu puffern. Dies ist immer dann nützlich, wenn viele Schreiboperationen gemacht werden, denn das Puffern macht insbesondere Dateioperationen wesentlich schneller, da so mehrere Schreiboperationen zu einer zusammengefasst werden. Um die Funktionalität eines Puffers zu erhalten, besitzen die Klassen einen internen Puffer, in dem die Ausgaben von `write()` zwischengespeichert werden. Standardmäßig fasst der Puffer 8.192 Symbole. Er kann aber über einen parametrisierten Konstruktor auf einen anderen Wert gesetzt werden. Erst wenn der Puffer voll ist oder die Methoden `flush()` oder `close()` aufgerufen werden, werden die gepufferten Ausgaben geschrieben. Durch die Verringerung der Anzahl tatsächlicher `write()`-Aufrufe an das externe Gerät erhöht sich die Geschwindigkeit der Anwendung im Allgemeinen deutlich.

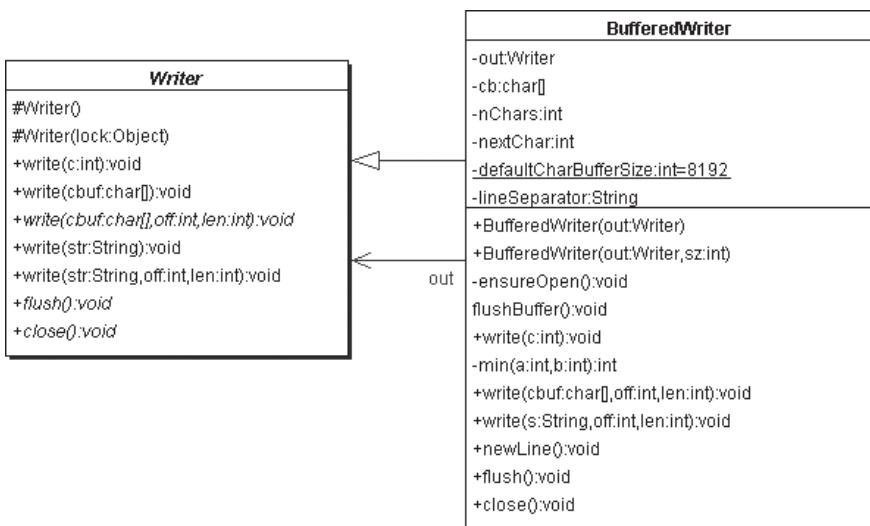


Abbildung 15.7: BufferedWriter ist ein Writer und dekoriert einen anderen Writer

Um einen `BufferedWriter/BufferedOutputStream` anzulegen, gibt es zwei Konstruktoren, denen ein bereits existierender `Writer/OutputStream` übergeben wird. An diesen `Writer/OutputStream` wird dann der Filter seinerseits die Ausgaben weiterleiten, insbesondere nach einem Aufruf von `flush()`, `close()` oder einem internen Überlauf.

<pre> class java.io.BufferedReader extends Writer </pre>	<pre> class java.io.BufferedOutputStream extends FilterOutputStream </pre>
<ul style="list-style-type: none"> ■ <code>BufferedReader(Writer out)</code> ■ <code>BufferedOutputStream(OutputStream out)</code> Erzeugt einen puffernden Ausgabestrom mit der Puffergröße von 8.192 Symbolen. ■ <code>BufferedReader(Writer out, int sz)</code> ■ <code>BufferedOutputStream(OutputStream out, int size)</code> Erzeugt einen puffernden Ausgabestrom mit einer Puffergröße. Ist sie nicht echt größer 0, gibt es eine <code>IllegalArgumentException</code>. 	

Alle `write()`- und `append()`-Methoden sind so implementiert, dass die Daten erst im Puffer landen. Wenn der Puffer voll ist – oder `flush()` aufgerufen wird –, werden sie an den im Konstruktor übergebenen `Writer` durchgespült.

Beispiel zum BufferedWriter mit FileWriter und PrintWriter

Ein `FileWriter` sichert Daten in einer Datei. Ein `BufferedWriter` soll aber vorher die Daten erst einmal sammeln, sodass sie erst beim `Flush` an den `FileWriter` gehen. Der Anwendungsentwickler soll in unserem Beispiel aber nicht direkt den `BufferedWriter` nutzen, sondern ihn als allgemeinen `Writer` im Konstruktor von `PrintWriter` übergeben. Ein `PrintWriter` besitzt die komfortablen Methoden `print()`, `println()` und `printf()`, sodass wir nicht mehr nur auf `write()`-Methoden vom `Writer` angewiesen sind:

Listing 15.8: com/tutego/insel/io/writer/ChainedWriter.java, main()

```
PrintWriter pw = null;
try
{
    Writer fw = new FileWriter( "charArrayWriterDemoPuffer.txt" );
    Writer bw = new BufferedWriter( fw );
    pw = new PrintWriter( bw );

    for ( int i = 1; i < 10000; i++ )
        pw.println( "Zeile " + i );
}
catch ( IOException e ) {
    System.err.println( "Error creating file!" );
}
finally {
    if ( pw != null )
        pw.close();
}
```

Zusätzlich bietet die Klasse `BufferedWriter` die Methode `newLine()`, die in der Ausgabe eine neue Zeile beginnt. Das Zeichen für den Zeilenwechsel wird aus der Systemeigenschaft `line.separator` genommen. Da sie intern mit der `write()`-Methode arbeitet, kann sie eine `IOException` auslösen.

15.6.3 Gepufferte Eingaben mit BufferedReader/BufferedInputStream

Die Klassen `BufferedReader` und `BufferedInputStream` puffern Eingaben. Die Daten werden also zuerst in einen Zwischenspeicher geladen, was insbesondere bei Dateien zu

weniger Zugriffen auf den Datenträger führt und so die Geschwindigkeit der Anwendung erhöht.

Die Klassen `BufferedReader` und `BufferedInputStream` besitzen je zwei Konstruktoren. Bei einem lässt sich die Größe des internen Puffers angeben. Die Puffergröße beträgt wie beim `BufferedWriter/BufferedOutputStream` standardmäßig 8.192 Einträge.

<code>class java.io.BufferedReader</code>	<code>class java.io.BufferedInputStream</code>
<code>extends Reader</code>	<code>extends FilterInputStream</code>

- `BufferedReader(Reader in)`
- `BufferedInputStream(InputStream in)`
Erzeugt einen puffernden Zeichenstrom mit der Puffergröße von 8.192.
- `BufferedReader(Reader in, int sz)`
- `BufferedInputStream(InputStream in, int size)`
Erzeugt einen puffernden Zeichenstrom mit der gewünschten Puffergröße.

Programm zur Anzeige von Dateien

Das folgende Programm implementiert ein einfaches »cat«-Kommando⁵ von Unix, um Dateiinhalte über die Standardausgabe auszugeben. Die Dateinamen werden auf der Kommandozeile übergeben:

Listing 15.9: com/tutego/insel/io/stream/cat.java

```
package com.tutego.insel.io.stream;

import java.io.*;

class cat
{
    public static void main( String[] args )
    {
        for ( String filename : args ) {
            try {
                InputStream in = new BufferedInputStream( new FileInputStream(filename) );

```

⁵ Der kurze Name »cat« stammt von »catenate«, einem Synonym für »concatenate«.

```

try {
    for ( int c; (c = in.read()) != -1 /* EOF */; )
        System.out.write( c );
}
finally {
    in.close();
}
}

catch ( IOException e ) {
    System.err.println( "cat: Fehler beim Verarbeiten von " + filename );
    System.exit( 1 );
}
} // end for
}
}

```

Die Dateiangaben nimmt das Programm über die Kommandozeile entgegen; etwa so:

```
$ java com.tutego.insel.io.stream.cat adam.txt eva.txt
```



Hinweis

Insbesondere bei externen Hintergrundspeichern ergibt eine Pufferung Sinn. So sollten zum Beispiel die dateiorientierten Klassen immer gepuffert werden, insbesondere, wenn einzelne Bytes/Zeichen gelesen oder geschrieben werden.

Ohne Pufferung

```

new FileReader(f)
new FileWriter(f)
new FileInputStream(f)
new FileOutputStream(f)

```

In der Regel schneller mit Pufferung

```

new BufferedReader(new FileReader(f))
new BufferedWriter(new FileWriter(f))
new BufferedInputStream(new FileInputStream(f))
new BufferedOutputStream(new FileOutputStream(f))

```

Zeilen lesen mit BufferedReader und readLine()

Die Klasse BufferedReader stellt die Methode `readLine()` zur Verfügung, die eine komplette Textzeile liest und als String an den Aufrufer zurückgibt; `BufferedOutputStream` als byte-orientierte Klasse bietet die Methode nicht an.

```
class java.io.BufferedReader
extends Reader
```

- `String readLine()`
Liest eine Zeile bis zum Zeilenende und gibt den String ohne die Endzeichen zurück.
Die Rückgabe ist `null`, wenn der Stream am Ende ist.

Da ein `BufferedReader` Markierungen und Sprünge erlaubt, werden die entsprechenden Methoden von `Reader` überschrieben.

15.7 Vermittler zwischen Byte-Streams und Unicode-Strömen

15.7.1 Datenkonvertierung durch den OutputStreamWriter

Die Klasse `OutputStreamWriter` ist sehr interessant, da sie Konvertierungen der Zeichen nach einer Zeichenkodierung vornimmt. So wird sie, unterstützt durch die einzige Unterklasse `FileWriter`, für Ausgaben in Dateien noch wichtiger. Jeder `OutputStreamWriter` konvertiert auf diese Weise Zeichenströme von einer Zeichenkodierung (etwa EBCDIC) in die andere (etwa Latin-1). Die Zeichenkodierung kann im Konstruktor eines `OutputStreamWriter`-Objekts angegeben werden. Ohne Angabe ist es der Standardkonvertierer, der in den Systemeigenschaften unter dem Schlüssel `file.encoding` geschrieben ist. Die Kodierung der Zeichen nimmt ein `StreamEncoder` im Paket `sun.nio.cs` vor.

```
class java.io.OutputStreamWriter
extends Writer
```

- `OutputStreamWriter(OutputStream out)`
Erzeugt einen `OutputStreamWriter`, der die Standardkodierung verwendet.
- `OutputStreamWriter(OutputStream out, Charset cs)`
- `OutputStreamWriter(OutputStream out, CharsetEncoder enc)`
Erzeugt einen `OutputStreamWriter` mit einem `Charset` oder einem `CharsetEncoder`.
- `OutputStreamWriter(OutputStream out, String enc)`
Erzeugt einen `OutputStreamWriter` mit der vorgegebenen Kodierung.
- `void close()`
Schließt den Datenstrom.
- `void flush()`
Schreibt den gepufferten Strom.

- `String getEncoding()`
Liefert die Kodierung des Datenstroms als String.
- `void write(char[] cbuf, int off, int len)`
Schreibt Zeichen des Felds.
- `void write(int c)`
Schreibt ein einzelnes Zeichen.
- `void write(String str, int off, int len)`
Schreibt den Teil eines Strings.

FileWriter, OutputStreamWriter und FileOutputStream

`OutputStreamWriter` ist die Basisklasse für die konkrete Klasse `FileWriter` und ist für die Konvertierung der Zeichen in Bytefolgen verantwortlich. Die Konstruktoren bauen ein `FileOutputStream`-Objekt auf und füttern damit den Konstruktor von `OutputStreamWriter`. Die `write()`-Methoden vom `OutputStreamWriter` konvertieren die Zeichen in Bytes, die letztendlich der `FileOutputStream` schreibt:

```
public class FileWriter extends OutputStreamWriter
{
    public FileWriter(String fileName) throws IOException {
        super(new FileOutputStream(fileName));
    }
    public FileWriter(String fileName, boolean append)
        throws IOException {
        super(new FileOutputStream(fileName, append));
    }
    ...
}
```

15.7.2 Automatische Konvertierungen mit dem InputStreamReader

Die konkrete Klasse `InputStreamReader` nimmt eine Konvertierung zwischen Byte- und Zeichen-Streams vor. Sie arbeitet wie ein `OutputStreamWriter` und konvertiert die Daten mithilfe eines `sun.nio.cs.StreamDecoder`.

```
class java.io.InputStreamReader  
extends Reader
```

- `InputStreamReader(InputStream in)`
Erzeugt einen InputStreamReader mit der Standardkodierung.
- `InputStreamReader(InputStream in, String enc)`
`throws UnsupportedEncodingException`
Erzeugt einen InputStreamReader, der die angegebene Zeichenkodierung anwendet.
- `String getEncoding()`
Liefert einen String mit dem Namen der Kodierung zurück. Der Name ist kanonisch und kann sich daher von dem String unterscheiden, der im Konstruktor übergeben wurde.
- `int read() throws IOException`
Liest ein einzelnes Zeichen oder gibt -1 zurück, falls der Stream am Ende ist.
- `int read(char[] cbuf, int off, int len) throws IOException`
Liest Zeichen in einen Teil eines Feldes.
- `boolean ready() throws IOException`
Kann vom Stream gelesen werden. Ein InputStreamReader ist bereit, wenn der Eingabe- puffer nicht leer ist oder Bytes des darunter befindlichen InputStreams anliegen.

Wie wir an dieser Stelle bemerken, unterstützt ein reiner `InputStream` kein `mark()` und `reset()`. Da `FileReader` die einzige Klasse in der Java-Bibliothek ist, die einen `InputStreamReader` erweitert, und diese Klasse ebenfalls kein `mark()` beziehungsweise `reset()` unterstützt, lässt sich sagen, dass kein `InputStreamReader` der Standardbibliothek Positionsmarkierungen erlaubt.



Kapitel 16

Einführung in die <XML>-Verarbeitung mit Java

»Haben Sie keine Angst vor der Zukunft, sie beginnt erst morgen.«
– Zarko Petan (*1929)

16.1 Auszeichnungssprachen

Auszeichnungssprachen dienen zur strukturierten Gliederung von Texten und Daten. Ein Text besteht zum Beispiel aus Überschriften, Fußnoten und Absätzen, eine Vektorgrafik dagegen aus einzelnen Grafikelementen wie Linien und Textfeldern. Auszeichnungssprachen liegt die Idee zugrunde, besondere Bausteine durch Auszeichnung hervorzuheben. Ein Text könnte etwa so beschrieben sein:

```
<Überschrift>
Mein Buch
<Ende Überschrift>
Hui ist das <fett>toll<Ende fett>.
```

Als Leser eines Buchs erkennen wir optisch eine Überschrift an der Schriftart. Ein Computer hat damit aber seine Schwierigkeiten. Wir wollen auch dem Rechner die Fähigkeit verleihen, diese Struktur zu erkennen.

HTML ist die erste populäre Auszeichnungssprache, die Auszeichnungselemente (engl. *tags*) wie `fett` benutzt, um bestimmte Eigenschaften von Elementen zu kennzeichnen. Damit wurde eine Visualisierung verbunden, etwa bei einer Überschrift fett und mit großer Schrift. Leider werden Auszeichnungssprachen wie HTML auch dazu benutzt, Formatierungseffekte zu erzielen. Beispielsweise werden Überschriften richtigerweise mit dem Überschriften-Tag ausgezeichnet. Wenn an anderer Stelle eine Textstelle fett und groß sein soll, wird diese aber auch oft mit dem Überschriften-Tag markiert, obwohl sie keine Überschrift ist.

16.1.1 Die Standard Generalized Markup Language (SGML)

Das Beispiel der Überschrift in einem Buch veranschaulicht die Idee, Bausteine mit Typen in Verbindung zu bringen. Der allgemeine Aufbau mit diesen Auszeichnungselementen ließe sich dann für beliebige hierarchische Dokumente nutzen. Die Definition einer Auszeichnungssprache (Metasprache) ist daher auch nicht weiter schwierig. Schon Mitte der 1980er-Jahre wurde als ISO-Standard die *Standard Generalized Markup Language* (SGML)¹ definiert, die die Basis für beliebige Auszeichnungssprachen ist. Ab der Version 2.0 ist auch HTML als SGML-Anwendung definiert. Leider kam mit den vielen Freiheiten und der hohen Flexibilität eine große und aufwändige Deklaration der Anwendungen hinzu. Ein SGML-Dokument musste einen ganz bestimmten Aufbau besitzen. SGML-Dateien waren daher etwas unflexibel, weil die Struktur genau eingehalten werden musste. Für HTML-Dateien wäre das schlecht, weil die Browser-Konkurrenten produktspezifische Tags definieren, die auf den Browser des jeweiligen Herstellers beschränkt bleiben. So interpretiert der Internet Explorer zum Beispiel das Tag `<blink>blinking</blink>` nicht. Tags, die ein Browser nicht kennt, überliest er einfach.

16.1.2 Extensible Markup Language (XML)

Für reine Internetseiten hat sich HTML etabliert, aber für andere Anwendungen wie Datenbanken oder Rechnungen ist HTML nicht geeignet. Für SGML sprechen die Korrektheit und Leistungsfähigkeit – dagegen sprechen die Komplexität und die Notwendigkeit, eine Beschreibung für die Struktur angeben zu müssen. Daher setzte sich das W3C zusammen, um eine neue Auszeichnungssprache zu entwickeln, die einerseits so flexibel wie SGML, andererseits aber so einfach zu nutzen und zu implementieren ist wie HTML. Das Ergebnis war die *Extensible Markup Language* (XML). Diese Auszeichnungssprache ist für Compiler einfach zu verarbeiten, da es genaue Vorgaben dafür gibt, wann ein Dokument in Ordnung ist.

XML ist nicht nur der Standard zur Beschreibung von Daten, denn oft verbinden sich mit diesem Ausdruck eine oder mehrere Technologien, die mit der Beschreibungssprache im Zusammenhang stehen. Und: Ohne XML kein WiX²! Die wichtigsten Technologien zur Verarbeitung von XML in Java werden hier kurz vorgestellt. Eine ausführliche Beschreibung mit allen Nachbartechnologien finden Sie bei Interesse auf den Webseiten des W3C unter <http://www.w3c.org/>.

1 Der Vorgänger von SGML war GML; hier standen die Buchstaben (sicherlich inoffiziell) für Charles Goldfarb, Edward Mosher und Raymond Lorie, die bei IBM in den 1960er-Jahren diese Dokumentenbeschreibungssprache geschaffen hatten.

2 Windows Installer XML – Definition von Auslieferungspaketen für Microsoft Windows

16.2 Eigenschaften von XML-Dokumenten

16.2.1 Elemente und Attribute

Der Inhalt eines XML-Dokuments besteht aus strukturierten *Elementen*, die hierarchisch geschachtelt sind. Dazwischen befindet sich der Inhalt, der aus weiteren Elementen (daher »hierarchisch«) und reinem Text bestehen kann. Die Elemente können *Attribute* enthalten, die zusätzliche Informationen in einem Element ablegen:

Listing 16.1: party.xml

```
<?xml version="1.0"?>
<party datum="31.12.01">
  <gast name="Albert Angsthase">
    <getraenk>Wein</getraenk>
    <getraenk>Bier</getraenk>
    <zustand ledig="true" nuechtern="false"/>
  </gast>
</party>
```

Die Groß- und Kleinschreibung der Namen für Elemente und Attribute ist für die Unterscheidung wichtig. Ein Attribut besteht aus einem Attributnamen und einem Wert. Der Attributwert steht immer in einfachen oder doppelten Anführungszeichen, und das Gleichheitszeichen weist den Wert dem Attributnamen zu.

Verwendung von Tags

Gemäß der *Reference Concrete Syntax* geben Elemente in spitzen Klammern die Tags an. Elemente existieren in zwei Varianten: Falls das Element einen Wert einschließt, besteht es aus einem Anfangs-Tag und einem End-Tag.

$$\text{Element} = \text{öffnendes Tag} + \text{Inhalt} + \text{schließendes Tag}$$

Der Anfangs-Tag gibt den Namen des Tags vor und enthält die Attribute. Der End-Tag hat den gleichen Namen wie der Anfangs-Tag und wird durch einen Schrägstrich nach der ersten Klammer gekennzeichnet. Zwischen dem Anfangs- und dem End-Tag befindet sich der Inhalt des Elements.

zB Beispiel

Das Element <getraenk> mit dem Wert Wein:

```
<getraenk>Wein</getraenk>
```

Ein Element, das keine Inhalte einschließt, besteht aus nur einem Tag mit einem Schrägstrich vor der schließenden spitzen Klammer. Diese Tags haben entweder Attribute als Inhalt, oder das Auftreten des Tags ist Bestandteil des Inhalts.

zB Beispiel

Das Element <zustand> mit den Attributen ledig und nuechtern:

```
<zustand ledig="true" nuechtern="false" />
```

Bedeutung der Tags

Durch die freie Namensvergabe in XML-Dokumenten ist eine formatierte Darstellung eines Dokuments nicht möglich. Anders als bei HTML gibt es keine festgelegte Menge von Tags, die den Inhalt nach bestimmten Kriterien formatieren. Falls das XML-Dokument in einem Browser dargestellt werden soll, sind zusätzliche Beschreibungen in Form von Formatvorlagen (Stylesheets) für die Darstellung in HTML notwendig.

Wohlgeformt

Ein XML-Dokument muss einige Bedingungen erfüllen, damit es *wohlgeformt* ist. Wenn es nicht wohlgeformt ist, ist es auch kein XML-Dokument. Damit ein XML-Dokument wohlgeformt ist, muss jedes Element aus einem Anfangs- und einem End-Tag oder nur aus einem abgeschlossenen Tag bestehen. Hierarchische Elemente müssen in umgekehrter Reihenfolge ihrer Öffnung wieder geschlossen werden. Die Anordnung der öffnenden und schließenden Tags legt die Struktur eines XML-Dokuments fest. Jedes XML-Dokument muss ein Wurzelement enthalten, das alle anderen Elemente einschließt.

zB Beispiel

Das Wurzelement heißt <party> und schließt das Element <gast> ein:

```
<party datum="31.12.01">
  <gast name="Albert Angsthase"></gast>
</party>
```

Spezielle Zeichen in XML (Entitäten)

Wir müssen darauf achten, dass einige Zeichen in XML bestimmte Bedeutungen haben. Dazu gehören &, <, >, " und '. Sie werden im Text durch spezielle Abkürzungen, die *Entitäten*, abgebildet. Dies sind für die oben genannten Zeichen &, <, >, " und '. Diese Entitäten für die Sonderzeichen sind als einzige durch den Standard festgelegt.

<!-- Kommentare -->

XML-Dokumente können auch Kommentare enthalten. Diese werden beim Auswerten der Daten übergeben. Kommentare verbessern die Qualität des XML-Dokuments für den Benutzer wesentlich. Sie können an jeder Stelle des Dokuments verwendet werden, nur nicht innerhalb der Tags. Kommentare haben die Form:

```
<!-- Text des Kommentars -->
```

Der beste Kommentar eines XML-Dokuments ist die sinnvolle Gliederung des Dokuments und die Wahl selbsterklärender Namen für Tags und Attribute.

Kopfdefinition

Die Wohlgeformtheit muss mindestens erfüllt sein. Zusätzlich dürfen andere Elemente eingebaut werden. Dazu gehört etwa eine Kopfdefinition, die beispielsweise

```
<?xml version="1.0"?>
```

lauten kann. Diese Kopfdefinition lässt sich durch Attribute erweitern. In diesem Beispiel werden die verwendete XML-Version und die Zeichenkodierung angegeben:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Wenn eine XML-Deklaration vorhanden ist, muss sie ganz am Anfang des Dokuments stehen. Dort lässt sich auch die benutzte Zeichenkodierung definieren, wenn sie nicht automatisch UTF-8 oder UTF-16 ist. Automatisch kann jedes beliebige Unicode-Zeichen unabhängig von der Kodierung über das Kürzel ꯍ (A, B, C, D stehen für Hexadezimalzeichen) dargestellt werden.

16.2.2 Beschreibungssprache für den Aufbau von XML-Dokumenten

Im Gegensatz zu HTML ist bei XML die Menge der Tags und deren Kombination nicht festgelegt. Für jede Anwendung können Entwickler beliebige Tags definieren und verwenden. Um aber überprüfen zu können, ob eine XML-Datei für eine bestimmte Anwendung die richtige Form hat, wird eine formale Beschreibung dieser Struktur benötigt. Diese formale Struktur ist in einem bestimmten Format beschrieben, wobei zwei Formate populär sind: das XML *Schema* und die *Document Type Definition* (DTD). Sie legen fest, welche Tags zwingend vorgeschrieben sind, welche Art Inhalt diese Elemente haben, wie Tags miteinander verschachtelt sind und welche Attribute ein Element besitzen darf. Hält sich ein XML-Dokument an die Definition, ist es *gültig* (engl. *valid*).

Mittlerweile gibt es eine große Anzahl von Beschreibungen in Form von Schemas und DTDs, die Gültigkeiten für die verschiedensten Daten definieren. Einige DTDs sind unter <http://tutego.de/go/xmlapplications> aufgeführt. Um einen Datenaustausch für eine bestimmte Anwendung zu gewährleisten, ist eine eindeutige Beschreibung unerlässlich. Es wäre problematisch, wenn die Unternehmen unter der Struktur einer Rechnung immer etwas anderes verstünden.

Document Type Definition (DTD)

Für die folgende XML-Datei entwickeln wir eine DTD zur Beschreibung der Struktur:

Listing 16.2: party.xml

```
<?xml version="1.0" ?>
<party datum="31.12.01">
    <gast name="Albert Angsthase">
        <getraenk>Wein</getraenk>
        <getraenk>Bier</getraenk>
        <zustand ledig="true" nuechtern="false"/>
    </gast>
    <gast name="Martina Mutig">
        <getraenk>Apfelsaft</getraenk>
        <zustand ledig="true" nuechtern="true"/>
    </gast>
    <gast name="Zacharias Zottelig"></gast>
</party>
```

Für diese XML-Datei legen wir die Struktur fest und beschreiben sie in einer DTD. Dazu sammeln wir zuerst die Daten, die in dieser XML-Datei stehen.

Elementname	Attribute	Untergeordnete Elemente	Aufgabe
party	datum Datum der Party	gast	Wurzelement mit dem Datum der Party als Attribut
gast	name Name des Gastes	getraenk und zustand	Die Gäste der Party; Name des Gastes als Attribut
getraenk			Getränk des Gastes als Text
zustand	ledig und nuechtern		Familienstand und Zustand als Attribute

Tabelle 16.1: Struktur der Beispiel-XML-Datei

Elementbeschreibung

Die Beschreibung der Struktur eines Elements besteht aus dem Elementnamen und dem Typ. Sie kann auch aus einem oder mehreren untergeordneten Elementen in Klammern bestehen. Der Typ legt die Art der Daten in dem Element fest. Mögliche Typen sind etwa PCDATA (*Parsed Character Data*) für einfachen Text oder ANY für beliebige Daten.

Untergeordnete Elemente werden als Liste der Elementnamen angegeben. Die Namen sind durch ein Komma getrennt. Falls verschiedene Elemente oder Datentypen alternativ vorkommen können, werden diese ebenfalls in Klammern angegeben und mit dem Oder-Operator (|) verknüpft. Hinter jedem Element und hinter der Liste von Elementen legt ein Operator fest, wie häufig das Element oder die Folgen von Elementen erscheinen müssen. Falls kein Operator angegeben ist, muss das Element oder die Elementliste genau einmal erscheinen. Folgende Operatoren stehen zur Verfügung:

Operator	Wie oft erscheint das Element?
?	Einmal oder gar nicht
+	Mindestens einmal
*	Keinmal, einmal oder beliebig oft

Tabelle 16.2: DTD-Operatoren für Wiederholungen

zB Beispiel

Das Element <party> erlaubt beliebig viele Unterelemente vom Typ <gast>:

```
<!ELEMENT party (gast)*>
```

Drückt aus, dass auf einer Party beliebig viele Gäste erscheinen können.

Attributbeschreibung

Die Beschreibung der Attribute sieht sehr ähnlich aus. Sie besteht aus dem Element, den Attributnamen, den Datentypen der Attribute und einem Modifizierer. In einem Attribut können als Werte keine Elemente angegeben werden, sondern nur Datentypen wie CDATA (*Character Data*). Der Modifizierer legt fest, ob ein Attribut zwingend vorgeschrieben ist oder nicht. Folgende Modifizierer stehen zur Verfügung:

Modifizierer	Erläuterung
#IMPLIED	Muss nicht vorkommen.
#REQUIRED	Muss auf jeden Fall vorkommen.
#FIXED [Wert]	Wert wird gesetzt und kann nicht verändert werden.

Tabelle 16.3: Attribut-Modifizierer

zB Beispiel

Das Attribut datum für das Element <party>:

```
<!ATTLIST party datum CDATA #REQUIRED>
```

Der Wert des Attributs `datum` ist Text und muss angegeben sein (festgelegt durch den Modifizierer `#REQUIRED`).

Kümmern wir uns um die Beschreibung eines Gasts, der einen Namen und einen Zustand hat:

```
<!ELEMENT gast (getraenk*, zustand?)>
<!ATTLIST gast name CDATA #REQUIRED>
```

Das Element hat als Attribut `name` und die Unterelemente `<getraenk>` und `<zustand>`. Ein Gast kann kein Getränk, ein Getränk oder viele einnehmen. Die Attribute des Elements `<zustand>` müssen genau einmal oder gar nicht vorkommen.

Das Element `<getraenk>` hat keine Unterelemente, aber einen Text, der das Getränk beschreibt:

```
<!ELEMENT getraenk (#PCDATA)>
```

Das Element `<zustand>` hat keinen Text und keine Unterelemente, aber die Attribute `ledig` und `nuechtern`, die mit Text gefüllt sind. Die Attribute müssen nicht unbedingt angegeben werden (Modifizierer `#IMPLIED`).

```
<!ELEMENT zustand EMPTY>
<!ATTLIST zustand ledig CDATA #IMPLIED
    nuechtern CDATA #IMPLIED>
```

Bezugnahme auf eine DTD

Falls die DTD in einer speziellen Datei steht, wird im Kopf der XML-Datei angegeben, wo die DTD für dieses XML-Dokument steht:

```
<!DOCTYPE party SYSTEM "dtd\partyfiles\party.dtd">
```

Hinter `DOCTYPE` steht das Wurzelement der zu beschreibenden XML-Datei. Hinter `SYSTEM` steht die URI mit der Adresse der DTD-Datei. Die DTD selbst kann in einer eigenen Datei stehen oder Bestandteil der XML-Datei sein.

Die vollständige DTD zu dem Party-Beispiel sieht folgendermaßen aus:

Listing 16.3: party.dtd

```
<!ELEMENT party (gast)*>
<!ATTLIST party datum CDATA #REQUIRED>
<!ELEMENT gast (getraenk*, zustand?)>
<!ATTLIST gast name CDATA #REQUIRED>
<!ELEMENT getraenk (#PCDATA)>
<!ELEMENT zustand EMPTY>
<!ATTLIST zustand ledig CDATA #IMPLIED nuechtern CDATA #IMPLIED>
```

Diese DTD definiert somit die Struktur aller XML-Dateien, die die Party beschreiben.

16.2.3 Schema – eine Alternative zu DTD

Ein anderes Verfahren, um die Struktur von XML-Dateien zu beschreiben, ist das *Schema*. Es ermöglicht eine Strukturbeschreibung wie eine DTD – nur in Form einer XML-Datei. Das vereinfacht das Parsen der Schema-Datei, da die Strukturbeschreibung und die Daten vom gleichen Dateityp sind. Ein Schema beschreibt im Gegensatz zu einer DTD die Datentypen der Elemente und Attribute einer XML-Datei viel detaillierter. Die üblichen Datentypen wie `string`, `integer` und `double` der gängigen Programmiersprachen sind bereits vorhanden. Weitere Datentypen wie `date` und `duration` existieren ebenfalls. Zusätzlich ist es möglich, eigene Datentypen zu definieren. Mit einem Schema kann weiterhin festgelegt werden, ob ein Element wie eine Ganzzahl in einem speziellen Wertebereich liegt oder ein String auf einen regulären Ausdruck passt. Die Vorteile sind eine genauere Beschreibung der Daten, die in einer XML-Datei dargestellt werden. Das macht aber auch die Strukturbeschreibung aufwändiger als mit einer DTD. Durch die detaillierte Beschreibung der XML-Struktur ist jedoch der Mehraufwand gerechtfertigt.

Party-Schema

Hier ist ein Beispiel für ein Schema, das die Struktur der Datei *party.xml* beschreibt:

Listing 16.4: party.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="party" type="partyType" />

  <xsd:complexType name="partyType">
    <xsd:sequence>
      <xsd:element name="gast" type="gastType" />
    </xsd:sequence>
    <xsd:attribute name="datum" type="datumType" />
  </xsd:complexType>

  <xsd:complexType name="gastType">
    <xsd:sequence>
      <xsd:element name="getraenk" type="xsd:string" />
      <xsd:element name="zustand" type="zustandType" />
    </xsd:sequence>
  </xsd:complexType>
```

```

<xsd:simpleType name="datumType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-3][0-9].[0-1][0-9].[0-9]{4}" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="zustandType">
  <xsd:complexContent>
    <xsd:restriction base="xsd:anyType">
      <xsd:attribute name="nuechtern" type="xsd:boolean" />
      <xsd:attribute name="ledig" type="xsd:boolean" />
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

</xsd:schema>

```

In diesem Beispiel werden die Typen `string` (für die Beschreibung des Elements `<getraenke>`) und `boolean` (für die Beschreibung des Elements `<ledig>`) verwendet. Die Typen `gastType` und `datumType` sind selbst definierte Typen. Ein sehr einfacher regulärer Ausdruck beschreibt die Form eines Datums. Ein Datum besteht aus drei Gruppen zu je zwei Ziffern, die durch Punkte getrennt werden. Die erste Ziffer der ersten Zifferngruppe muss aus dem Zahlenbereich 0 bis 3 stammen.

In der Schema-Datei basieren die Typen `datumType` und `zustandType` auf vorhandenen Schema-Typen, um diese einzuschränken. So schränkt `datumType` den Typ `string` auf die gewünschte Form eines Datums ein, und `zustandType` schränkt den `anyType` auf die beiden Attribute `nuechtern` und `ledig` ein. Die Schreibweise erzeugt einen neuen Typ, der keinen Text als Inhalt enthält, sondern nur die beiden Attribute `nuechtern` und `ledig` erlaubt. Der Wert der beiden Attribute ist ein Wahrheitswert.

16

Simple und komplexe Typen

Ein XML-Schema unterscheidet zwischen simplen und komplexen Typen. Simple Typen sind alle Typen, die keine Unterelemente und keine Attribute haben, sondern nur textbasierten Inhalt.

zB Beispiel

Das Element <getraenk> besteht nur aus einer Zeichenkette:

```
<xsd:element name="getraenk" type="xsd:string" />
```

Komplexe Typen können neben textbasiertem Inhalt auch Unterelemente und Attribute inkludieren.

zB Beispiel

Das Element <gast> hat den Typ gastType und die Unterelemente <getraenk> und <zustand>:

```
<xsd:element name="gast" type="gastType" />
<xsd:complexType name="gastType">
    <xsd:sequence>
        <xsd:element name="getraenk" type="xsd:string" />
        <xsd:element name="zustand" type="zustandType" />
    </xsd:sequence>
</xsd:complexType>
```

Simple und komplexe Typen können andere Typen einschränken. Komplexe Typen können zusätzlich andere Typen erweitern. Beim Erweitern ist es möglich, mehrere Typen miteinander zu kombinieren, um einen neuen Typ mit Eigenschaften verschiedener Typen zu erschaffen.

Das vorige Beispiel kann nur einen kleinen Einblick in die Möglichkeiten von XML-Schemas geben. Eine umfangreiche Dokumentation ist unter der URL <http://www.w3.org/XML/Schema> vorhanden. Dort gibt es drei verschiedene Dokumentationen zum Schema:

- **Schema Part0 Primer:** gut lesbares Tutorial mit vielen Beispielen
- **Schema Part1 Structures:** genaue Beschreibung der Struktur einer Schema-Datei
- **Schema Part2 Datatypes:** Beschreibung der Datentypen, die in XML-Schemas verwendet werden

Der erste Teil bietet eine grundlegende Einführung mit vielen Beispielen. Die beiden anderen Teile dienen als Referenzen für spezielle Fragestellungen.

16.2.4 Namensraum (Namespace)

Das Konzept des *Namensraums* ist besonders wichtig, wenn

- XML-Daten nicht nur lokal mit einer Anwendung benutzt werden,
- Daten ausgetauscht oder
- XML-Dateien kombiniert werden.

Eine Überschneidung der Namen der Tags, die in den einzelnen XML-Dateien verwendet werden, lässt sich nicht verhindern. Daher ist es möglich, einer XML-Datei einen Namensraum oder mehrere Namensräume zuzuordnen.

Der Namensraum ist eine Verknüpfung zwischen einem Präfix, das vor den Elementnamen steht, und einer URI. Ein Namensraum wird als Attribut an ein Element (typischerweise das Wurzelement) gebunden und kann dann von allen Elementen verwendet werden. Das Attribut hat die Form:

```
xmlns:Präfix="URI"
```

Dem Element, das den Namensraum deklariert, wird ein Präfix vorangestellt. Es hat die Form:

```
<Präfix:lokaler Name xmlns:Präfix="URI">
```

Das Präfix ist ein frei wählbares Kürzel, das den Namensraum benennt. Dieses Kürzel wird dem Namen der Elemente, die zu diesem Namensraum gehören, vorangestellt. Der Name eines Elements des Namensraums Präfix hat die Form:

```
<Präfix:lokaler Name>...</Präfix:lokaler Name>
```

Angenommen, wir möchten für unsere Party das Namensraum-Präfix `geburtstag` verwenden. Die URI für diesen Namensraum ist `http://www.geburtstag.de`. Der Namensraum wird in dem Wurzelement `party` deklariert. Das Präfix wird jedem Element zugeordnet:

```
<geburtstag:party xmlns:geburtstag="http://www.geburtstag.de">
    geburtstag:datum="31.12.01">
        <geburtstag:gast geburtstag:name="Albert Angsthase">
            </geburtstag:gast>
        </geburtstag:party>
```

Eine weitere wichtige Anwendung von Namensräumen ist es, Tags bestimmter Technologien zu kennzeichnen. Für die XML-Technologien, etwa für Schemas, werden feste Namensräume vergeben.

zB

Beispiel

Namensraumdefinition für ein XML-Schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Eine Anwendung, die XML-Dateien verarbeitet, kann anhand des Namensraums erkennen, welche Technologie verwendet wird. Dabei ist nicht das Präfix, sondern die URI für die Identifikation des Namensraums entscheidend. Für XML-Dateien, die eine Strukturbeschreibung in Form eines Schemas definieren, ist es üblich, das Präfix xsd zu verwenden. Es ist aber jedes andere Präfix möglich, wenn die URI auf die Adresse <http://www.w3.org/2001/XMLSchema> verweist. Diese Adresse muss nicht unbedingt existieren, und eine Anwendung kann auch nicht erwarten, dass sich hinter dieser Adresse eine konkrete HTML-Seite verbirgt. Die URI dient nur zur Identifikation des Namensraums für eine XML-Datei.

16.2.5 XML-Applikationen *

Eine *XML-Applikation* ist eine festgelegte Auswahl von XML-Elementen und einem Namensraum. XHTML ist eine XML-Applikation, bei der die XML-Elemente die HTML-Elemente zur Beschreibung von Webseiten sind. Durch die Beschränkung auf eine bestimmte Menge von Elementen ist es möglich, diese XML-Dateien für bestimmte Anwendungen zu nutzen. Der Namensraum legt fest, zu welcher Applikation die einzelnen XML-Elemente gehören. Dadurch können verschiedene XML-Applikationen miteinander kombiniert werden.

Eine bekannte XML-Applikation ist XHTML. Unterschiedliche DTDs beschreiben die Menge möglicher Tags. Für XHTML 1.0 sind es folgende:

- *XHTML1-strict.dtd*: minimale Menge von HTML-Tags
- *XHTML1-transitional.dtd*: die gängigsten HTML-Tags
- *XHTML1-frameset.dtd*: HTML-Tags zur Beschreibung von Frames

Der Standard XHTML 1.1 geht noch einen Schritt weiter und bietet modulare DTDs an. Hier kann sehr genau differenziert werden, welche HTML-Tags für die eigene XML-Applikation gültig sind. Dadurch ist es sehr einfach möglich, XHTML-Elemente mit eige-

nen XML-Elementen zu kombinieren. Durch die Verwendung von Namensräumen können die XHTML- und die XML-Tags zur Datenbeschreibung unterschieden werden.

16.3 Die Java-APIs für XML

Für XML-basierte Daten gibt es vier Verarbeitungstypen:

- **DOM-orientierte APIs** (repräsentieren den XML-Baum im Speicher): *W3C-DOM, JDOM, dom4j, XOM ...*
- **Pull-API** (wie ein Tokenizer wird über die Elemente gegangen): Dazu gehören *XPP* (XML Pull Parser), wie sie der StAX-Standard definiert.
- **Push-API** (nach dem Callback-Prinzip ruft der Parser Methoden auf und meldet Elementvorkommen): *SAX* (Simple API for XML) ist der populäre Repräsentant.
- **Mapping-API** (der Nutzer arbeitet überhaupt nicht mit den Rohdaten einer XML-Datei, sondern bekommt die XML-Datei auf ein Java-Objekt umgekehrt abgebildet): *JAXB, Castor, XStream, ...*

Während DOM das gesamte Dokument in einer internen Struktur einliest und bereitstellt, verfolgt SAX einen ereignisorientierten Ansatz. Das Dokument wird in Stücken geladen, und immer dann, wenn ein angemeldetes Element beim Parser vorbeikommt, meldet er dies in Form eines Ereignisses, das für die Verarbeitung abgefangen werden kann.

Klassische Anwendungen für SAX und StAX sind:

- die Suche nach bestimmten Inhalten
- das Einlesen von XML-Dateien, um eine eigene Datenstruktur aufzubauen

Für einige Anwendungen ist es erforderlich, die gesamte XML-Struktur im Speicher zu verarbeiten. Für diese Fälle ist eine Struktur, wie DOM sie bietet, notwendig:

- Sortierung der Struktur oder einer Teilstruktur der XML-Datei
- Auflösen von Referenzen zwischen einzelnen XML-Elementen
- interaktives Arbeiten mit der XML-Datei

Ob ein eigenes Programm DOM oder StAX einsetzt, ist von Fall zu Fall unterschiedlich. In manchen Fällen ist dies auch Geschmackssache, doch unterscheidet sich das Programmiermodell, sodass eine Umstellung nicht so angenehm ist.

16.3.1 Das Document Object Model (DOM)

DOM ist eine Entwicklung des W3C und wird von vielen Programmiersprachen unterstützt. Das Standard-DOM ist so konzipiert, dass es unabhängig von einer Programmiersprache ist und eine strikte Hierarchie erzeugt. DOM definiert eine Reihe von Schnittstellen, die durch konkrete Programmiersprachen implementiert werden.

16.3.2 Simple API for XML Parsing (SAX)

SAX ist zum schnellen Verarbeiten der Daten von David Megginson als Public Domain entworfen worden. SAX ist im Gegensatz zu DOM nicht so speicherhungrig, weil das XML-Dokument nicht vollständig im Speicher abgelegt ist, und daher auch für sehr große Dokumente geeignet. Da SAX auf einem Ereignismodell basiert, wird die XML-Datei wie ein Datenstrom gelesen, und für erkannte Elemente wird ein Ereignis ausgelöst. Dies ist aber mit dem Nachteil verbunden, dass wahlfreier Zugriff auf ein einzelnes Element nicht ohne Zwischenspeicherung möglich ist.

16.3.3 Pull-API StAX

Im Gegensatz zu SAX, bei dem Methoden bereitgestellt werden, die beim Parsen aufgerufen werden, wird bei der Pull-API wie StAX aktiv der nächste Teil eines XML-Dokuments angefordert. Das Prinzip entspricht dem Iterator-Design-Pattern, das auch von der Collection-API bekannt ist. Es werden die beiden grundsätzlichen Verarbeitungsmodelle *Iterator* und *Cursor* unterschieden. Die Verarbeitung mit dem Iterator ist flexibler, aber auch ein bisschen aufwändiger. Die Cursor-Verarbeitung ist einfacher und schneller, aber nicht so flexibel. Beide Formen sind sich sehr ähnlich. Später werden beide Verfahren vorgestellt.

16.3.4 Java Document Object Model (JDOM)

JDOM ist eine einfache Möglichkeit, XML-Dokumente leicht und effizient mit einer schönen Java-API zu nutzen. Die Entwicklung von JDOM geht von Brett McLaughlin und Jason Hunter aus.

Im Gegensatz zu SAX und DOM, die unabhängig von einer Programmiersprache sind, wurde JDOM speziell für Java entwickelt. Während das Original-DOM keine Rücksicht auf die Java-Datenstrukturen nimmt, nutzt JDOM konsequent die Collection-API. Auch

ermöglicht JDOM eine etwas bessere Performance und eine bessere Speichernutzung als beim Original-DOM.

Warum behandle ich JDOM in diesem Buch?

Das Original-W3C-DOM für Java ist historisch am ältesten, und JDOM war eine der ersten alternativen Java-XML-APIs. Mittlerweile steht JDOM nicht mehr alleine als W3C-DOM-Alternative da, und APIs wie *dom4j* (<http://www.dom4j.org/>) oder *XOM* (<http://www.xom.nu/>) gesellen sich dazu. Obwohl die Entwicklung von JDOM schon lange eingestellt wurde – das letzte Release, JDOM 1.1.1, stammt aus dem Juli 2009 –, zählt JDOM immer noch zu den populärsten³ XML-APIs, wohl auch wegen der üppigen Dokumentation. Hätte ich mich heute für eine Java-XML-DOM-API entscheiden müssen, hätte ich XOM gewählt.



16.3.5 JAXP als Java-Schnittstelle zu XML

Die angesprochenen Technologien wie DOM, SAX, XPath, StAX sind erst einmal pure APIs. Für die APIs sind grundsätzlich verschiedene Implementierungen denkbar, jeweils mit Schwerpunkten wie Performance, Speicherverbrauch, Unicode-4-Unterstützung und so weiter. Zwei Parser-Implementierungen sind zum Beispiel:

- **Xerces** (<http://tutego.de/go/xerces>): Die Standardimplementierung im aktuellen JDK 5. XSL-Stylesheet-Transformationen werden standardmäßig über einen *Compiling XSLT Processor* (XSLTC) verarbeitet.
- **Crimson** (<http://tutego.de/go/crimson>): Die Referenzimplementierung in Java 1.4.

Java API for XML Parsing (JAXP)

Der Nachteil bei der direkten Nutzung der Parser ist die Abhängigkeit von bestimmten Klassen. Daher wurde eine API mit dem Namen *Java API for XML Parsing (JAXP)* entworfen, die als Abstraktionsschicht über folgenden Technologien liegt:

- DOM Level 3
- SAX 2.0.2
- StAX
- XSLT 1.0
- XPath 1.0

³ Laut <http://www.servlets.com/polls/results.tea?name=doms>

Die unterstützten XML-Standards sind 1.0 sowie 1.1. Die Parser validieren mit DTD oder einem W3C XML Schema und können mit *XInclude* Dokumente integrieren. Von DOM wird *DOM Level 3 Core* und *DOM Level 3 Load and Save* unterstützt.

JAXP 1.3 ist Teil von Java 5, und die Version JAXP 1.4 ist Teil von Java 6 und Java 7 – in der Version 1.4 ist nicht viel hinzugekommen. Java 7 aktualisiert nur die Unterversion auf JAXP 1.4.5. Mehr Informationen zu den Versionen und Implementierungen gibt die Webseite <https://jaxp.dev.java.net/>.

Mit JAXP können Entwickler also einfach zwischen verschiedenen Parsern und XSLT-Transformern wählen, ohne den eigentlichen Code zu verändern. Das ist das gleiche Prinzip wie bei den Datenbanktreibern.

16.3.6 DOM-Bäume einlesen mit JAXP *

Um einen DOM-Baum einzulesen, soll unser folgendes Beispiel mit JAXP arbeiten. Eine Fabrik liefert uns einen XML-Parser, sodass wir den DOM-Baum einlesen können:

Listing 16.5: com/tutego/insel/xml/dom/DOMParty.java

```
package com.tutego.insel.xml.dom;

import java.io.File;
import javax.xml.parsers.*;
import org.w3c.dom.Document;

public class DOMParty
{
    public static void main( String[] args ) throws Exception
    {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse( new File("party.xml") );
        System.out.println( document.getFirstChild().gettextContent() );
    }
}
```

Die Parser sind selbstständig bei `DocumentBuilderFactory` angemeldet, und `newInstance()` liefert eine Unterklasse des `DocumentBuilder`.

16.4 Java Architecture for XML Binding (JAXB)

Java Architecture for XML Binding (JAXB) ist eine API zum Übertragen von Objektzuständen auf XML-Dokumente und umgekehrt. Anders als eine manuelle Abbildung von Java-Objekten auf XML-Dokumente oder das Parsen von XML-Strukturen und Übertragen der XML-Elemente auf Geschäftsobjekte arbeitet JAXB automatisch. Die Übertragungsregeln definieren Annotationen, die Entwickler selbst an die JavaBeans setzen können, aber JavaBeans werden gleich zusammen mit den Annotationen von einem Werkzeug aus einer XML-Schema-Datei generiert.

Java 6 integriert JAXB 2.0, und das JDK 6 Update 4 – sehr ungewöhnlich für ein Update – aktualisiert auf JAXB 2.1. Java 7 aktualisiert auf JAXB 2.2.

16.4.1 Bean für JAXB aufbauen

Wir wollen einen Player deklarieren, und JAXB soll ihn anschließend in ein XML-Dokument übertragen:

Listing 16.6: com/tutego/insel/xml/jaxb/Player.java, Player

```
@XmlElement
class Player
{
    private String name;
    private Date   birthday;

    public String getName()
    {
        return name;
    }

    public void setName( String name )
    {
        this.name = name;
    }

    public void setBirthday( Date birthday )
    {
```

```

        this.birthday = birthday;
    }

    public Date getBirthday()
    {
        return birthday;
    }
}

```

Die Klassen-Annotation `@XmlRootElement` ist an der JavaBean nötig, wenn die Klasse das Wurzelement eines XML-Baums bildet. Die Annotation stammt aus dem Paket `javax.xml.bind.annotation`.

16.4.2 JAXBContext und die Marshaller

Ein kleines Testprogramm baut eine Person auf und bildet sie dann in XML ab – die Ausgabe der Abbildung kommt auf den Bildschirm:

Listing 16.7: com/tutego/insel/xml/xml/jaxb/PlayerMarshaller.java, main()

```

Player johnPeel = new Player();
johnPeel.setName( "John Peel" );
johnPeel.setBirthday( new GregorianCalendar(1939,Calendar.AUGUST,30).getTime() );

JAXBContext context = JAXBContext.newInstance( Player.class );
Marshaller m = context.createMarshaller();
m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE );
m.marshal( johnPeel, System.out );

```

Nach dem Lauf erscheint auf dem Schirm:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
    <birthday>1939-08-30T00:00:00+01:00</birthday>
    <name>John Peel</name>
</player>

```

Alles bei JAXB beginnt mit der zentralen Klasse `JAXBContext`. Die statische Methode `JAXBContext.newInstance()` erwartet standardmäßig eine Aufzählung der Klassen, die JAXB

behandeln soll. Der JAXBContext erzeugt den Marshaller zum Schreiben und den Unmarshaller zum Lesen. Die Fabrikmethode createMarshaller() liefert einen Schreiberling, der mit marshal() das Wurzelobjekt in einen Datenstrom schreibt. Das zweite Argument von marshal() ist unter anderem ein OutputStream (wie System.out in unserem Beispiel), Writer oder File-Objekt.

JAXB beachtet standardmäßig alle Bean-Properties, also birthday und name, und nennt die XML-Elemente nach den Properties.

```
class javax.xml.bind.JAXBContext
```

- static JAXBContext newInstance(Class... classesToBeBound) throws JAXBException
Liefert ein Exemplar vom JAXBContext mit Klassen, die als Wurzelklassen für JAXB verwendet werden können.
 - abstract Marshaller createMarshaller()
Erzeugt einen Marshaller, der Java-Objekte in XML-Dokumente konvertieren kann.
- abstract Unmarshaller createUnmarshaller()
Erzeugt einen Unmarshaller, der XML-Dokumente in Java-Objekte konvertiert.

```
class javax.xml.bind.Marshaller
```

- void marshal(Object jaxbElement, File output)
- void marshal(Object jaxbElement, OutputStream os)
- void marshal(Object jaxbElement, Writer writer)
Schreibt den Objektgraph von jaxbElement in eine Datei oder einen Ausgabestrom.
- void marshal(Object jaxbElement, Node node)
Erzeugt vom Objekt einen DOM-Knoten. Der kann dann in ein XML-Dokument gesetzt werden.
- void marshal(Object jaxbElement, XMLEventWriter writer)
- void marshal(Object jaxbElement, XMLStreamWriter writer)
Erzeugt für ein jaxbElement einen Informationsstrom für den XMLEventWriter beziehungsweise XMLStreamWriter. Die StAX-Klassen werden später genauer vorgestellt.
- void setProperty(String name, Object value)
Setzt eine Eigenschaft der Marshaller-Implementierung. Eine Einrückung etwa setzt das Paar Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE.

16.4.3 Ganze Objektgraphen schreiben und lesen

JAXB bildet nicht nur das zu schreibende Objekt ab, sondern auch rekursiv alle referenzierten Unterobjekte. Wir wollen den Spieler dazu in einen Raum setzen und den Raum in XML abbilden. Dazu muss der Raum die Annotation `@XmlRootElement` bekommen, und bei `Player` kann sie entfernt werden, wenn nur der Raum selbst, aber keine `Player` als Wurzelobjekte zum Marshaller kommen:

Listing 16.8: com/tutego/insel/xml/xml/jaxb/Room.java, Room

```
@XmlElement( namespace = "http://tutego.com/" )
public class Room
{
    private List<Player> players = new ArrayList<Player>();

    @XmlElement( name = "player" )
    public List<Player> getPlayers()
    {
        return players;
    }

    public void setPlayers( List<Player> players )
    {
        this.players = players;
    }
}
```

Zwei Annotationen kommen vor: Da `Room` der Start des Objektgraphen ist, trägt es `@XmlRootElement`. Als Erweiterung ist das Element `namespace` für den Namensraum gesetzt, da bei eigenen XML-Dokumenten immer ein Namensraum genutzt werden soll. Weiterhin ist eine Annotation `@XmlElement` am Getter `getPlayers()` platziert, um den Namen des XML-Elements zu überschreiben, damit das XML-Element nicht `<players>` heißt, sondern `<player>`.

Kommen wir abschließend zu einem Beispiel, das einen Raum mit zwei Spielern aufbaut und diesen Raum dann in eine XML-Datei schreibt. Statt allerdings `JAXBContext` direkt zu nutzen und einen Marshaller zum Schreiben und Unmarshaller zum Lesen zu erfragen, kommt im zweiten Beispiel die Utility-Klasse `JAXB` zum Einsatz, die ausschließlich statische überladene `marshal()`- und `unmarshal()`-Methoden anbietet:

Listing 16.9: com/tutego/insel/xml/jaxb/RoomMarshaller.java, main()

```

Player john = new Player();
john.setName( "John Peel" );

Player tweet = new Player();
tweet.setName( "Zwitscher Zoe" );

Room room = new Room();
room.setPlayers( Arrays.asList( john, tweet ) );

File file = new File( "room.xml" );
JAXB.marshal( room, file );

Room room2 = JAXB.unmarshal( file, Room.class );
System.out.println( room2.getPlayers().get( 0 ).getName() ); // John Peel

file.deleteOnExit();

```

Falls etwas beim Schreiben oder Lesen misslingt, werden die vorher geprüften Ausnahmen in einer `DataBindingException` ummantelt, die eine `RuntimeException` ist.

Die Ausgabe ist:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:room xmlns:ns2="http://tutego.com/">
    <player>
        <name>John Peel</name>
    </player>
    <player>
        <name>Zwitscher Zoe</name>
    </player>
</ns2:room>

```

Da beim Spieler das Geburtsdatum nicht gesetzt war (`null` wird referenziert), wird es auch nicht in XML abgebildet.

`class javax.xml.bind.JAXB`

- `static void marshal(Object jaxbObject, File xml)`

- static void marshal(Object jaxbObject, OutputStream xml)
- static void marshal(Object jaxbObject, Result xml)
- static void marshal(Object jaxbObject, String xml)
- static void marshal(Object jaxbObject, URI xml)
- static void marshal(Object jaxbObject, URL xml)
- static void marshal(Object jaxbObject, Writer xml)

Schreibt das XML-Dokument in die angegebene Ausgabe. Im Fall von `URI/URL` wird ein HTTP-POST gestartet. Ist der Parametertyp `String`, wird er als URL gesehen und führt ebenfalls zu einem HTTP-Zugriff. `Result` ist ein Typ für eine XML-Transformation und wird später vorgestellt.

- static <T> T unmarshal(File xml, Class<T> type)
- static <T> T unmarshal(InputStream xml, Class<T> type)
- static <T> T unmarshal(Reader xml, Class<T> type)
- static <T> T unmarshal(Source xml, Class<T> type)
- static <T> T unmarshal(String xml, Class<T> type)
- static <T> T unmarshal(URI xml, Class<T> type)
- static <T> T unmarshal(URL xml, Class<T> type)

Rekonstruiert aus der gegebenen XML-Quelle den Java-Objektgraph.



Performance-Tipp

Den `JAXBContext` aufzubauen, kostet Zeit und Speicher. Er sollte daher für wiederholte Operationen gespeichert werden. Noch eine Information: `Marshaller` und `Unmarshaller` sind nicht thread-sicher; es darf keine zwei Threads geben, die gleichzeitig den `Marshaller`/`Unmarshaller` nutzen.

16.5 XML-Dateien mit JDOM verarbeiten

Über JDOM lassen sich die XML-formatierten Dateien einlesen, manipulieren und dann wieder schreiben. Mit einfachen Aufrufen lässt sich ein Dokument im Speicher erstellen. Zur internen JDOM-Repräsentation werden einige Java-typische Features verwendet, beispielsweise die Collection-API zur Speicherung, Reflection oder schwache Referenzen. Die Nutzung der Collection-API ist ein Vorteil, der unter dem herkömmlichen

DOM nicht zum Tragen kommt. Durch JDOM können mit dem `new`-Operator auch Elemente und Attribute einfach erzeugt werden. Es gibt spezielle Klassen für das Dokument, nämlich Elemente, Attribute und Kommentare. Es sind keine Fabrikschnittstellen, die konfiguriert werden müssen, sondern alles wird direkt erzeugt.

Die Modelle StAX, SAX oder DOM liegen eine Ebene unter JDOM, denn sie dienen als Ausgangspunkt zum Aufbau eines JDOM-Baums. Das heißt, dass ein vorgeschalteter SAX- oder StAX-Parser (bei JDOM *Builder* genannt) die JDOM-Baumstruktur im Speicher erzeugt. Die Bibliothek bietet daher eine neutrale Schnittstelle für diverse Parser, um die Verarbeitung der XML-Daten so unabhängig wie möglich von den Implementierungen zu machen. JDOM unterstützt dabei aktuelle Standards wie DOM Level 3, SAX 2.0 oder XML Schema. Wenn es nötig wird, DOM oder SAX zu unterstützen, bieten Schnittstellen diesen Einstieg an.

Mit JDOM wird auch eine interne Datenstruktur der XML-Datei erzeugt. Dadurch kann jederzeit auf alle Elemente der XML-Datei zugegriffen werden. Da JDOM Java-spezifische Datenstrukturen verwendet, ist die Verarbeitung effizienter als bei DOM. JDOM stellt eine echte Alternative zu DOM dar. Eine Zusammenarbeit von JDOM und SAX ist auch möglich, weil JDOM in der Lage ist, als Ausgabe SAX-Ereignisse auszulösen. Diese können mit SAX-basierten Tools weiterverarbeitet werden. So lässt sich JDOM auch sehr gut in Umgebungen einsetzen, in denen weitere Tools zur Verarbeitung von XML genutzt werden.

16.5.1 JDOM beziehen

Die Webseite <http://www.jdom.org/> bietet Download, Dokumentation und Mailinglisten. Das Zip-Archiv <http://jdom.org/dist/binary/jdom-1.1.1.zip> enthält im *build*-Ordner die Datei *jdom.jar*, die wir dem Klassenpfad hinzufügen. Die API-Dokumentation liegt online unter <http://jdom.org/docs/apidocs/index.html>. JDOM ist freie Software, die auf der Apache-Lizenz beruht. Das heißt, dass JDOM auch in kommerziellen Produkten eingesetzt werden kann, ohne dass diese automatisch Open Source sein müssen.

16.5.2 Paketübersicht *

JDOM besteht aus sieben Paketen mit den Klassen zur Repräsentation des Dokuments, zum Einlesen und Ausgeben, zur Transformation und für XPath-Anfragen.

Das Paket org.jdom

Das Paket `org.jdom` fasst alle Klassen zusammen, um ein XML-Dokument im Speicher zu repräsentieren. Dazu gehören zum Beispiel die Klassen `Attribute`, `Comment`, `CDATA`, `DocType`, `Document`, `Element`, `Entity` und `ProcessingInstruction`. Ein Dokument-Objekt hat ein Wurzelement, eventuell Kommentare, einen `DocType` und eine `ProcessingInstruction`. `Content` ist die abstrakte Basisklasse und Oberklasse von `Comment`, `DocType`, `Element`, `EntityRef`, `ProcessingInstruction` und `Text`. Die Schnittstelle `Parent` implementieren alle Klassen, die `Content` haben können. Viele Schnittstellen gibt es in JDOM nicht. Andere XML-APIs verfolgen bei dieser Frage andere Ansätze; `domj4` definiert zentrale Elemente als Schnittstellen, und die pure DOM-API beschreibt alles über Schnittstellen – konkrete Objekte kommen nur aus Fabriken, und die Implementierung ist unsichtbar.

Die Pakete `org.jdom.output` und `org.jdom.input`

In den beiden Paketen `org.jdom.output` und `org.jdom.input` liegen die Klassen, die XML-Dateien lesen und schreiben können. `XMLOutputter` übernimmt die interne Repräsentation und erzeugt eine XML-Ausgabe in einen `PrintWriter`. Daneben werden die unterschiedlichen Verarbeitungsstrategien DOM und SAX durch die Ausgabeklassen `SAXOutputter` und `DOMOutputter` berücksichtigt. `SAXOutputter` nimmt einen JDOM-Baum und erzeugt benutzerdefinierte SAX2-Ereignisse. Der `SAXOutputter` ist eine sehr einfache Klasse und bietet lediglich eine `output(Document)`-Methode an. Mit `DOMOutputter` wird aus dem internen Baum ein DOM-Baum erstellt.

Ein Builder nimmt XML-Daten in verschiedenen Formaten entgegen und erzeugt daraus ein JDOM-`Document`-Objekt. Das ist bei JDOM der wirkliche Verdienst, dass unabhängig von der Eingabeverarbeitung ein API-Set zur Verfügung steht. Die verschiedenen DOM-Implementierungen unterscheiden sich an manchen Stellen. Die Schnittstelle `Builder` wird von allen einlesenden Klassen implementiert. Im `Input`-Paket befinden sich dafür die Klassen `DOMBuilder`, die einen JDOM-Baum mit DOM erzeugt, und `SAXBuilder`, die dafür SAX verwendet. Damit kann das Dokument aus einer Datei, einem Stream oder einer URL erzeugt werden. Nach dem Einlesen sind die Daten vom konkreten Parser des Herstellers unabhängig und können weiterverarbeitet werden. `SAXBuilder` ist schneller und speicherschonender. Ein `DOMBuilder` wird meistens nur dann benutzt, wenn ein DOM-Baum weiterverarbeitet werden soll. `org.jdom.input.StAXBuilder` ist eine Klasse aus dem Hilfspaket unter <http://tutego.de/go/staxmisc>.

Im `org.jdom.contrib`-Package gibt es noch einige Erweiterungen für JDOM. Eine bemerkenswerte Erweiterung ist der `ResultSetBuilder`. Diese Klasse ermöglicht das Erstellen einer JDOM-Datenstruktur anhand eines `java.sql.ResultSet`. Dadurch ist eine Brücke

zwischen Datenbanken und XML sehr einfach zu realisieren. Diese und noch viele weitere nützliche Erweiterungen sind nicht in der JDOM-Standarddistribution enthalten, sondern im Contrib-Paket.

Das Paket org.jdom.transform

Mit dem Paket `org.jdom.transform` wird das JAXP-TraX-Modell in JDOM integriert. Dies ermöglicht es JDOM, XSLT-Transformationen von XML-Dokumenten zu unterstützen. Das Paket enthält die beiden Klassen `JDOMResult` und `JDOMSource`. Die Klasse `JDOMSource` ist eine Wrapper-Klasse, die ein JDOM-Dokument als Parameter nimmt und diesen als Eingabe für das JAXP-TraX-Modell bereitstellt. Die Klasse `JDOMResult` enthält das Ergebnis der Transformation als JDOM-Dokument. Die beiden Klassen haben nur wenige Methoden, und in der API sind Beispiele für die Benutzung dieser Klassen angegeben.

Das Paket org.jdom.xpath

Im Paket `org.jdom.xpath` befindet sich nur eine Utility-Klasse `XPath`. Diese Klasse bildet die Basis für die Verwendung der Abfragesprache XPath mit JDOM. Neben der Implementierung, die mit JDOM geliefert wird, kann auch eine spezielle Implementierung der XPath-Methoden für JDOM eingesetzt werden. JDOM bringt keine eigene XPath-Implementierung mit, sondern basiert auf der Open-Source-Implementierung *Jaxen* (<http://jaxen.org/>).

Das Paket org.jdom.adapters

Die Klassen des Pakets `org.jdom.adapters` erlauben es JDOM, existierende DOM-Implementierungen zu nutzen. Sie sind nur interessant für diejenigen, die selbst einen XML-Parser an JDOM anpassen wollen.

16.5.3 Die Document-Klasse

Dokumente werden bei JDOM über die Klasse `Document` verwaltet. Ein Dokument besteht aus einem `DocType`, einer `ProcessingInstruction`, einem Wurzelement und Kommentaren. Die Klasse `Document` gibt es auch in der Standardschnittstelle für das DOM. Falls sowohl JDOM als auch DOM verwendet werden, muss für die Klasse `Document` der voll qualifizierte Klassename mit vollständiger Angabe der Pakete verwendet werden, weil sonst nicht klar ist, welche `Document`-Klasse verwendet wird.

Ein JDOM-Document im Speicher erstellen

Um ein Document-Objekt zu erzeugen, bietet die Klasse drei Konstruktoren an. Über einen Standard-Konstruktor erzeugen wir ein leeres Dokument. Dieses können wir später bearbeiten, indem wir zum Beispiel Elemente (Objekte vom Typ Element), Entitäten oder Kommentare einfügen. Ein neues Dokument mit einem Element erhalten wir über einen Konstruktor, zu dem wir ein Wurzelement angeben. Jedes XML-Dokument hat ein Wurzelement.

zB Beispiel

Die folgende Zeile erzeugt ein JDOM-Dokument mit einem Wurzelement:

Listing 16.10: com/tutego/insel/xml/jdom/CreateRoot.java, main()

```
Document doc = new Document( new Element("party") );
```

In XML formatiert, könnte das so aussehen:

```
<party>
</party>
```

16.5.4 Eingaben aus der Datei lesen

Ein zweiter Weg, um ein JDOM-Dokument anzulegen, führt über einen Eingabestrom oder eine Datei. Dafür benötigen wir einen Builder, zum Beispiel den SAXBuilder (den wir bevorzugen wollen).

zB Beispiel

Lies die Datei *party.xml* ein:

Listing 16.11: com/tutego/insel/xml/jdom/ReadXmlFile.java, main()

```
String filename = "party.xml";
Document doc = new SAXBuilder().build( filename );
```

Die möglichen Ausnahmen `IOException` und `JDOMException` muss die Anwendung abfangen.

Die Klasse Document bietet selbst keine Lese-Methoden. Es sind immer die Builder, die Document-Objekte liefern. Es ist ebenso möglich, ein JDOM-Dokument mithilfe des DOM-Parsers über DOMBuilder zu erzeugen. Neben den Standard-Konstruktoren bei SAXBuilder

und `DOMBuilder` lässt sich unter anderem ein `boolean`-Wert angeben, der die Validierung auf wohldefinierten XML-Code einschaltet.

Tipp



Wenn ein DOM-Baum nicht schon vorliegt, ist es sinnvoll, ein JDOM-Dokument stets mit dem SAX-Parser zu erzeugen. Das schont die Ressourcen und geht viel schneller, weil keine spezielle Datenstruktur für den DOM-Baum erzeugt werden muss. Das Ergebnis ist in beiden Fällen ein JDOM-Dokument, das die XML-Datei in einer baumähnlichen Struktur abbildet.

```
class org.jdom.input.SAXBuilder  
implements Parent
```

- `SAXBuilder()`
Baut einen XML-Leser auf Basis von SAX auf. Es wird nicht validiert.
- `SAXBuilder(boolean validate)`
Baut einen validierenden `SAXBuilder` auf.
- `Document build(File file)`
- `Document build(InputSource in)`
- `Document build(InputStream in)`
- `Document build(InputStream in, String systemId)`
- `Document build(Reader characterStream)`
- `Document build(Reader characterStream, String systemId)`
- `Document build(String systemId)`
- `Document build(URL url)`
Baut ein JDOM-Dokument aus der gegebenen Quelle auf. Im Fall des `String`-Arguments handelt es sich um einen URI-Namen und nicht um ein XML-Dokument im `String`.

16.5.5 Das Dokument im XML-Format ausgeben

Mit einem `XMLOutputter` lässt sich der interne JDOM-Baum als XML-Datenstrom in einen `OutputStream` oder `Writer` schieben.

zB Beispiel

Gib das JDOM-Dokument auf der Konsole aus:

Listing 16.12: com/tutego/insel/xml/jdom/ReadXmlFile.java, main()

```
XMLOutputter out = new XMLOutputter();
out.output( doc, System.out );
```

Die Standard-Parametrisierung des Formatierers schreibt die XML-Daten mit schönen Einrückungen. Jeder Eintrag kommt in eine einzelne Zeile. Weitere Anpassungen der Formatierung übernimmt ein org.jdom.output.Format-Objekt. Einige statische Methoden bereiten Format-Objekte mit unterschiedlichen Belegungen vor, so getPrettyFormat() für hübsch eingerückte Ausgaben und getCompactFormat() mit sogenannter Leerraum-Normalisierung, wie es die API-Dokumentation nennt.

```
XMLOutputter out = new XMLOutputter( Format.getPrettyFormat() );
out.output( doc, System.out );
```

Unterschiedliche setXXX()-Methoden auf dem XMLOutputter-Objekt ermöglichen eine weitere individuelle Anpassung der Format-Objekte. Soll das Ergebnis als String vorliegen, kann outputString() verwendet werden, was ein String-Objekt liefert.

16.5.6 Elemente

Jedes Dokument besteht aus einem Wurzelement. Wir haben schon gesehen, dass dieses durch die allgemeine Klasse Element abgebildet wird. Mit dem Wurzelement gelingt der Zugriff auf die anderen Elemente des Dokumentenbaums.

Wurzelement

Die folgenden Beispieldateien verwenden die XML-Datei *party.xml*, um die Methoden von JDOM vorzustellen. Durch das Erzeugen eines leeren JDOM-Dokuments und die Methoden zur Erstellung von Elementen und Attributen kann JDOM den Dateinhalt auch leicht aufbauen:

Listing 16.13: party.xml

```
<party datum="31.12.01">
  <gast name="Albert Angsthase">
    <getraenk>Wein</getraenk>
```

```

<getraenk>Bier</getraenk>
<zustand ledig="true" nuechtern="false"/>
</gast>
<gast name="Martina Mutig">
    <getraenk>Apfelsaft</getraenk>
    <zustand ledig="true" nuechtern="true"/>
</gast>
<gast name="Zacharias Zottelig"></gast>
</party>
```

Um an das Wurzelement `<party>` zu gelangen und von dort aus weitere Elemente oder Attribute auslesen zu können, erzeugen wir zunächst ein JDOM-Dokument aus der Datei `party.xml` und nutzen zum Zugriff `getRootElement()`.

Beispiel

zB

Lies die Datei `party.xml`, und erfrage das Wurzelement:

Listing 16.14: com/tutego/insel/xml/jdom/RootElement.java, main()

```
Document doc = new SAXBuilder().build( "party.xml" );
Element party = doc.getRootElement();
```

16

```
class org.jdom.Document
implements Parent
```

- `Element getRootElement()`
Gibt das Root-Element zurück oder `null`, falls kein Root-Element vorhanden ist.
- `boolean isRootElement()`
Gibt einen Wahrheitswert zurück, der ausdrückt, ob das Element die Wurzel der JDOM-Datenstruktur ist.

Durch die oben gezeigten Anweisungen wird aus der XML-Datei `party.xml` eine JDOM-Datenstruktur im Speicher erzeugt. Um mit dem Inhalt der XML-Datei arbeiten zu können, ist der Zugriff auf die einzelnen Elemente notwendig. Durch die Methode `getRootElement()` wird das Wurzelement der XML-Datei zurückgegeben. Dieses Element ist der Ausgangspunkt für die weitere Verarbeitung der Datei.

Zugriff auf Elemente

Um ein bestimmtes Element zu erhalten, gibt es die Methode `getChild(String name)`. Mit dieser Methode wird das nächste Unterelement des Elements zurückgegeben, das diesen Namen trägt.

zB Beispiel

Wenn wir den ersten Gast auf der Party haben möchten, schreiben wir:

Listing 16.15: com/tutego/insel/xml/jdom/AlbertTheFirst.java, main()

```
Element party = doc.getRootElement();
Element albert = party.getChild( "gast" );
```

Wenn wir wissen wollen, was Albert trinkt, schreiben wir:

```
Element albertGetraenck = albert.getChild( "getraenck" );
```

Durch eine Kaskadierung ist es möglich, über das Wurzelement auf das Getränk des ersten Gastes zuzugreifen:

```
Element albertGetraenck = party.getChild( "gast" ).getChild( "getraenck" );
```

Eine Liste mit allen Elementen liefert die Methode `getChildren()`. Sie gibt eine nicht generisch verwendete `java.util.List` mit allen Elementen dieses Namens zurück.⁴

zB Beispiel

Falls wir eine Gästeliste der Party haben wollen, schreiben wir:

```
List<?> gaeste = party.getChildren( "gast" );
```

Diese Liste enthält alle Elemente der Form `<gast ...> ... </gast>`, die direkt unter dem Element `<party>` liegen.

⁴ JDOM ist vor Java 5 entwickelt worden, und die Entwickler investieren keine Zeit mehr in die Weiterentwicklung. <http://code.google.com/p/coffeedom/> ist eine Entwicklung von anderer Stelle, die Generics einführt. <http://git.tuis.net/jdom/> ist ein alternativer Port. XOM nutzt nach außen keine Typen der Collection-API, denn spezielle XOM-Klassen wie `Nodes` und `Elements` sind Container und liefern typisierte Objekte. Leider implementieren sie keine neuen Java-Collection-API-Klassen wie `Iterable`. `dom4j` ist die einzige bekanntere XML-Bibliothek, die Generics einsetzt.

```
class org.jdom.Element
extends Content
implements Parent
```

- `Element getChild(String name)`
Rückgabe des ersten untergeordneten Elements mit dem lokalen Namen `name`, das keinem Namensraum zugeordnet ist.
- `Element getChild(String name, Namespace ns)`
Rückgabe des ersten untergeordneten Elements mit dem lokalen Namen `name`, das dem Namensraum `ns` zugeordnet ist.
- `List getChildren()`
Rückgabe einer Liste der Elemente, die diesem Element direkt untergeordnet sind. Falls keine Elemente existieren, wird eine leere Liste zurückgegeben. Änderungen an der Liste spiegeln sich auch in der JDOM-Datenstruktur wider.
- `List getChildren(String name)`
Rückgabe einer Liste der Elemente mit dem Namen `name`, die diesem Element direkt untergeordnet sind. Falls keine Elemente existieren, wird eine leere Liste zurückgegeben. Änderungen an der Liste spiegeln sich auch in der JDOM-Datenstruktur wider.
- `List getChildren(String name, Namespace ns)`
Rückgabe einer Liste der Elemente mit dem Namen `name`, die diesem Namensraum zugeordnet und diesem Element direkt untergeordnet sind. Falls keine Elemente existieren, wird eine leere Liste zurückgegeben. Änderungen an der Liste spiegeln sich auch in der JDOM-Datenstruktur wider.
- `boolean hasChildren()`
Rückgabe eines `boolean`-Werts, der ausdrückt, ob Elemente untergeordnet sind oder nicht.

16.5.7 Zugriff auf Elementinhalte

Von Beginn eines Elements bis zu dessen Ende treffen wir auf drei unterschiedliche Informationen:

- Es können weitere Elemente folgen. Im oberen Beispiel folgt in `<gast>` noch ein Element `<getraenk>`.
- Das Element enthält Text (wie das Element `<getraenk>`).

- Zusätzlich kann ein Element auch Attribute beinhalten. Dies haben wir auch beim Element <gast> gesehen, das als Attribut den Namen des Gasts enthält. Der Inhalt von Attributen ist immer Text.

Für diese Aufgaben bietet die `Element`-Klasse unterschiedliche Anfrage- und Setzermethoden. Wir wollen mit dem Einfachsten, dem Zugriff auf den Textinhalt eines Elements, beginnen.

Elementinhalte auslesen und setzen

Betrachten wir das Element, dessen Inhalt wir auslesen wollen, so nutzen wir dazu die Methode `getText()`:

```
<getraenk>Wein</getraenk>
```

Sie liefert einen String, sofern eine String-Präsentation des Inhalts erlaubt ist. Falls das Element keinen Text oder nur Unterelemente besitzt, ist der Rückgabewert ein Leerstring.

zB Beispiel

Um an das erste Getränk von Albert zu kommen, schreiben wir:

Listing 16.16: com/tutego/insel/xml/jdom/AlbertsDrink.java, main()

```
Element party = doc.getRootElement();
Element albertGetraenk = party.getChild( "gast" ).getChild( "getraenk" );
String getraenk = albertGetraenk.getText();
```

```
class org.jdom.Element
extends Content
implements Parent
```

- `String getText()`

Rückgabe des Inhalts des Elements. Dies beinhaltet alle Leerzeichen und CDATA-Sektionen. Falls der Elementinhalt nicht zurückgegeben werden kann, wird der leere String zurückgegeben.

- `String getTextNormalize()`

Verhält sich wie `getText()`. Leerzeichen am Anfang und am Ende des Strings werden entfernt. Leerzeichen innerhalb des Strings werden auf ein Leerzeichen normalisiert. Falls der Text nur aus Leerzeichen besteht, wird der leere String zurückgegeben.

- `String getTextTrim()`

Verhält sich wie `getTextNormalize()`. Leerzeichen innerhalb des Strings bleiben erhalten.

Für die Methode `getText()` muss das Element vorliegen, dessen Inhalt gelesen werden soll. Mit der Methode `getChildText()` kann der Inhalt eines untergeordneten Elements auch direkt ermittelt werden.

Beispiel

zB

Lies den Text des ersten untergeordneten Elements mit dem Namen `getraenk`. Das übergeordnete Element von `Getränk` ist `albert`:

Listing 16.17: com/tutego/insel/xml/jdom/AlbertsDrink.java, main()

```
Element albert = party.getChild( "gast" );
String getraenk = albert.getChildText( "getraenk" );
```

In der Implementierung der Methode `getChildText()` sind die Methoden `getChild()` und `getText()` zusammengefasst.

```
class org.jdom.Element
extends Content
implements Parent
```

16

- `String getChildText(String name)`

Rückgabe des Inhalts des Elements mit dem Namen `name`. Falls der Inhalt kein Text ist, wird ein leerer String zurückgegeben. Falls das Element nicht existiert, wird `null` zurückgegeben.

- `String getChildText(String name, Namespace ns)`

Verhält sich wie `getChildText(String)` im Namensraum `ns`.

- `String getChildTextTrim(String name)`

Verhält sich wie `getChildText(String)`. Leerzeichen am Anfang und am Ende des Strings werden entfernt. Leerzeichen innerhalb des Strings bleiben erhalten.

- `String getChildTextTrim(String name, Namespace ns)`

Verhält sich wie `getChildTextTrim(String)` im Namensraum `ns`.

- `String getName()`

Rückgabe des lokalen Namens des Elements ohne Namensraum-Präfix.

- `Namespace getNamespace()`
Rückgabe des Namensraums oder eines leeren Strings, falls diesem Element kein Namensraum zugeordnet ist.
- `Namespace getNamespace(String prefix)`
Rückgabe des Namensraums des Elements mit diesem Präfix. Dies beinhaltet das Hochlaufen in der Hierarchie des JDOM-Dokuments. Falls kein Namensraum gefunden wird, gibt diese Methode `null` zurück.
- `String getNamespacePrefix()`
Rückgabe des Namensraum-Präfixes. Falls kein Namensraum-Präfix existiert, wird ein Leerstring zurückgegeben.
- `String getNamespaceURI()`
Rückgabe der Namensraum-URI, die dem Präfix dieses Elements zugeordnet ist, oder des Standardnamensraums. Falls keine URI gefunden werden kann, wird ein leerer String zurückgegeben.

16.5.8 Attributinhalte lesen und ändern

Ein Element kann auch einen Attributwert enthalten. Dies ist der Wert, der direkt in dem Tag mit angegeben ist. Betrachten wir dazu folgendes Element:

```
<gast name="Albert Angsthase">
```

Das Element hat als Attribut `name="Albert Angsthase"`. Diesen Wert liefert die Methode `getAttribute(String).getValue()` der Klasse `Element`.

zB

Beispiel

Lies den Namen des ersten Gastes:

Listing 16.18: com/tutego/insel/xml/jdom/Wedding, main()

```
Element party      = doc.getRootElement();
Element albert     = party.getChild( "gast" );
Attribute albertAttr = albert.getAttribute( "name" );
String albertName   = albert.getAttribute( "name" ).getValue();
```

Martina möchte wissen, ob Albert noch ledig ist:

```
albert.getChild( "zustand" ).getAttribute( "ledig" ).getValue();
```

Auf ähnliche Weise lässt sich der Wert eines Attributs ändern. Dazu gibt es die Methoden `setAttribute(String)` der Klasse `Attribute` und `addAttribute(Attribute)` der Klasse `Element`.

Beispiel

zB

Martina und Albert haben geheiratet, und Albert nimmt den Namen von Martina an:

```
albert.getAttribute( "name" ).setAttribute( "Albert Mutig" );
```

Seit der Hochzeit mit Albert trinkt Martina auch Wein. Also muss ein neues Element `wein` unter dem Element `<gast name="Martina Mutig">` eingefügt werden. Zuerst erzeugen wir ein Element der Form `<getraenk>Wein</getraenk>`:

```
Element wein = new Element( "getraenk" );
wein.addContent( "Wein" );
```

Danach suchen wir Martina in der Gästeliste und fügen das Element `<wein>` ein:

```
Iterator<?> gaesteListe = party.getChildren( "gast" ).iterator();
while ( gaesteListe.hasNext() )
{
    Element gast = (Element) gaesteListe.next();

    if ( "Martina Mutig".equals(
        gast.getAttribute( "name" ).getValue() ) )
        gast.addContent( wein );
}
```

16

Das Beispiel macht deutlich, wie flexibel die Methode `addContent(Inhalt)` ist. Es zeigt ebenso, wie JDOM für Java, etwa durch die Implementierung der Schnittstelle `List`, optimiert wurde.

```
class org.jdom.Element
extends Content
implements Parent
```

- `Attribute getAttribute(String name)`

Rückgabe des Attributs mit dem Namen `name`, das keinem Namensraum zugeordnet ist. Falls das Element kein Attribut mit dem Namen `name` hat, ist die Rückgabe `null`.

- Attribute.getAttribute(String name, Namespace ns)
Verhält sich wie `getAttribute(String)` in dem Namensraum ns.
- List getAttributes()
Rückgabe einer Liste aller Attribute eines Elements oder einer leeren Liste, falls das Element keine Attribute hat.
- String getAttributeValue(String name)
Rückgabe des Attributwerts mit dem Namen `name`, dem kein Namensraum zugeordnet ist. Es wird `null` zurückgegeben, falls keine Attribute dieses Namens existieren, und der leere String, falls der Wert des Attributs leer ist.
- String getAttributeValue(String name, Namespace ns)
Verhält sich wie `getAttributeValue(String)` in dem Namensraum ns.
- Element setAttributes(List attributes)
Fügt alle Attribute der Liste dem Element hinzu. Alle vorhandenen Attribute werden entfernt. Das geänderte Element wird zurückgegeben.
- Element addAttribute(Attribute attribute)
Einfügen des Attributs `attribute`. Bereits vorhandene Attribute mit gleichem Namen und gleichem Namensraum werden ersetzt.
- Element addAttribute(String name, String value)
Einfügen des Attributs mit dem Namen `name` und dem Wert `value`. Um Attribute mit einem Namensraum hinzuzufügen, sollte die Methode `addAttribute(Attribute attribute)` verwendet werden.

```
class org.jdom.Attribute
    implements Serializable, Cloneable
```

- String getValue()
Rückgabe des Werts dieses Attributs

Die folgenden Methoden versuchen eine Umwandlung in einen primitiven Datentyp. Falls eine Umwandlung nicht möglich ist, wird eine `DataConversionException` ausgelöst.

- `getBooleanValue()`
Gibt den Wert des Attributs als `boolean` zurück.
- `double getDoubleValue()`
Gibt den Wert des Attributs als `double` zurück.
- `float getFloatValue()`
Gibt den Wert des Attributs als `float` zurück.

- `int getIntValue()`
Gibt den Wert des Attributs als `int` zurück.
- `long getLongValue()`
Gibt den Wert des Attributs als `long` zurück.
- `String getName()`
Gibt den lokalen Namen des Attributs zurück. Falls der Name die Form `[namespacePrefix]:[elementName]` hat, wird `[elementName]` zurückgegeben. Wenn der Name kein Namensraum-Präfix hat, wird einfach nur der Name ausgegeben.
- `Namespace getNamespace()`
Gibt den Namensraum des Attributs zurück. Falls kein Namensraum vorhanden ist, wird das konstante Namensraum-Objekt `NO_NAMESPACE` zurückgegeben. Diese Konstante enthält ein Namensraum-Objekt mit dem leeren String als Namensraum.
- `String getNamespacePrefix()`
Gibt das Präfix des Namensraums zurück. Falls kein Namensraum zugeordnet ist, wird ein leerer String zurückgegeben.
- `String getNamespaceURI()`
Gibt die URI zurück, die zu dem Namensraum dieses Elements gehört. Falls kein Namensraum zugeordnet ist, wird ein leerer String zurückgegeben.
- `Element getParent()`
Gibt das Element zurück, das dem Element dieses Attributs übergeordnet ist. Falls kein übergeordnetes Element vorhanden ist, wird `null` zurückgegeben.
- `String getQualifiedName()`
Rückgabe des qualifizierten Namens des Attributs. Falls der Name die Form `[namespacePrefix]:[elementName]` hat, wird dies zurückgegeben. Ansonsten wird der lokale Name zurückgegeben.
- `Attribute setValue(String value)`
Setzt den Wert dieses Attributs.

16.6 Zum Weiterlesen

»Java 7 – Mehr als eine Insel« stellt weitere XML-Verarbeitungsformen vor. Dazu zählen StAX und SAX als Pull- und Push-Parser und Anfragen mit XPath sowie Transformationen mit XSLT.



Kapitel 17

Einführung ins Datenbankmanagement mit JDBC

»Alle Entwicklung ist bis jetzt nichts weiter als ein Taumeln von einem Irrtum in den anderen.«

– Henrik Ibsen (1828–1906)

Das Sammeln, Zugreifen auf und Verwalten von Informationen ist im »Informationszeitalter« für die Wirtschaft eine der zentralen Säulen. Während früher Informationen auf Papier gebracht wurden, bietet die EDV hierfür Datenbankverwaltungssysteme (DBMS, engl. *database management systems*) an. Diese arbeiten auf einer *Datenbasis*, also auf Informationseinheiten, die miteinander in Beziehung stehen. Die Programme, die die Datenbasis kontrollieren, bilden die zweite Hälfte der DBMS. Die Netzwerk- oder hierarchischen Datenmodelle sind mittlerweile den relationalen Modellen – kurz gesagt, Tabellen, die miteinander in Beziehung stehen – gewichen.

Mittlerweile gibt es neben den relationalen Modellen auch andere Speicherformen für Datenbanken. Immer populärer werden objektorientierte Datenbanken und XML-Datenbanken. Auch mit ihnen werden wir uns kurz beschäftigen.

17

17.1 Relationale Datenbanken

17.1.1 Das relationale Modell

Die Grundlage für relationale Datenbanken sind Tabellen mit ihren Spalten und Zeilen. In der Vertikalen sind die Spalten und in der Horizontalen die Zeilen angegeben. Eine *Zeile* (auch *Tupel* genannt) entspricht einem Element einer Tabelle, eine *Spalte* (auch *Attribut* genannt) einem Eintrag einer Tabelle.

Lfr_Code	Lfr_Name	Adresse	Wohnort
004	Hoven G. H.	Sandweg 50	Linz
009	Baumgarten R.	Tankstraße 23	Hannover
011	Strauch GmbH	Beerenweg 34a	Linz
013	Spitzmann	Hintergarten 9	Aalen
...

Tabelle 17.1: Eine Beispieltabelle

Jede Tabelle entspricht einer logischen Sicht des Benutzers. Die Zeilen einer Relation stellen die *Datenbankausprägung* dar, während das *Datenbankschema* die Struktur der Tabelle – also Anzahl, Name und Typ der Spalten – beschreibt.

Wenn wir nun auf diese Tabellen Zugriff erhalten wollen, um damit die Datenbankausprägung zu erfahren, benötigen wir Abfragemöglichkeiten. Java erlaubt mit JDBC den Zugriff auf relationale Datenbanken.

17.2 Einführung in SQL

SQL ist die bedeutendste Abfragesprache für relationale Datenbanken, in der Benutzer angeben, auf welche Daten sie zugreifen möchten.¹ Obgleich die Bezeichnung »Abfragesprache« etwas irreführend klingt, beinhaltet SQL auch Befehle zur Datenmanipulation und Datendefinition, um beispielsweise neue Tabellen zu erstellen. Nachdem Anfang der 1970er-Jahre das relationale Modell für Datenbanken populär geworden war, entstand im IBM-Forschungslabor San José (jetzt Almaden) ein Datenbanksystem namens *System R*. Das relationale Modell wurde 1970 von Dr. Edgar F. Codd² entwickelt. System R bot eine Abfragesprache, die SEQUEL (*Structured English Query Language*) genannt wurde. Später wurde SEQUEL in SQL umbenannt. Da sich relationale Systeme großer Beliebtheit erfreuten, wurde 1986 die erste SQL-Norm vom ANSI-Konsortium verabschiedet. 1988 wurde der Standard geändert, und 1992 entstand die zweite Version von SQL (SQL-2 beziehungsweise SQL-92 genannt). Da die wichtigen Datenbanken alle SQL-2 verarbeiten, kann ein Programm über diese Befehle die Datenbank steuern, ohne ver-

¹ Bei einer OODB ist dies OQL und bei XML-Datenbanken in der Regel XQuery.

² 1981 erhielt er für seine Arbeit mit relationalen Datenbanken den Turing-Award – eine Art Nobelpreis für Informatiker.

schiedene proprietäre Schnittstellen nutzen zu müssen. Dennoch können über SQL die speziellen Leistungen einer Datenbank genutzt werden.

Sprache	Entwicklung
SQUARE	1975
SEQUEL	1975, IBM Research Labs San José
SEQUEL2	1976, IBM Research Labs San José
SQL	1982, IBM
ANSI-SQL	1986
ISO-SQL (SQL 89, SQL-1)	1989, drei Sprachen: Level 1, Level 2, +IEF
SQL-2 (bzw. SQL-92)	1992
SQL-3 (auch SQL-1999)	1999

Tabelle 17.2: Entwicklung von SQL

Damit sich ein Datenbanktreiber JDBC-kompatibel nennen kann, muss er mindestens SQL-92 unterstützen. Das bedeutet jedoch nicht, dass die existierenden Treiber alle Eigenschaften von SQL-92 implementieren müssen; über Kannst-du-Methoden lässt sich der Treiber fragen, ob er Eigenschaften unterstützt oder nicht.

17.2.1 Ein Rundgang durch SQL-Abfragen

Da das Wort »Abfragesprache« eine Art Programmiersprache suggeriert, sind wir an einem Beispiel interessiert. Um es vorwegzusagen: Es gibt nur eine Handvoll wichtiger Befehle, und SELECT, UPDATE und CREATE decken schon einen Großteil davon ab. Obwohl die Groß- und Kleinschreibung der Befehle unbedeutend ist, sind sie zwecks besserer Lesbarkeit im Folgenden großgeschrieben.

Beispiel

Eine einfache Abfrage in SQL:

```
SELECT Lfr_Name
FROM Lieferanten
```

Die Tabellen und Spalten sind auch für die folgenden Beispiele fiktiv.

Tabellen nehmen die Benutzerdaten auf, und mit dem Kommando `FROM` wählen wir die Tabelle `Lieferanten` aus, die für die Berechnung erforderlich ist. Die Tabelle `Lieferanten` enthält drei Attribute (die Spalten), die wir mit `SELECT` auswählen. In einer Datenbank werden normalerweise mehrere Tabellen verwendet, ein sogenanntes *Datenbankschema*. Jede Tabelle gehört zu genau einem Schema.

SQL-Abfragen sind nahe an der Umgangssprache formuliert. Im oberen Beispiel liest sich die Abfrage einfach als: »Wähle die Spalte `Lfr_Name` aus der Tabelle `Lieferanten`«. Der Designer einer Datenbank muss sich natürlich vor der Umsetzung der Tabellen und somit der Relationen gründlich Gedanken machen. Eine spätere Änderung der Struktur wird nämlich teuer. So muss schon am Anfang einkalkuliert werden, welche Daten in welchen Ausprägungen auftreten können. Nach Statistiken der amerikanischen *Library of Congress* verdoppelt sich insgesamt alle fünf Jahre die Informationsmenge. Was wäre, wenn diese Informationen alles Einträge in einer endlos verzweigten Datenbank wären und jemand feststellen würde, dass das Tabellenschema ungünstig ist? Datenbankdesigner nennen den Vorgang von einem ersten Modell bis zur fertigen Relation *Normalisierung*.

Bevor wir mit den einzelnen Sprachelementen von SQL fortfahren, sind an dieser Stelle einige Regeln für SQL-Ausdrücke zusammengefasst:

- Die SQL-Anweisungen sind unabhängig von der Groß- und Kleinschreibung. Im Text sind die SQL-Kommandos großgeschrieben, damit die Ausdrücke besser lesbar sind.
- Leerzeichen, Return und Tabulatoren sind in einer Abfrage bedeutungslos. Im Folgenden werden zur besseren Lesbarkeit Zeilenumbrüche verwendet.
- SQL-Anweisungen werden mit einer Zeichenkette abgeschlossen. Sie unterscheidet sich aber von Datenbank zu Datenbank. Häufig ist dies ein Semikolon; es kann aber auch ein `\go` sein. Wir werden die Anweisungen in den Beispielen nicht abschließen, da JDBC diesen Abschluss automatisch vornimmt.

Wir werden uns im Folgenden etwas intensiver um SQL-Abfragen kümmern. Es zeigt sich, dass eine einzelne SQL-Anweisung sehr ausdrucksstark sein kann. JDBC hat mit dieser Ausdrucksstärke aber nichts zu schaffen, es kennt nicht einmal ihre Korrektheit. JDBC leitet den SQL-Befehl einfach an den Treiber, und dieser leitet das Kommando an die Datenbank weiter.

SQL gliedert sich in eine Reihe von unterschiedlichen Abfragetypen – die einen sind mehr mit der Modifikation von Daten, die anderen eher mit deren Abfrage beschäftigt. Anbei die wichtigsten drei Sprachen:

- **DDL (Data Definition Language):** Erstellen der Tabellen, Beziehungen (mit Schlüsseln) und Indizes. Typische SQL-Anweisungen sind CREATE/DROP DATABASE, CREATE/DROP INDEX, CREATE/DROP SYNONYM, CREATE/DROP TABLE und CREATE/DROP VIEW.
- **DML (Data Manipulation Language):** Daten hinzufügen und löschen. Typische SQL-Anweisungen sind hier INSERT, DELETE und UPDATE.
- **DQL (Data Query Language):** Daten auswählen und filtern. Die typischste SQL-Anweisung ist SELECT mit den Spezialisierungen ALL, DISTINCT, ORDER BY, GROUP BY, HAVING, Unterabfragen (IN, ANY, ALL, EXISTS), Schnittmengen und Joins.

17.2.2 Datenabfrage mit der Data Query Language (DQL)

Die DQL umfasst Abfragekommandos, um auf die Inhalte zuzugreifen. In SQL steckt auch schon das Wort »Query«, unsere Abfrage. Das wichtigste Element ist hierbei das oben bereits erwähnte SELECT:

```
SELECT {Feldname, Feldname,...|*} ( * = alle Felder )
FROM Tabelle [, Tabelle, Tabelle....]
[WHERE {Bedingung}]
[ORDER BY Feldname [ASC|DESC]...]
[GROUP BY Feldname [HAVING {Bedingung}]]
```

Das Angenehme an SQL ist die Tatsache, dass wir uns nicht um das *Wie* kümmern müssen, sondern nur um das *Was*. Wir fragen also etwa: »Welche Lieferanten wohnen in Aalen?« und formulieren:

```
SELECT Lfr_Name
FROM Lieferanten
WHERE Wohnort='Aalen'
```

Dabei ist es uns egal, wie die Datenbankimplementierung mit dieser Abfrage umgeht. Hier unterscheiden sich auch die verschiedenen Datenbanken in ihrer Leistungsfähigkeit und in den Preisen.

Kümmern wir uns nun um die verschiedenen Schreibweisen von SELECT. Geben wir in SELECT einen Spaltennamen oder Spaltenindex an, so bekommen wir nur die Ergebnisse dieser Spalte zurück. Eine Abfrage mit »*« liefert alle Spalten zurück. Damit wir also nicht nur den Namen des Kunden bekommen, sondern auch die anderen Angaben – um ihm gleich einen Auftrag zu geben –, schreiben wir Folgendes, um eine Liste aller Lieferanten in Aalen zu erhalten:

```
SELECT * FROM Lieferanten WHERE Wohnort='Aalen'
```

Wir sehen, dass es keinen Unterschied macht, ob die Abfragen in mehrere Zeilen aufgespaltet sind oder in einer Zeile stehen.

Zeilen ausfiltern und logische Operatoren

Die SELECT-Anweisung geht über die Spalten, und die WHERE-Angabe filtert Zeilen nach einem Kriterium heraus. Wir haben zunächst mit einer Gleich-Abfrage gearbeitet. SQL kennt die üblichen Vergleichsoperatoren: = gleich, <> ungleich, > größer, < kleiner, >= größer gleich, <= kleiner gleich. Vergleiche werden mit einem einfachen Gleichheitszeichen und nicht durch == formuliert. Die Vergleichsoperatoren lassen sich durch die Operatoren AND, OR und NOT weiter verfeinern. Bei numerischen Daten können wir auch die Rechenoperatoren (+, -, *, /) anwenden. Anstelle vielfacher AND-Abfragen lässt sich mit zwei SQL-Anweisungen auch der Wertebereich weiter einschränken. Die Einschränkung BETWEEN Wert1 AND Wert2 testet, ob sich ein Vergleichswert zwischen Wert1 und Wert2 befindet. Und IN (Werteliste) prüft, ob der Vergleichswert in der angegebenen Werteliste liegt. Für Zeichenketten spielt noch LIKE eine Rolle, da hier ein Mustervergleich vorgenommen werden kann. IS NULL erlaubt die Abfrage nach einem NULL-Wert in der Spalte.

Wenn wir diese SQL-Anweisung von der Datenbank ausführen lassen, wollen wir die Daten gern entsprechend dem Preis sortiert haben. Dazu lässt sich die SELECT-Anweisung mit einem ORDER BY versehen. Dahinter folgt die Spalte, nach der sortiert wird. Jetzt wird die Tabelle aufsteigend sortiert, also der kleinste Wert unten. Wünschen wir die Sortierung absteigend, dann setzen wir noch DESC hintenan.

Wir wollen nun die Informationen mehrerer Tabellen miteinander verbinden. Dazu führen wir eine Tupel-Variable ein. Diese kann eingesetzt werden, wenn sich Attribute nicht eindeutig den Relationen zuordnen lassen. Dies ist genau dann der Fall, wenn zwei Relationen verbunden werden sollen und beide den gleichen Attributnamen besitzen.

zB

Beispiel

Die SQL-Anweisung zeigt die Verwendung der Variablen, die hier jedoch nicht erforderlich ist, weil wir nur eine Tabelle verwenden:

```
SELECT L.Lfr_Name, L.Wohnort
FROM Lieferanten L
WHERE L.Wohnort = 'Aalen'
```

Der Buchstabe »L« ist hier nur eine Abkürzung, eine Art Variable. Abkürzungen für Spalten werden in SQL auch mit AS abgetrennt, etwa so:

```
FROM Lieferant AS L
```

Dann lässt sich auf die Spalte Lieferant in der Tabelle Lieferanten kurz mit L zugreifen.

Gruppenfunktionen

Mit *Gruppenfunktionen* (auch *Aggregationsfunktionen* genannt) lassen sich zum Beispiel Durchschnittswerte oder Minima über Spalten beziehen. Sie liefern genau einen Wert, beziehen sich jedoch auf mehrere Tabellenzeilen. Die folgende Abfrage zählt alle Anbieter aus Aalen:

```
SELECT COUNT(*)
FROM Lieferanten
WHERE Wohnort = 'Aalen'
```

Die Spalten, die die Gruppenfunktion bearbeiten, stehen in Klammern hinter dem Namen. Die SQL-Standardfunktionen (es gibt datenbankabhängig noch viel mehr) sind in der folgenden Tabelle aufgeführt:

Funktion	Beschreibung
AVG	Durchschnittswert
COUNT	Anzahl aller Einträge
MAX	Maximalwert
MIN	Minimalwert
SUM	Summe aller Einträge in einer Spalte

Tabelle 17.3: Die Standardfunktionen in SQL

17.2.3 Tabellen mit der Data Definition Language (DDL) anlegen

Die SQL-Anweisung CREATE TABLE legt eine neue Tabelle (Relation) an. Dazu gibt die Anweisung die Spalten (Attribute) mit ihren Wertebereichen an. Optional lassen sich Integritätsbedingungen definieren, etwa, ob eine Spalte mit einem Eintrag belegt sein muss (NOT NULL) oder welcher Fremdschlüssel eingetragen sein soll:

```
CREATE TABLE Tabelle
  (Spaltendefinition [,Spaltendefinition] ...
   [, Primärschlüsseldefinition]
   [, Fremdschlüsseldefinition
    [,Fremdschlüsseldefinition] ... ] )
  [IN Tabellspace]
  [INDEX IN Tabellspace2]
  [LONG IN Tabellspace3];
```

17.3 Datenbanken und Tools

Vor dem Glück, eine Datenbank in Java ansprechen zu können, steht die Inbetriebnahme des Datenbanksystems (für dieses Kapitel ist das fast schon der schwierigste Teil). Nun gibt es eine große Anzahl von Datenbanken – manche sind frei und Open Source, manche sehr teuer –, sodass sich dieses Tutorial nur auf eine Datenbank beschränkt. Das Rennen macht in dieser Auflage die pure Java-Datenbank *HSQLDB*, die sehr leicht ohne Administratorrechte läuft und leistungsfähig genug ist. Da JDBC aber von Datenbanken abstrahiert, ist der Java-Programmcode natürlich auf jeder Datenbank lauffähig.



Hinweis

Ab dem JDK 6 ist im Unterverzeichnis *db*, also etwa *C:\Program Files\Java\jdk1.6.0\db*, die Datenbank *Java DB* (<http://developers.sun.com/javadb/>) integriert. Sie basiert auf Apache *Derby*, dem früheren *Cloudscape* von IBM.

17.3.1 HSQLDB

HSQLDB (<http://hsqldb.org/>) ist ein pures Java-RDBMS unter der freien BSD-Lizenz. Die Datenbank lässt sich in zwei Modi fahren: als eingebettetes Datenbanksystem und als Netzwerkserver. Im Fall eines eingebauten Datenbanksystems ist lediglich die Treiberklasse zu laden und die Datenbank zu bestimmen, und schon geht's los. Wir werden für die folgenden Beispiele diese Variante wählen.

Auf der Download-Seite <http://sourceforge.net/projects/hsqldb/files/> von SourceForge befindet sich ein Archiv wie *hsqldb-2.2.5.zip* (8,2 MiB), das wir auspacken, zum Beispiel nach *c:\Programme\hsqldb*. Unter *C:\Programme\hsqldb\bin* befindet sich ein Skript *runManagerSwing.bat*, das ein kleines Datenbank-Frontend öffnet. Im folgenden Dialog CONNECT setzen wir

1. den Typ auf HSQL DATABASE ENGINE STANDALONE und
2. die JDBC-URL auf *jdbc:hsqldb:file:c:\TutegoDB* (statt *c:\TutegoDB* einen anderen Pfad eintragen, wohin die Datenbank erzeugt werden soll). Der Teil hinter *file:* gibt also den Pfad zu der Datenbank an, wobei der Pfad relativ oder absolut sein kann. Liegt die Datenbank im Eclipse-Workspace, kann später die absolute Angabe entfallen. Existiert die Datenbank nicht, wird sie unter dem angegebenen Pfad angelegt – das machen wir im ersten Schritt –, existiert sie, wird sie zum Bearbeiten geöffnet.

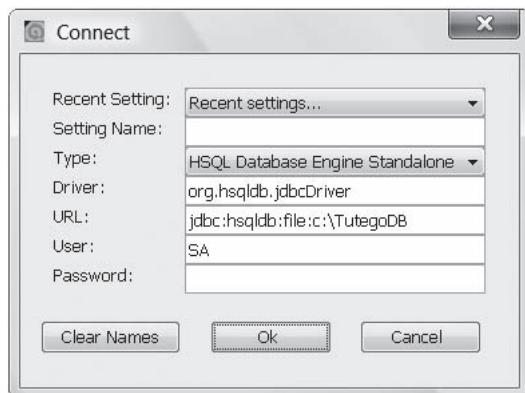


Abbildung 17.1: Verbindung aufbauen zur Datenbank

Nach dem Beenden des Dialogs mit OK fügt im Menü OPTIONS die Operation INSERT TEST DATA einige Tabellen mit Dummy-Daten ein und führt ein SQL-SELECT aus, das uns den Inhalt der Customer-Tabelle zeigt. Beenden wir anschließend das Swing-Programm mit FILE • EXIT. Im Dateisystem hat der Manager jetzt eine *.log*-Datei angelegt – zu ihr gesellt sich später noch eine *.script*-Datei –, eine *.properties*-Datei und eine *.lck*-Datei.

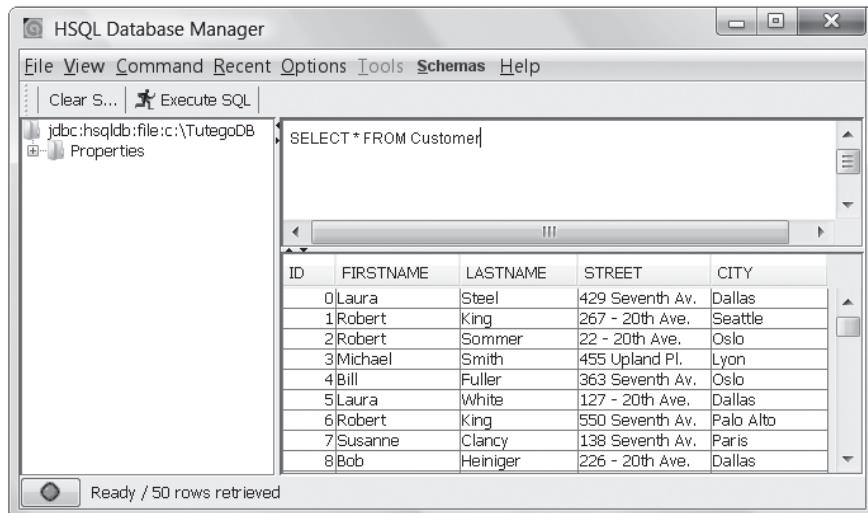


Abbildung 17.2: SQL-Kommandos absetzen und Resultate einsehen

Für den Datenbankzugriff aus Java ist nur das Archiv *hsqldb.jar* aus dem *lib*-Verzeichnis von HSQLDB in den Klassenpfad aufzunehmen.

17.3.2 Eclipse-Plugins zum Durchschauen von Datenbanken

Es gibt fast genauso viele Tools zum Administrieren von Datenbanken wie Datenbanken selbst. Zwar bringt NetBeans direkt ein Plugin zum Durchstöbern von Datenbanken mit, doch leider nicht die Eclipse IDE – auch nicht in der Ausgabe *Eclipse IDE for Java EE Developers*. So muss ein extra Plugin installiert werden. Von der Eclipse-Foundation gibt es *Eclipse Data Tools Platform* (DTP) und eine Webseite <http://www.eclipse.org/datatools/>. Frei und in Java (aber kein Eclipse-Plugin) ist *SQuirreL* (<http://squirrel-sql.sourceforge.net/>).

Eclipse Data Tools Platform (DTP)

Die DTP wird über den Online-Update-Mechanismus installiert. Wir wählen **HELP • INSTALL NEW SOFTWARE...** und geben bei **WORK WITH** die URL <http://download.eclipse.org/datatools/updates/> ein. Wir aktivieren **ADD...** und bestätigen den nächsten Dialog mit **OK**. Es folgt ein Dialog mit Versionen, aus denen wir die letzte DTP-Version auswählen können, etwa **ECLIPSE DATA TOOLS PLATFORM SDK 1.9.0**.

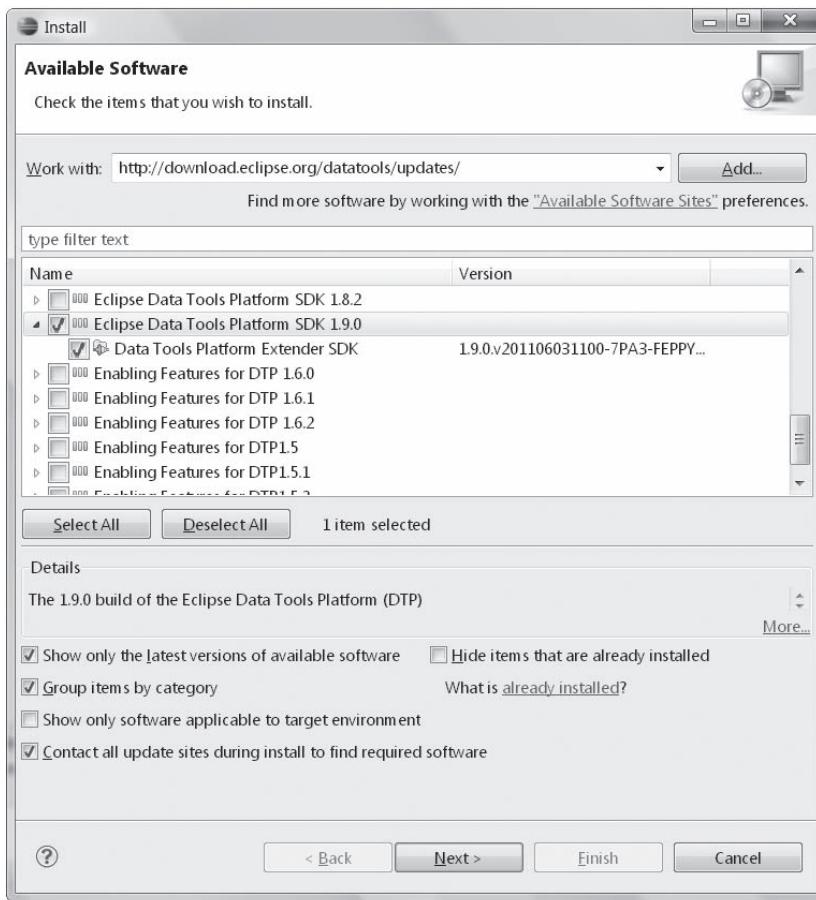


Abbildung 17.3: Auswählen des DTP-Plugins

Nach ein paar Ja-Next-Dialogen wird Eclipse neu gestartet, und das Plugin ist installiert. Wir wechseln anschließend die Perspektive mit **WINDOW • OPEN PERSPECTIVE • OTHER...** und wählen dann **DATABASE DEVELOPMENT**.

Es gibt in der Perspektive einige neue Ansichten. Eine ist **DATA SOURCE EXPLORER**, die sich auch durch **WINDOW • SHOW VIEW • DATA SOURCE EXPLORER** für andere Perspektiven aktivieren lässt. In der Ansicht wählen wir im Zweig **DATABASE CONNECTIONS** über das Kontextmenü den Eintrag **NEW...** So lässt sich eine neue Datenbankverbindung einrichten. Im folgenden Dialog wählen wir **HSQLDB** aus der Liste. **NEXT** bringt uns zu einem neuen Dialog. Rechts neben dem Auswahlfeld bei **DRIVERS** ist eine unscheinbare Schaltfläche mit einem Kreis und + -Symbol.

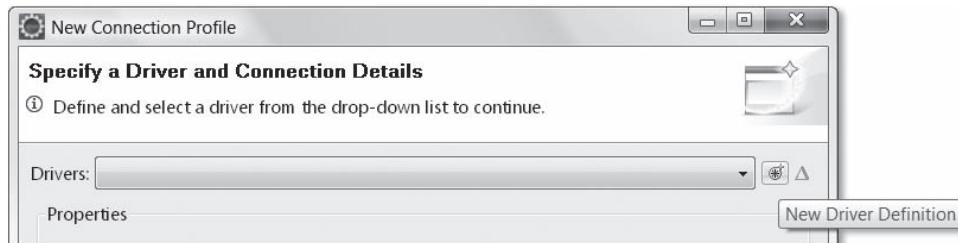


Abbildung 17.4: Treiberdetails bestimmen

Nach dem Aktivieren öffnet sich ein weiterer Dialog mit dem TITEL NEW DRIVER DEFINITION. Aus der Liste wählen wir unter DATABASE den HSQLB JDBC DRIVER aus und gehen auf den zweiten Reiter, auf JAR LIST. Mit ADD JAR/ZIP... kommt ein Auswahldialog, und wir navigieren zu HSQLDB.JAR.

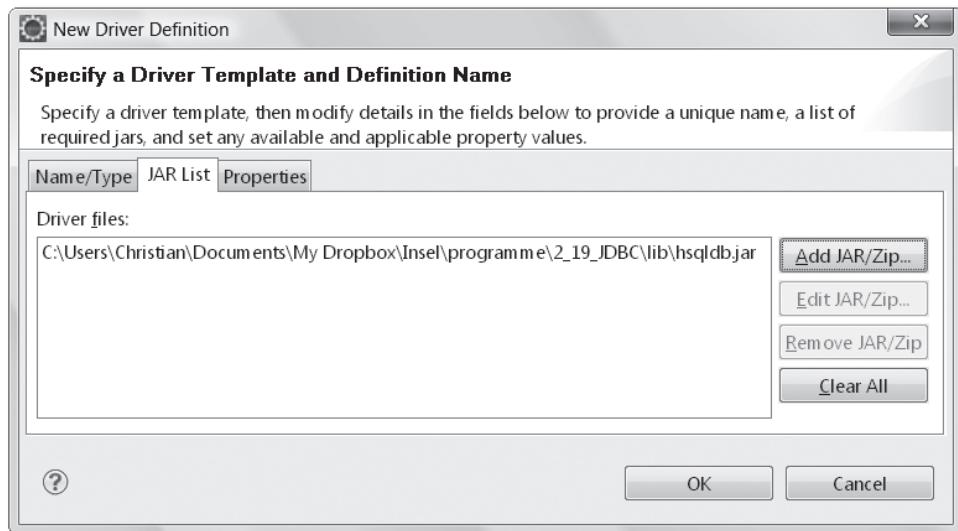


Abbildung 17.5: Jar-Datei auswählen

Nach diesem Eintrag aktivieren wir den dritten Reiter, PROPERTIES. Wir tragen Folgendes ein:

- CONNECTION URL: die JDBC-URL für die angelegte Datenbank, etwa `jdbc:hsqldb:file:c:/TutegoDB`
- DATABASE NAME: ein beliebiger Name, der nur zur Anzeige dient, etwa `tutegoDB`

- DRIVER CLASS: die Treiberklasse *org.hsqldb.jdbcDriver*
- Unter USER ID tragen wir »sa« ein.

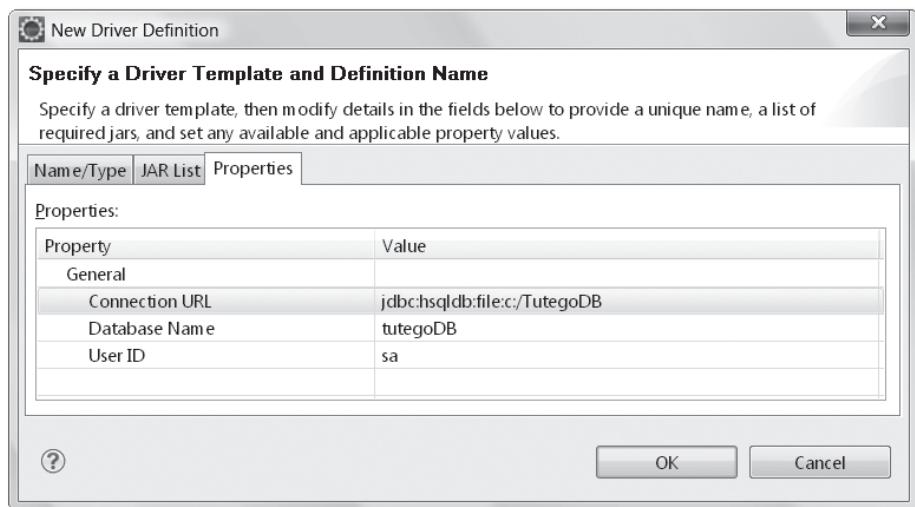


Abbildung 17.6: Verbindungsdaten eintragen

Mit OK bestätigen wir den Dialog, und anschließend sollte der Klick auf die Schaltfläche TEST CONNECTION bezeugen, dass alles gut geht und es keine Probleme mit den Parametern gab. FINISH schließt den Dialog, und nach einer erfolgreichen Verbindung sind in der Ansicht die Datenbank sowie ihre Schemas zu sehen.

Um eine SQL-Abfrage auszuführen, öffnen wir den Dialog unter FILE • NEW • OTHER... • SQL DEVELOPMENT • SQL FILE, klicken auf NEXT und geben einen Dateinamen wie *test* für eine Skriptdatei an. Im unteren Bereich des Dialogs lässt sich direkt die Datenbank auswählen. Wählen wir für DATABASE SERVER TYPE den Eintrag HSQLDB_1.8, für CONNECTION PROFILE NAME anschließend NEW HSQLDB und abschließend als DATABASE NAME aus dem Auswahlmenü PUBLIC. FINISH schließt den Dialog, legt eine Datei *test.sqlpage* an und öffnet diese in einem neuen Editor für SQL-Anweisungen. Tragen wir dort Folgendes ein:

```
SELECT * FROM Customer
```

Das Kontextmenü im SQL-Editor bietet EXECUTE ALL. In der Ansicht SQL RESULTS sind die Ergebnisse dann abzulesen.

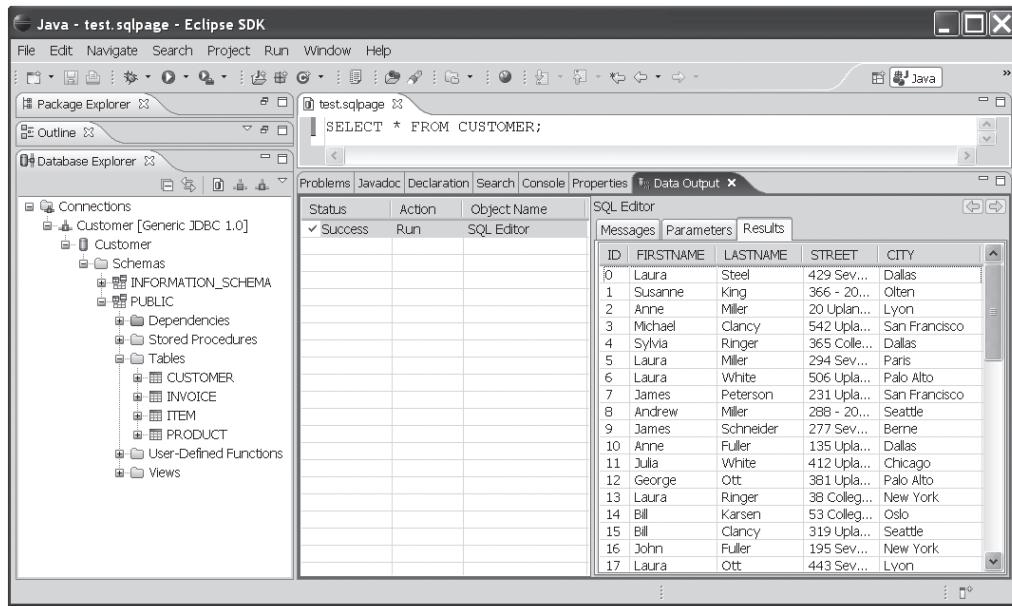


Abbildung 17.7: Die drei Ansichten »Database Explorer«, »Data Output« und der Editor für das SQL Scrapbook



Tipp

Mit der rechten Maustaste lassen sich im Kontextmenü EDIT IN SQL QUERY BUILDER... die Abfragen auch etwas mehr grafisch visualisieren.

Wenn wir unsere Beispiele beendet haben, sollten wir im DATA SOURCE EXPLORER die Verbindung wieder schließen; dazu ist auf unserer Datenbank in der Ansicht DATABASE EXPLORER im Kontextmenü DISCONNECT zu wählen.

17.4 JDBC und Datenbanktreiber

JDBC ist die inoffizielle Abkürzung für *Java Database Connectivity* und bezeichnet einen Satz von Schnittstellen, um relationale Datenbanksysteme von Java zu nutzen.

Die erste JDBC-Spezifikation gab es im Juni 1996. Die Schnittstellen und wenigen Klassen sind ab dem JDK 1.1 im Core-Paket integriert. Die JDBC-API und ihre Treiber erreichen eine wirksame Abstraktion von relationalen Datenbanken, sodass durch die einheitliche Programmierschnittstelle die Funktionen differierender Datenbanken in gleicher Weise genutzt werden können. Das Lernen von verschiedenen Zugriffsmethoden für unterschiedliche Datenbanken der Hersteller entfällt. Wie jedoch diese spezielle Datenbank nun wirklich aussieht, verheimlicht uns die Abstraktion. Jede Datenbank hat ihr eigenes Protokoll (und eventuell auch Netzwerkprotokoll), doch die Implementierung ist nur dem Datenbanktreiber bekannt.

Das Modell von JDBC setzt auf dem X/OPEN-SQL-Call-Level-Interface (CLI) auf und bietet somit die gleiche Schnittstelle wie Microsofts ODBC (Open³ Database Connectivity). Dem Programmierer gibt JDBC Methoden, um Verbindungen zu Datenbanken aufzubauen, Datensätze zu lesen oder neue Datensätze zu verfassen. Zusätzlich können Tabellen aktualisiert und Prozeduren auf der Serverseite ausgeführt werden.

Hinweis

Ein JDBC-Treiber muss nicht unbedingt relationale Datenbanken ansprechen, obwohl das der häufigste Fall ist. Mit dem freien *xSQL* (<https://xsql.dev.java.net/>) steht ein JDBC-Treiber bereit, der auf Excel-Tabellen beziehungsweise CSV-Dateien arbeitet, und Oracle bietet mit *Synopsis* (<http://www.sunopsis.com/corporate/us/products/jdbc-forxml/>) ein Produkt, das statt relationaler Datenbanken XML-Dokumente verwendet.

17

Implementierung der JDBC-API

Um eine Datenbank ansprechen zu können, müssen wir einen Treiber haben, der die JDBC-API implementiert und zwischen dem Java-Programm und der Datenbank vermittelt. Jeder Treiber ist üblicherweise anders implementiert, denn er muss die datenbankunabhängige JDBC-API auf die konkrete Datenbank übertragen. Oracle veröffentlicht unter <http://developers.sun.com/product/jdbc/drivers> Treiber zu allen möglichen Datenbanken. Eine Suchmaske erlaubt die Eingabe einer Datenbank und die Auswahl eines gewünschten Typs.

³ Microsoft und Open? Eine ungewohnte Kombination ...

17.5 Eine Beispielabfrage

17.5.1 Schritte zur Datenbankabfrage

Wir wollen kurz die Schritte skizzieren, die für einen Zugriff auf eine relationale Datenbank mit JDBC erforderlich sind:

1. Einbinden der JDBC-Datenbanktreiber in den Klassenpfad
2. unter Umständen Anmelden der Treiberklassen
3. Verbindung zur Datenbank aufzubauen
4. eine SQL-Anweisung erzeugen
5. SQL-Anweisung ausführen
6. das Ergebnis der Anweisung holen, bei Ergebnismengen über diese iterieren
7. die Datenbankverbindung schließen

Wir beschränken uns im Folgenden auf die Verbindung zum freien Datenbanksystem HSQLDB.

17.5.2 Ein Client für die HSQLDB-Datenbank

Ein Beispiel soll zu Beginn die Programmkonzepte für JDBC veranschaulichen, bevor wir im Folgenden das Java-Programm weiter sezieren. Das Programm in der Klasse FirstSqlAccess nutzt die Datenbank TutegoDB, die sich im Suchpfad befinden muss; wir können ebenso absolute Pfade bei HSQLDB angeben, etwa *C:/TutegoDB*. Bei der Parametrisierung »*jdbc:hsqldb:file:....*« von HSQLDB liest die Datenbank beim ersten Start die Daten aus der Datei ein, verwaltet sie im Speicher und schreibt sie am Ende des Programms wieder in eine Datei zurück.

Da wir die Datenbank schon früher mit Demo-Daten gefüllt haben, lässt sich jetzt eine SQL-SELECT-Abfrage absetzen:

Listing 17.1: com/tutego/insel/jdbc/FirstSqlAccess.java

```
package com.tutego.insel.jdbc;

import java.sql.*;

public class FirstSqlAccess
{gc
    public static void main( String[] args )
```

```
{  
    try  
    {  
        Class.forName( "org.hsqldb.jdbcDriver" );  
    }  
    catch ( ClassNotFoundException e )  
    {  
        System.err.println( "Keine Treiber-Klasse!" );  
        return;  
    }  
  
    Connection con = null;  
  
    try  
    {  
        con = DriverManager.getConnection(  
            "jdbc:hsqldb:file:TutegoDB;shutdown=true", "sa", "" );  
        Statement stmt = con.createStatement();  
  
        //      stmt.executeUpdate( "INSERT INTO CUSTOMER " +  
        //      "VALUES(50,'Christian','Ullenboom','Immengarten 6','Hannover')" );  
  
        ResultSet rs = stmt.executeQuery( "SELECT * FROM Customer" );  
  
        while ( rs.next() )  
            System.out.printf( "%s, %s %s%n", rs.getString(1),  
                rs.getString(2), rs.getString(3) );  
  
        rs.close();  
  
        stmt.close();  
    }  
    catch ( SQLException e )  
    {  
        e.printStackTrace();  
    }  
}
```

```

        finally
    {
        if ( con != null )
            try { con.close(); } catch ( SQLException e ) { e.printStackTrace(); }
    }
}
}
}

```

Dem Beispiel ist in diesem Status schon die aufwändige Fehlerbehandlung anzusehen. Das Schließen vom ResultSet und Statement ist vereinfacht, aber okay, weil das finally auf jeden Fall die Connection schließt. Ab Java 7 kann auch try-mit-Ressourcen die Verbindung automatisch schließen.



Hinweis

Es ist möglich, auch ohne ODBC-Eintrag Zugriff auf eine Access-Datenbank aufzubauen – nützlich ist das zum Beispiel dann, wenn der Name der Datenbank erst später bekannt wird.

```

con = DriverManager.getConnection( "jdbc:odbc:Driver="+
    "{Microsoft Access Driver (*.mdb)};DBQ=c:/daten/test.mdb",
    "name", "pass" );

```

Ein ähnlicher String kann auch für den Zugriff auf eine dBase-Datenbank genutzt werden, für die ein ODBC-Treiber angemeldet ist:

```
jdbc:odbc:Driver={Microsoft dBase Driver (*.dbf)};DBQ=c:\database.dbf
```

17.5.3 Datenbankbrowser und eine Beispielabfrage unter NetBeans

Wer mit NetBeans arbeitet, der kann einfach mit der ab Java 6 mitgelieferten Datenbank JavaDB arbeiten, denn NetBeans bringt eine Beispieldatenbank für Java DB mit. Im Folgenden soll

1. die Beispieldatenbank gestartet,
2. die Datenbank mit der Browser untersucht und
3. ein Java-Programm geschrieben werden, das diese Datenbank anspricht.

Beispieldatenbank starten

Ist NetBeans gestartet, wählen wir im Menü WINDOW • SERVICES. Links kommt in der Darstellung ein Punkt DATABASES hinzu, wobei unser Interesse der Beispieldatenbank dient, zu der wir mit CONNECT über das Kontextmenü eine Verbindung aufbauen wollen.

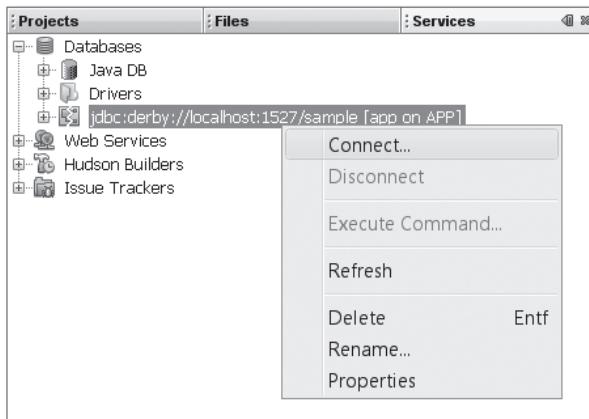


Abbildung 17.8: Demo-Datenbank starten

In der Ausgabe ist zu erkennen, dass die Datenbank nun gestartet und bereit für Verbindungen ist.

```
Output - Java DB Database Process
2011-08-09 16:55:34.206 GMT : Sicherheitsmanager mit einfacher Server-Sicherheitsrichtlinie installiert.
2011-08-09 16:55:34.821 GMT : Apache Derby Network Server 10.5.3.0 - (802917) wurde gestartet und ist bereit, Verbindungen am Port 1527 zu akzeptieren.
```

Abbildung 17.9: Ausgabe nach dem Start der Datenbank

SQL-Anweisungen absetzen

Links ist anschließend der Baum mit vielen Informationen gefüllt, und alle Tabelleninformationen sind zugänglich. Mit der rechten Maustaste und dem Kontextmenü lassen sich anschließend SQL-Anweisungen über EXECUTE COMMAND... an die Datenbank absetzen.

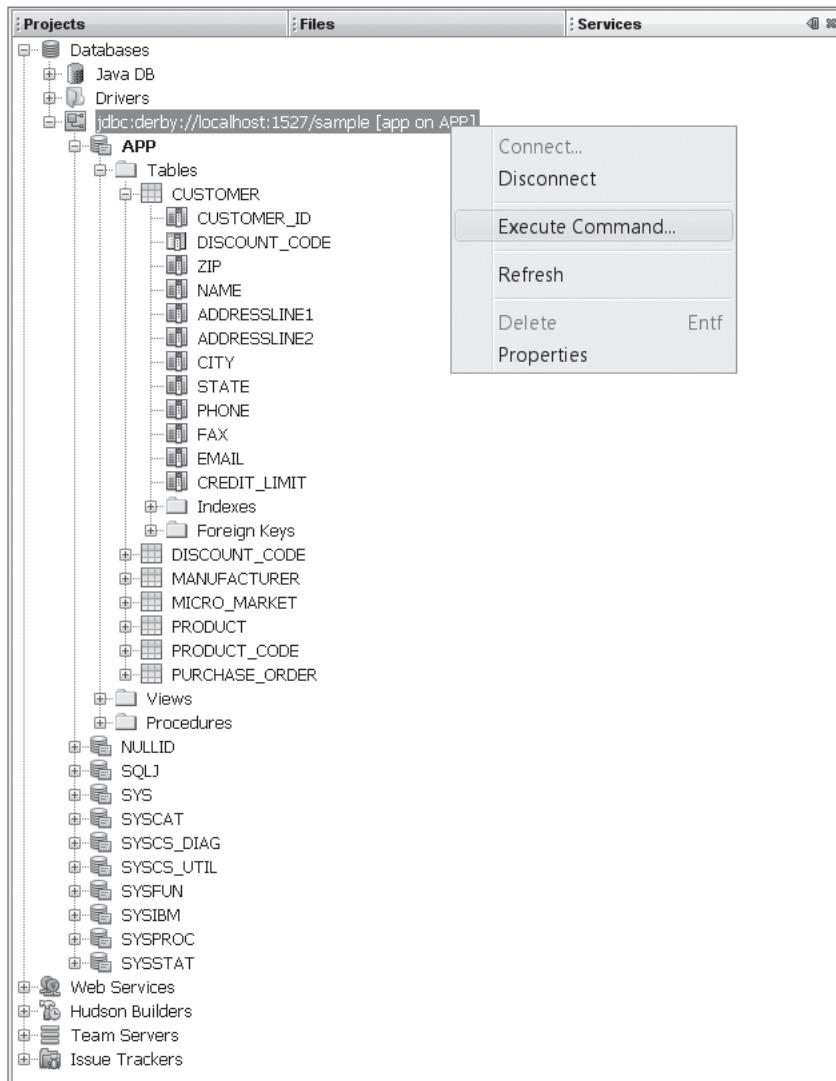


Abbildung 17.10: SQL-Abfragen starten

Es öffnet sich ein SQL-Editor, der Tastaturvervollständigung beherrscht und sogar in die Datenbank schaut, um Tabellen und Spaltennamen korrekt zu vervollständigen.

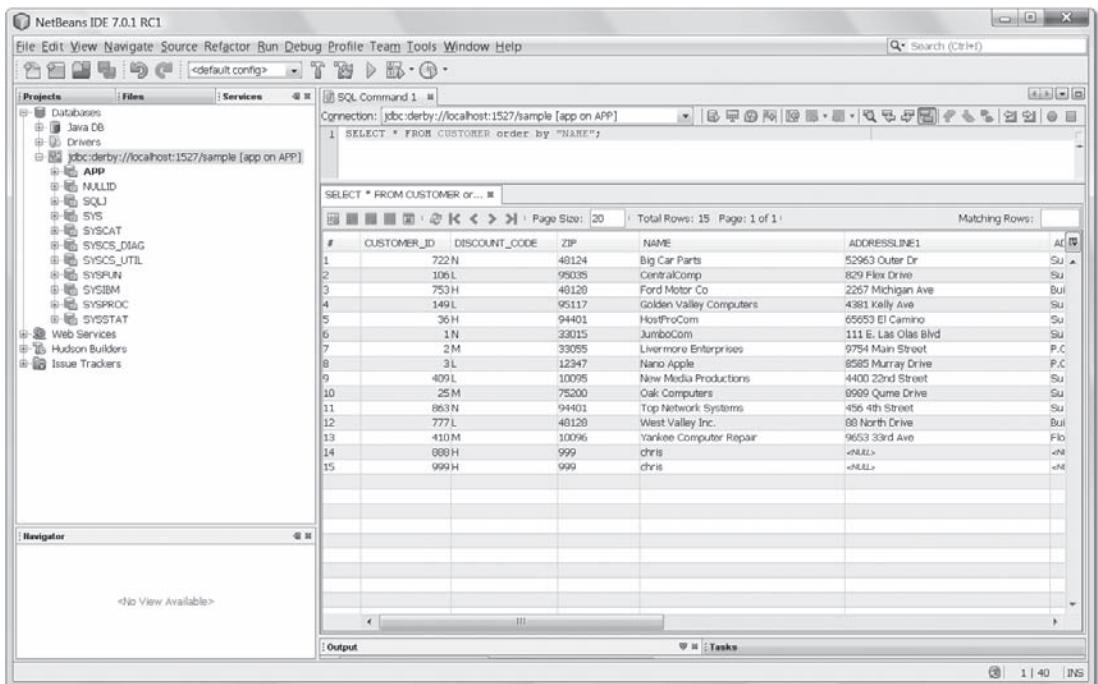


Abbildung 17.11: Ergebnis einer SQL-Abfrage

JDBC-Beispiel

Das JDBC-Beispiel von eben können wir leicht auf die NetBeans-Datenbank übertragen. Drei Dinge müssen wir anpassen:

- Der Treiber muss im Klassenpfad stehen.
- Die Treiberklasse ist `org.apache.derby.jdbc.ClientDriver`. Das explizite Laden kann aber entfallen, da Java einen JDBC 4-Treiber selbstständig findet, wenn er im Klassenpfad steht.
- Die Datenbank-URL ist `jdbc:derby://localhost:1527/sample`.

Die letzten beiden Dinge sind schnell im Quellcode angepasst. Um den Treiber in den Klassenpfad zu setzen, wählen wir links im Projekt bei LIBRARIES das Kontextmenü und dann ADD JAR/FOLDER...



Abbildung 17.12: Java-Archive hinzufügen

Aus dem JDK-Installationsverzeichnis unter *db/lib* wählen wir **DERBYCLIENT.JAR**.

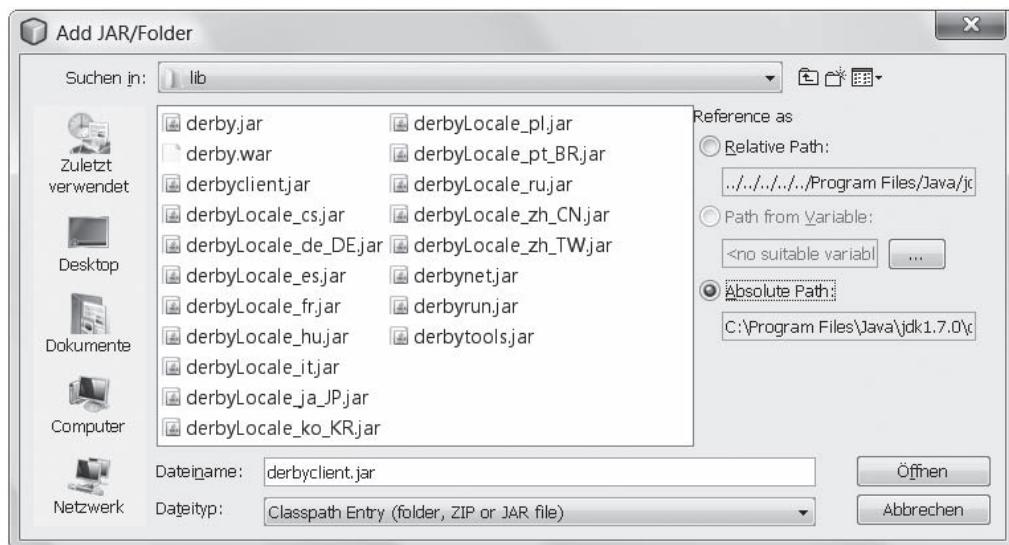


Abbildung 17.13: »derbyclient.jar« auswählen

ÖFFNEN fügt das Jar-Archiv hinzu.

Wir wollen das SELECT noch etwas anpassen, und dann folgt:

Listing 17.2: com/tutego/insel/jdbc/FirstSqlAccess.java

```
package com.tutego.insel.jdbc;
```

```
import java.sql.*;
```

```
public class SecondSqlAccess
{
```

```
public static void main( String[] args )
{
    Connection con = null;

    try
    {
        con = DriverManager.getConnection( "jdbc:derby://localhost:1527/sample",
                                         "app", "app" );
        Statement stmt = con.createStatement();

        ResultSet rs = stmt.executeQuery(
            "SELECT NAME, ADDRESSLINE1, PHONE FROM Customer" );

        while ( rs.next() )
            System.out.printf( "%s, %s %s%n", rs.getString(1),
                               rs.getString(2), rs.getString(3) );

        rs.close();

        stmt.close();
    }
    catch ( SQLException e )
    {
        e.printStackTrace();
    }
    finally
    {
        if ( con != null )
            try { con.close(); } catch ( SQLException e ) { e.printStackTrace(); }
    }
}
```




Kapitel 18

Bits und Bytes und Mathematisches

»Vieles hätte ich verstanden, wenn man es mir nicht erklärt hätte.«

– Stanislaw Jerzy Lec (1909–1966)

Dieses Kapitel betrachtet die Repräsentationen der Zahlen genauer und wie binäre Operatoren auf diesen Werten arbeiten. Nachdem die Ganzzahlen genauer beleuchtet wurden, folgen eine detaillierte Darstellung der Fließkommazahlen und anschließend mathematische Grundfunktionen wie `max()`, `sin()`, `abs()`, die in Java die Klasse `Math` realisiert.

18.1 Bits und Bytes *

Ein *Bit* ist ein Informationsträger für die Aussage wahr oder falsch. Durch das Zusammensetzen von einzelnen Bits entstehen größere Folgen wie das *Byte*, das aus 8 Bit besteht. Da jedes Bit anders belegt sein kann, bildet es in der Summe unterschiedliche Werte. Werden 8 Bit zugrunde gelegt, lassen sich durch unterschiedliche Belegungen 256 unterschiedliche Zahlen bilden. Ist kein Bit des Bytes gesetzt, so ist die Zahl 0. Jede Stelle im Byte bekommt dabei eine Wertigkeit zugeordnet. Die Werteberechnung für die Zahl 19 berechnet sich aus $16 + 2 + 1$, da sie aus einer Anzahl von Summanden der Form 2^n zusammengesetzt ist: $19_{\text{dez}} = 16 + 2 + 1 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10011_{\text{bin}}$.

18

Bit	7	6	5	4	3	2	1	0
Wertigkeit	$2^7=128$	$2^6=64$	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
Belegung für 19	0	0	0	1	0	0	1	1

Tabelle 18.1: Werteberechnung

18.1.1 Die Bit-Operatoren Komplement, Und, Oder und Xor

Mit Bit-Operatoren lassen sich Binäroperationen auf Operanden durchführen, um beispielsweise ein Bit eines Bytes zu setzen. Zu den Bit-Operationen zählen Verknüpfungen, Schiebeoperationen und das Komplement. Durch die bitweisen Operatoren können einzelne Bits abgefragt und manipuliert werden. Als Verknüpfungen bietet Java die folgenden Bit-Operatoren an:

Operator	Bezeichnung	Aufgabe
<code>~</code>	Komplement	Invertiert jedes Bit.
<code> </code>	bitweises Oder	Bei $a b$ wird jedes Bit von a und b einzeln Oder-verknüpft.
<code>&</code>	bitweises Und	Bei $a \& b$ wird jedes Bit von a und b einzeln Und-verknüpft.
<code>^</code>	bitweises exklusives Oder (Xor)	Bei $a ^ b$ wird jedes Bit von a und b einzeln Xor-verknüpft; es ist kein a hoch b .

Tabelle 18.2: Bit-Operatoren in Java

Betrachten wir allgemein die binäre Verknüpfung $a \# b$. Bei der binären bitweisen Und-Verknüpfung mit `&` gilt für jedes Bit: Ist im Operand a irgendein Bit gesetzt und an gleicher Stelle auch im Operand b , so ist auch das Bit an der Stelle im Ergebnis gesetzt. Bei der Oder-Verknüpfung mit `|` muss nur einer der Operanden gesetzt sein, damit das Bit im Ergebnis gesetzt ist. Bei einem exklusiven Oder (Xor) ist das Ergebnis 1, wenn nur genau einer der Operanden 1 ist. Sind beide gemeinsam 0 oder 1, ist das Ergebnis 0. Dies entspricht einer binären Addition oder Subtraktion. Fassen wir das Ergebnis noch einmal in einer Tabelle zusammen:

Bit 1	Bit 2	\sim Bit 1	Bit 1 & Bit 2	Bit 1 Bit 2	Bit 1 ^ Bit 2
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Tabelle 18.3: Die Bit-Operatoren Komplement, Und, Oder und Xor in einer Wahrheitstafel

Nehmen wir zum Beispiel zwei Ganzahlen:

	binär	dezimal
Zahl 1	010011	$16 + 2 + 1 = 19$
Zahl 2	100010	$32 + 2 = 34$
Zahl 1 & Zahl 2	000010	$19 \& 34 = 2$
Zahl 1 Zahl 2	110011	$19 34 = 51$
Zahl 1 ^ Zahl 2	110001	$19 ^ 34 = 49$

Tabelle 18.4: Binäre Verknüpfung zweier Ganzzahlen

Variablen mit Xor vertauschen

Eine besonders trickreiche Idee für das Vertauschen von Variableninhalten arbeitet mit dem Xor-Operator und benötigt keine temporäre Zwischenvariable. Die Zeilen zum Vertauschen von x und y lauten wie folgt:

```
int x = 12,
y = 49;
x ^= y; // x = x ^ y = 001100bin ^ 110001bin = 111101bin
y ^= x; // y = y ^ x = 110001bin ^ 111101bin = 001100bin
x ^= y; // x = x ^ y = 111101bin ^ 001100bin = 110001bin
System.out.println( x + " " + y ); // Ausgabe ist: 49 12
```

18

Der Trick funktioniert, da wir mit Xor etwas »hinein- und herausrechnen« können. Zuerst rechnet die erste Zeile das y in das x . Wenn wir anschließend die Zuweisung an das y machen, dann ist das der letzte schreibende Zugriff auf y , also muss hier schon das vertauschte Ergebnis stehen. Das stimmt auch, denn expandieren wir die zweite Zeile, steht dort: » $y ^ x$ wird zugewiesen an y «, und dies ist $y ^ (x ^ y)$. Der letzte Ausdruck verkürzt sich zu $y = x$, da aus der Definition des Xor-Operators für einen Wert a hervorgeht: $a ^ a = 0$. Die Zuweisung hätten wir zwar gleich so schreiben können, aber dann wäre der Wert von y verloren gegangen. Der steckt aber noch in x aus der ersten Zuweisung. Betrachten wir daher die letzte Zeile $x ^ y$: y hat den Startwert von x , doch in x steckt ein Xor- y . Daher ergibt $x ^ y$ den Wert $x ^ x ^ y$, und der verkürzt sich zu y . Demnach haben wir den Inhalt der Variablen vertauscht. Im Übrigen können wir für die drei Xor-Zeilen alternativ schreiben:

```
y ^= x ^= y; // Auswertung automatisch y ^= (x ^= y)
x ^= y;
```

Da liegt es doch nahe, die Ausdrücke weiter abzukürzen zu $x \wedge= y \wedge= x \wedge= y$. Doch leider ist das falsch (es kommt für x immer null heraus). Motivierten Lesern bleibt dies als Denksportaufgabe überlassen.

18.1.2 Repräsentation ganzer Zahlen in Java – das Zweierkomplement

Das *Zweierkomplement* definiert für positive und negative Ganzzahlen folgende Kodierung:

- Das Vorzeichen einer Zahl bestimmt ein Bit, das 1 bei negativen und 0 bei positiven Zahlen ist.
- Um eine 0 darzustellen, ist kein Bit gesetzt.

Java kodiert die Ganzahldatentypen `byte`, `short`, `int` und `long` immer im Zweierkomplement (der Datentyp `char` definiert keine negativen Zahlen). Mit dieser Kodierung gibt es eine negative Zahl mehr als positive, da es im Zweierkomplement keine positive und negative 0 gibt, sondern nur eine »positive« mit der Bitmaske 0000...0000.

dezimal	binär	hexadezimal
-32.768	1000 0000 0000 0000	80 00
-32.767	1000 0000 0000 0001	80 01
-32.766	1000 0000 0000 0010	80 02
	...	
-2	1111 1111 1111 1110	FF FE
-1	1111 1111 1111 1111	FF FF
0	0000 0000 0000 0000	00 00
1	0000 0000 0000 0001	00 01
2	0000 0000 0000 0010	00 02
	...	
32.766	0111 1111 1111 1110	7F FE
32.767	0111 1111 1111 1111	7F FF

Tabelle 18.5: Darstellungen beim Zweierkomplement beim Datentyp `short`

Bei allen negativen Ganzzahlen ist also das oberste Bit mit 1 gesetzt.

18.1.3 Das binäre (Basis 2), oktale (Basis 8), hexadezimale (Basis 16) Stellenwertsystem

Die Literale für Ganzzahlen lassen sich in vier unterschiedlichen Stellenwertsystemen angeben. Das natürlichste ist das *Dezimalsystem* (auch *Zehnersystem* genannt), bei dem die Literale aus den Ziffern »0« bis »9« bestehen. Zusätzlich existieren die *Binär-* (erst ab Java 7), *Oktal-* und *Hexadezimalsysteme*, die die Zahlen zur Basis 2, 8 und 16 schreiben. Bis auf Dezimalzahlen beginnen die Zahlen in anderen Formaten mit einem besonderen Präfix.

Präfix	Stellenwertsystem	Basis	Darstellung von 1
0b oder 0B	binär	2	0b1 oder 0B1
0	oktal	8	01
Kein	dezimal	10	1
0x oder 0X	hexadezimal	16	0x1 oder 0X1

Tabelle 18.6: Die Stellenwertsysteme und ihre Schreibweise

Ein hexadezimaler Wert beginnt mit »0x« oder »0X«. Da zehn Ziffern für 16 hexadezimale Zahlen nicht ausreichen, besteht eine Zahl zur Basis 16 zusätzlich aus den Buchstaben »a« bis »f« (beziehungsweise »A« bis »F«). Das Hexadezimalsystem heißt auch *Sedezimalsystem*.¹

Ein oktaler Wert beginnt mit dem Präfix »0«. Mit der Basis 8 werden nur die Ziffern »0« bis »7« für oktale Werte benötigt. Der Name stammt von dem lateinischen »octo«, was auf Deutsch »acht« heißt. Das Oktalsystem war früher eine verbreitete Darstellung, da nicht mehr einzelne Bits solo betrachtet werden mussten, sondern 3 Bits zu einer Gruppe zusammengefasst wurden. In der Kommunikationselektronik ist das Oktalsystem noch weiterhin beliebt, spielt aber sonst keine Rolle.

Für Dualzahlen (also Binärzahlen zur Basis 2) wurde eine neue Notation in Java 7 eingeführt. Das Präfix ist »0b« oder »0B«. Es sind nur die Ziffern »0« und »1« erlaubt.

¹ Das Präfix »octo« bei »Oktalsystem« stammt aus dem Lateinischen. Das Wort »Hexadezimal« enthält zwei Bestandteile aus zwei verschiedenen Sprachen: »hexa« stammt aus dem Griechischen und »decem« (zehn) aus dem Lateinischen. Die alternative Bezeichnung *Sedezimalzahl* bzw. *sedezimal* (engl. *sexadecimal* – nicht *sexagesimal*, das ist Basis 60) ist rein aus dem Lateinischen abgeleitet, aber im Deutschen unüblich. Über den Ursprung des Wortes »Hexadezimal« finden Sie mehr unter <http://en.wikipedia.org/wiki/Hexadecimal#Etymology>.

zB Beispiel

Gib Dezimal-, Binär, Oktal- und Hexadezimalzahlen aus:

```
System.out.println( 1243 );           // 1243
System.out.println( 0b10111011 );     // 187
System.out.println( 01230 );          // 664
System.out.println( 0xcafebabe );    // -889275714
System.out.println( 0xCOBOL );        // 49328
```

In Java-Programmen sollten Oktalzahlen mit Bedacht eingesetzt werden. Wer aus optischen Gründen mit der 0 eine Zahl linksbündig auffüllt, erlebt eine Überraschung:

```
int i = 118;
int j = 012;                      // Oktal 012 ist dezimal 10
```

18.1.4 Auswirkung der Typanpassung auf die Bitmuster

Die Typanpassung führt dazu, dass bei Ganzzahlen die oberen Bits einfach abgeschnitten werden. Bei einer Anpassung von Fließkommazahlen auf Ganzzahlen wird gerundet. Was genau passiert, soll dieser Abschnitt zeigen.

Explizite Typumwandlung bei Ganzzahlen

Bei der Konvertierung eines größeren Ganzzahltyps in einen kleineren werden die oberen Bits abgeschnitten. Eine Anpassung des Vorzeichens findet *nicht* statt. Die Darstellung in Bit zeigt das sehr anschaulich:

```
int ii = 123456789;      // 000001110101101111000110100010101
int ij = -123456;        // 111111111111111100001110110000000

short si = (short) ii;   // 1100110100010101
short sj = (short) ij;   // 0001110111000000

System.out.println( si ); // -13035
System.out.println( sj ); // 7616
```

si wird eine negative Zahl, da das 16. Bit beim int ii gesetzt war und nun beim short das negative Vorzeichen anzeigt. Die Zahl hinter ij hat kein 16. Bit gesetzt, und so wird das short sj positiv.

Umwandlung von short und char

Ein `short` hat wie ein `char` eine Länge von 16 Bit. Doch diese Umwandlung ist nicht ohne ausdrückliche Konvertierung möglich. Das liegt am Vorzeichen von `short`. Zeichen sind per Definition immer ohne Vorzeichen. Würde ein `char` mit einem gesetzten höchstwertigen letzten Bit in ein `short` konvertiert, käme eine negative Zahl heraus. Ebenso wäre, wenn ein `short` eine negative Zahl bezeichnet, das oberste Bit im `char` gesetzt, was unerwünscht ist. Die ausdrückliche Umwandlung erzeugt immer nur positive Zahlen.

Der Verlust bei der Typumwandlung von `char` nach `short` tritt etwa bei der Han-Zeichenkodierung für chinesische, japanische oder koreanische Zeichen auf, weil dort im Unicode das erste Bit gesetzt ist, das bei der Umwandlung in ein `short` dem nicht gesetzten Vorzeichen-Bit weichen muss.

Typumpassungen von int und char

Die Methode `printXXX()` ist mit den Typen `char` und `int` überladen, und eine Typumwandlung führt zur gewünschten Ausgabe:

```
int c1 = 65;
char c2 = 'A';
System.out.println( c1 );           // 65
System.out.println( (int)c2 );      // 65
System.out.println( (char)c1 );     // A
System.out.println( c2 );          // A
System.out.println( (char)(c1 + 1) ); // B
System.out.println( c2 + 1 );       // 66
```

Einen Ganzzahlwert in einem `int` können wir als Zeichen ausgeben, genauso wie eine `char`-Variable als Zahlenwert. Wir sollten beachten, dass eine arithmetische Operation auf `char`-Typen zu einem `int` führt. Daher funktioniert für ein `char c` Folgendes nicht:

```
c = c + 1;
```

Richtig wäre:

```
c = (char)(c + 1)
```

Unterschiedliche Wertebereiche bei Fließ- und Ganzzahlen

Natürlich kann die Konvertierung `double ↔ long` nicht verlustfrei sein. Wie sollte das auch gehen? Zwar verfügt sowohl ein `long` als auch ein `double` über 64 Bit zur Datenspe-

cherung, aber ein `double` kann eine Ganzzahl nicht so effizient speichern wie ein `long` und hat etwas »Overhead« für einen großen Exponenten. Bei der impliziten Konvertierung eines `long` in ein `double` können einige Bit als Informationsträger herausfallen, wie das folgende Beispiel illustriert:

```
long l = 111111111111111111L; // 111111111111111111
double d = l;                // 111111111111111110 (1.111111111111117E18)
long m = (long) d;           // 111111111111111116
```

Java erlaubt ohne explizite Anpassung die Konvertierung eines `long` an ein `double` und auch an ein noch kleineres `float`, was vielleicht noch merkwürdiger ist, da `float` nur eine Genauigkeit von 6 bis 7 Stellen hat, `long` hingegen 18 Stellen hat.

```
long l = 1000000000000000000L;
float f = l;
System.out.printf( "%f", f );    // 99999984306749440,000000
```

Materialverlust durch Überläufe *

Überläufe bei Berechnungen können zu schwerwiegenden Fehlern führen, so wie beim Absturz der Ariane 5 am 4. Juni 1996, genau 36,7 Sekunden nach dem Start. Die europäische Raumfahrtbehörde European Space Agency (ESA) hatte die unbemannte Rakete, die vier Satelliten an Bord hatte, von Französisch-Guayana aus gestartet. Glücklicherweise kamen keine Menschen ums Leben, doch der materielle Schaden belief sich auf etwa 500 Millionen US-Dollar. In dem Projekt steckten zusätzlich Entwicklungskosten von etwa 7 Milliarden US-Dollar. Der Grund für den Absturz war ein Rundungsfehler, der durch die Umwandlung einer 64-Bit-Fließkommazahl (die horizontale Geschwindigkeit) in eine vorzeichenbehaftete 16-Bit-Ganzzahl auftrat. Die Zahl war leider größer als $2^{15} - 1$ und die Umwandlung nicht gesichert, da die Programmierer diesen Zahlenbereich nicht angenommen hatten. Als Konsequenz brach das Lenksystem zusammen, und die Selbstzerstörung wurde ausgelöst, da die Triebwerke abzubrechen drohten. Das wirklich Dumme an dieser Geschichte ist, dass die Software nicht unbedingt für den Flug notwendig war und nur den Startvorbereitungen diente. Im Fall einer Unterbrechung während des Countdowns hätte das Programm schnell abgebrochen werden können. Ungünstig war, dass der Programmteil unverändert durch Wiederverwendung per Copy & Paste aus der Ariane-4-Software kopiert worden war, die Ariane 5 aber schneller flog.

18.1.5 byte als vorzeichenlosen Datentyp nutzen

Ein byte kann zwar automatisch in einen int konvertiert werden, aber durch den beschränkten Wertebereich eines byte kann nicht jedes int in einem byte Platz finden. So muss bei Zahlen, die nicht im Wertebereich -128 bis +127 liegen, eine explizite Typanpassung durchgeführt werden. Nun lassen sich zwei Fälle unterscheiden: Der in ein byte eingewöngte Wert ist kleiner gleich 255 oder echt größer als 255. Ist die Zahl wirklich größer gleich 256, so gehen Bits verloren, denn mehr als 8 Bit kann ein Byte nicht aufnehmen. Liegt die Zahl jedoch zwischen +127 und +255, so kann das byte prinzipiell das gegebene Bitmuster annehmen, und das Vorzeichenbit kann zur Speicherung verwendet werden. In der Konsolenausgabe sieht das dann merkwürdig aus, da die Zahlen negativ sind, aber das Bitmuster ist korrekt. Das folgende Beispiel zeigt das (angenommen System und Integer sind statisch importiert):

```
byte b = (byte) 255;
int i = 255;
out.printf( "%d %s%n", b, toBinaryString(b) ); // -1 11111111111111111111111111111111
out.printf( "%d %s%n", i, toBinaryString(i) ); // 255 11111111
```

Die Belegung der unteren 8 Bit von b und i ist identisch.

Um bei der Konsolenausgabe einen Datenwert zwischen 0 und 255 zu bekommen, also das Byte vorzeichenlos zu sehen, schneiden wir mit der Und-Verknüpfung die unteren 8 Bit heraus – alle anderen Bits bleiben also ausgenommen:

```
static int byteToInt( byte b )
{
    return b & 0xff;
}
```

Eine explizite Typanpassung mit (int)(b & 0xff) ist nicht nötig, da der Compiler bei der arithmetischen Und-Operation automatisch in ein int konvertiert. Damit lässt sich für unser b die 255 erfragen:

```
byte b = (byte) 255; // oder byte b = 255y; seit Java 7
System.out.println( byteToInt( b ) ); // 255
```

Konvertierungen von byte in ein char

Mit einer ähnlichen Arbeitsweise können wir auch die Frage lösen, wie sich ein Byte, dessen Integerwert im Minusbereich liegt, in ein `char` konvertieren lässt. Der erste Ansatz über eine Typumwandlung (`char`) `byte` ist falsch, und auf der Ausgabe dürfte nur ein rechteckiges Kästchen oder ein Fragezeichen erscheinen:

```
byte b = (byte) 'ß';
System.out.println( (char) b );           // Ausgabe ist ?
```

Das Dilemma ist wieder die fehlerhafte Vorzeichenanpassung. Bei der Benutzung des Bytes wird es zuerst in ein `int` konvertiert. Das »ß« wird dann zu `-33`. Im nächsten Schritt wird diese `-33` dann zu einem `char` umgesetzt. Das ergibt `65.503`, was einen Unicode-Bereich trifft, der zurzeit kein Zeichen definiert. Es wird wohl auch noch etwas dauern, bis die ersten Außerirdischen uns neue Zeichensätze schenken. Gelöst wird der Fall wie oben, indem von `b` nur die unteren 8 Bit betrachtet werden. Das geschieht wieder durch ein Ausblenden über den Und-Operator. Damit ergibt sich korrekt:

```
char c = (char) (b & 0x00ff);
System.out.println( c );                 // Ausgabe ist ß
```

18.1.6 Die Verschiebeoperatoren

Unter Java gibt es drei *Verschiebeoperatoren* (engl. *shift-operators*), die die Bits eines Wertes um eine gewisse Anzahl Positionen verschieben können:

- `n << s`. Linksverschieben der Bits von `n` um `s` Positionen
- `n >> s`. Arithmetisches Rechtsverschieben um `s` Positionen mit Vorzeichen
- `n >>> s`. Logisches Rechtsverschieben um `s` Positionen ohne Vorzeichen

Die binären Verschiebeoperatoren bewegen alle Bits eines Datenworts (das Bitmuster) nach rechts oder links. Bei der Verschiebung steht nach dem binären Operator, also im rechten Operanden, die Anzahl an Positionen, um die verschoben wird. Obwohl es nur zwei Richtungen gibt, muss noch der Fall betrachtet werden, ob das Vorzeichen bei der Rechtsverschiebung beachtet wird oder nicht. Das wird dann *arithmetisches Verschieben* (Vorzeichen verschiebt sich mit) oder *logisches Verschieben* (Vorzeichen wird mit 0 aufgefüllt) genannt.

n << s

Die Bits des Operanden n werden unter Berücksichtigung des Vorzeichens s -mal nach links geschoben (mit 2 multipliziert). Der rechts frei werdende Bit-Platz wird immer mit 0 aufgefüllt. Das Vorzeichen ändert sich jedoch, sobald eine 1 von der Position $MSB - 1$ nach MSB geschoben wird (MSB steht hier für *Most Significant Bit*, also das Bit mit der höchsten Wertigkeit in der binären Darstellung).

Hinweis

Zwar ist der Datentyp des rechten Operators erst einmal ein `int` beziehungsweise `long` mit vollem Wertebereich, doch als Verschiebepositionen sind bei `int` nur Werte bis 31 sinnvoll und für ein `long` Werte bis 63 Bit, da nur die letzten 5 beziehungsweise 6 Bit berücksichtigt werden. Sonst wird immer um den Wert verschoben, der sich durch das Teilen durch 32 beziehungsweise 64 als Rest ergibt, sodass $x \ll 32$ und $x \ll 0$ auch gleich ist.

```
System.out.println( 1 << 30 ); // 1073741824
System.out.println( 1 << 31 ); // -2147483648
System.out.println( 1 << 32 ); // 1
```

n >> s (arithmetisches Rechtsschieben)

Beim Verschieben nach rechts wird je nachdem, ob das Vorzeichen-Bit gesetzt ist oder nicht, eine 1 oder eine 0 von links eingeschoben; das linke Vorzeichen-Bit bleibt unberührt.

18

Beispiel

Ein herausgeschobenes Bit ist für immer verloren!

```
System.out.println( 65535 >> 8 ); // 255
System.out.println( 255 << 8 ); // 65280
```

Es ist $65.535 = 0xFFFF$, und nach der Rechtsverschiebung $65.535 >> 8$ ergibt sich $0x00FF = 255$. Schieben wir nun wieder nach links, also $0x00FF << 8$, dann ist das Ergebnis $0xFF00 = 65.280$.

Bei den Ganzzahldatentypen folgt unter Berücksichtigung des immer vorhandenen Vorzeichens bei normalen Rechtsverschiebungen eine vorzeichenrichtige Ganzzahldivision durch 2.

n >>> s (logisches Rechtsschieben)

Der Operator `>>>` berücksichtigt das Vorzeichen der Variablen nicht, sodass eine vorzeichenlose Rechtsverschiebung ausgeführt wird. So werden auf der linken Seite (MSB) nur Nullen eingeschoben; das Vorzeichen wird mitgeschoben.

zB Beispiel

Mit den Verschiebe-Operatoren lassen sich die einzelnen Bytes eines größeren Datentyps, etwa eines 4 Byte großen `int`, einfach extrahieren:

```
byte b1 = (byte)(v >>> 24),
b2 = (byte)(v >>> 16),
b3 = (byte)(v >>> 8),
b4 = (byte)(v );
```

Bei einer positiven Zahl hat dies keinerlei Auswirkungen, und das Verhalten ist wie beim `>>`-Operator.

zB Beispiel

Die Ausgabe ist für den negativen Operanden besonders spannend:

```
System.out.println( 64 >>> 1 ); // 32
System.out.println( -64 >>> 1 ); // 2147483616
```

Ein `<<<`-Operator ergibt keinen Sinn, da die Linksverschiebung ohnehin nur Nullen rechts einfügt.

18.1.7 Ein Bit setzen, löschen, umdrehen und testen

Die Bit-Operatoren lassen sich zusammen mit den Verschiebeoperatoren gut dazu verwenden, ein Bit zu setzen respektive herauszufinden, ob ein Bit gesetzt ist. Betrachten wir die folgenden Methoden, die ein bestimmtes Bit setzen, abfragen, invertieren und löschen:

```
static int setBit( int n, int pos )
{
    return n | (1 << pos);
}
```

```

static int clearBit( int n, int pos )
{
    return n & ~(1 << pos);
}

static int flipBit( int n, int pos )
{
    return n ^ (1 << pos);
}

static boolean testBit( int n, int pos )
{
    int mask = 1 << pos;

    return (n & mask) == mask;
    // alternativ: return (n & 1<<pos) != 0;
}

```

18.1.8 Bit-Methoden der Integer- und Long-Klasse

Die Klassen `Integer` und `Long` bieten eine Reihe von statischen Methoden zur Bit-Manipulation und zur Abfrage diverser Bit-Zustände von ganzen Zahlen. Die Schreibweise `int|long` kennzeichnet durch `int` die statischen Methoden der Klasse `Integer` und durch `long` die statischen Methoden der Klasse `Long`.

```

final class java.lang.Integer | java.lang.Long
extends Number
implements Comparable<Integer> | implements Comparable<Long>

```

- `static int Integer.bitCount(long i)`
- `static int Long.bitCount(long i)`
Liefert die Anzahl gesetzter Bits.
- `static int Integer.reverse(int i)`
- `static long Long.reverse(long i)`
Dreht die Reihenfolge der Bits um.
- `static int Integer.reverseBytes(int i)`

- static long Long.reverseBytes(long i)
Setzt die Bytes in die umgekehrte Reihenfolge, also das erste Byte an die letzte Position, das zweite Byte an die zweitletzte Position usw.
- static int Integer.rotateLeft(int i, int distance)
- static long Long.rotateLeft(long i, int distance)
- static int Integer.rotateRight(int i, int distance)
- static long Long.rotateRight(long i, int distance)
Rotiert die Bits um distance Positionen nach links oder nach rechts.
- static int Integer.highestOneBit(int i)
- static long Long.highestOneBit(long i)
- static int Integer.lowestOneBit(int i)
- static long lowestOneBit(long i)
Liefert einen Wert, wobei nur das höchste (links stehende) beziehungsweise niedrigste (rechts stehende) Bit gesetzt ist. Es ist also nur höchstens 1 Bit gesetzt; beim Argument 0 ist natürlich kein Bit gesetzt und das Ergebnis ebenfalls null.
- static int Integer.numberOfLeadingZeros(int i)
- static long Long.numberOfLeadingZeros(long i)
- static int Integer.numberOfTrailingZeros(int i)
- static long Long.numberOfTrailingZeros(long i)
Liefert die Anzahl der Null-Bits vor dem höchsten beziehungsweise nach dem niedrigsten gesetzten Bit.

zB Beispiel

Anwendung der statischen Bit-Methoden der Klasse Long.

Statische Methode der Klasse Long	Methodenergebnis
Long.highestOneBit(8 - 1)	4
Long.lowestOneBit(2 + 4)	2
Long.numberOfLeadingZeros(Long.MAX_VALUE)	1
Long.numberOfLeadingZeros(-Long.MAX_VALUE)	0
Long.numberOfTrailingZeros(16)	4
Long.numberOfTrailingZeros(3)	0

Tabelle 18.7: Statische Methoden von »Long«

zB

Beispiel (Forts.)

Statische Methode der Klasse Long	Methodenergebnis
Long.bitCount(8 + 4 + 1)	3
Long.rotateLeft(12, 1)	24
Long.rotateRight(12, 1)	6
Long.reverse(Long.MAX_VALUE)	-2
Long.reverse(0x0F00000000000000L)	240
Long.reverseBytes(0x0F00000000000000L)	15

Tabelle 18.7: Statische Methoden von »Long« (Forts.)

18.2 Fließkommaarithmetik in Java

Zahlen mit einem Komma nennen sich Gleitkomma-, Fließkomma-, Fließpunkt- oder Bruchzahlen (gebrochene Zahlen). Der Begriff »Gleitkommazahl« kommt daher, dass die Zahl durch das Gleiten (Verschieben) des Dezimalpunkts als Produkt aus einer Zahl und einer Potenz der Zahl 10 dargestellt wird (also $1,23 = 123 \cdot 10^{-2}$).

Java unterstützt für Fließkommazahlen die Typen `float` und `double`, die sich nach der Spezifikation IEEE 754 richten. Diesen Standard des *Institute of Electrical and Electronics Engineers* gibt es seit Mitte der 1980er-Jahre. Ein `float` hat die Länge von 32 Bit und ein `double` die Länge von 64 Bit. Die Rechenoperationen sind im *IEEE Standard for Floating-Point Arithmetic* definiert.

18

Hinweis

Wir sollten uns bewusst sein, dass die Genauigkeit von `float` wirklich nicht so toll ist. Schnell beginnt die Ungenauigkeit zuzuschlagen:

```
System.out.println( 2345678.88f ); // 2345679.0
```

18.2.1 Spezialwerte für Unendlich, Null, NaN

Die Datentypen `double` und `float` können nicht nur »Standardzahlen« speichern. Java definiert Sonderwerte für eine positive oder negative Null, positives und negatives Unendlich (engl. *infinity*) und *NaN*, die Abkürzung für *Not a Number*.

Unendlich

Der Überlauf führt zu einem positiven oder negativen Unendlich.

zB

Beispiel

Multiplikation zweier wirklich großer Werte:

```
System.out.println( 1E300 * 1E20 );    // Infinity
System.out.println( -1E300 * 1E20 );   // -Infinity
```

Für die Werte deklariert die Java-Bibliothek in Double und Float zwei Konstanten; zusammen mit der größten und kleinsten darstellbaren Fließkommazahl sind das:

Wert für	Float	Double
positiv unendlich	Float.POSITIVE_INFINITY	Double.POSITIVE_INFINITY
negativ unendlich	Float.NEGATIVE_INFINITY	Double.NEGATIVE_INFINITY
kleinster Wert	Float.MIN_VALUE	Double.MIN_VALUE
größter Wert	Float.MAX_VALUE	Double.MAX_VALUE

Tabelle 18.8: Spezialwerte und ihre Konstanten

Das Minimum für double-Werte liegt bei etwa 10^{-324} und das Maximum bei etwa 10^{308} . Weiterhin deklarieren Double und Float Konstanten für MAX_EXPONENT/MIN_EXPONENT.



Hinweis

Die Anzeige des Über-/Unterlaufs und des undefinierten Ergebnisses gibt es nur bei Fließkommazahlen, nicht aber bei Ganzzahlen.

Positive, negative Null

Es gibt eine positive Null (+0,0) und eine negative Null (-0,0), die etwa beim Unterlauf auftauchen.

zB

Beispiel

Der Unterlauf erzeugt:

```
System.out.println( 1E-322 * 0.0001 ); // 0.0
System.out.println( 1E-322 * -0.0001 ); // -0.0
```

Für den Vergleichsoperator `==` ist die positive Null gleich der negativen Null, sodass `0.0 == -0.0` das Ergebnis `true` ergibt. Damit ist auch `0.0 > -0.0` falsch. Die Bitmaske ist jedoch unterscheidbar, was der Vergleich `Double.doubleToLongBits(+0.0) != Double.doubleToLongBits(-0.0)` zeigt.

Es gibt einen weiteren kleinen Unterschied, den die Rechnung `1.0 / -0.0` und `1.0 / 0.0` zeigt. Durch den Grenzwert geht das Ergebnis einmal gegen negativ unendlich und einmal gegen positiv unendlich.

NaN

`NaN` wird als Fehlerindikator für das Ergebnis von undefinierten Rechenoperationen benutzt, etwa `0/0`.

Beispiel

zB

Erzeuge `NaN` durch den Versuch, die Wurzel einer negativen Zahl zu bilden, und durch eine Nullkommanix-Division:

```
System.out.println( Math.sqrt(-4) );      // NaN
System.out.println( 0.0 / 0.0);           // NaN
```

`NaN` ist als Konstante in den Klassen `Double` und `Float` deklariert. Die statische Methode `isNaN()` testet, ob eine Zahl `NaN` ist. Die API-Dokumentation am Beispiel von `Double` sieht so aus:

18

```
public final class java.lang.Double
extends Number
implements Comparable<Double>
■ public static final double NaN = 0.0 / 0.0;
Deklaration von NaN bei Double.
■ boolean isNaN()
Liefert true, wenn das aktuelle Double-Objekt NaN ist.
■ public static boolean isNaN(double v)
Liefert true, wenn die übergebene Zahl v NaN ist.
```

Die Implementierung von `isnan(double v)` ist einfach: `return v != v;`

Alles hat seine Ordnung *

Außer für den Wert NaN ist auf allen Fließkommazahlen eine totale Ordnung definiert. Das heißt, sie lassen sich von der kleinsten Zahl bis zur größten aufzählen. Am Rand steht die negative Unendlichkeit, dann folgen die negativen Zahlen, negative Null, positive Null, positive Zahlen und positives Unendlich. Bleibt nur noch die einzige unsortierte Zahl NaN. Alle numerischen Vergleiche $<$, \leq , $>$, \geq mit der Java-NaN liefern false. Der Vergleich mit == ist false, wenn einer der Operanden NaN ist. != verhält sich umgekehrt, ist also true, wenn einer der Operanden NaN ist.

zB**Beispiel**

NaN beim Gleichheitstest:

```
System.out.println( Double.NaN == Double.NaN );      // false
System.out.println( Double.NaN != Double.NaN );     // true
```

Da NaN nicht zu sich selbst gleich ist, wird die folgende Konstruktion, die üblicherweise eine Endloschleife darstellt, mit d als Double.NaN einfach übersprungen:

```
while ( d == d ) {}
```

Ein NaN-Wert auf eine Ganzzahl angepasst, also etwa `(int) Double.NaN`, ergibt 0.

Stille NaNs *

Eine Problematik in der Fließkomma-Arithmetik ist, dass keine Ausnahmen die Fehler anzeigen; NaNs solcher Art heißen auch *stille NaNs* (engl. *Quiet NaNs [qNaNs]*). Als Entwickler müssen wir also immer selbst schauen, ob das Ergebnis während einer Berechnung korrekt bleibt. Ein durchschnittlicher numerischer Prozessor unterscheidet ein qNaN und ein *signaling NaN* (*sNaN*).

18.2.2 Standard-Notation und wissenschaftliche Notation bei Fließkommazahlen *

Zur Darstellung der Fließkommaliterale gibt es zwei Notationen: Standard und wissenschaftlich. Die *wissenschaftliche Notation* ist eine Erweiterung der Standardnotation. Bei ihr folgt hinter den Nachkommastellen ein »E« (oder »e«) mit einem Exponenten zur Basis 10. Der Vorkommaanteil darf durch die Vorzeichen »+« oder »-« eingeleitet wer-

den. Auch der Exponent kann positiv oder negativ² sein, muss aber eine Ganzzahl sein. Die Tabelle stellt drei Beispiele zusammen:

Standard	Wissenschaftlich
123450.0	1.2345E5
123450.0	1.2345E+5
0.000012345	1.2345E-5

Tabelle 18.9: Notationen der Fließkommazahlen

Beispiel

Nutzen der wissenschaftlichen Notation:

```
double x = 3.00e+8;
float y = 3.00E+8F;
```

zB

18.2.3 Mantisse und Exponent *

Intern bestehen Fließkommazahlen aus drei Teilen: einem Vorzeichen, einem ganzzahligen *Exponenten* und einer *Mantisse* (engl. *mantissa*). Während die Mantisse die Genauigkeit bestimmt, gibt der Exponent die Größenordnung der Zahl an.

Die Berechnung für Fließkommazahlen aus den drei Elementen ist im Prinzip wie folgt: *Vorzeichen* \times *Mantisse* \times 2^{Exponent} , wobei Vorzeichen -1 oder $+1$ sein kann. Die Mantisse m ist keine Zahl mit beliebigem Wertebereich, sondern normiert mit dem Wertebereich $1 \leq m < 2$, also eine Fließkommazahl, die mit 1 beginnt und daher auch *1-plus-Form* heißt.³ Auch der zunächst vorzeichenbehaftete Exponent wird nicht direkt gespeichert, sondern als *angepasster Exponent* (engl. *biased exponent*) in der IEEE-kodierten Darstellung abgelegt. Zu unserem Exponenten wird, abhängig von der Genauigkeit, $+127$ (bei *float*) und $+1023$ (bei *double*) addiert; nach der Berechnung steht in der Darstellung immer eine ganze Zahl. 127 und 1023 nennen sich *Bias*.

Das Vorzeichen kostet immer 1 Bit, und die Anzahl der Bits für Exponent und Mantisse richtet sich nach dem Datentyp.

18

2 LOGO verwendet für negative Exponenten den Buchstaben N anstelle des E. In Java steht das E mit einem folgenden unären Plus- oder Minuszeichen.

3 Es gibt eine Ausnahme durch denormalisierte Zahlen, aber das spielt für das Verständnis keine Rolle.

Datentyp	Anzahl Bits für den Exponenten	Anzahl Bits für die Mantisse
float	8	23
double	11	52

Tabelle 18.10: Anzahl der Bits für Exponent und Mantisse

zB**Beispiel**

Das Folgende sind Kodierungen für die Zahl 123456,789 als float und double. Das »..« trennt Vorzeichen, Exponent und Mantisse:

0·10001111·11100010010000001100101

0·10000001111·111000100100000011001001111100111011011001001

Um von dieser Darstellung auf die Zahl zu kommen, schreiben wir:

```
BigInteger biasedExponent = new BigInteger( "10001111", 2 );
BigInteger mantisse = new BigInteger( "11100010010000001100101", 2 );
int exponent = (int) Math.pow( 2, biasedExponent.longValue() - 127 );
double m = 1. + (mantisse.longValue() / Math.pow( 2, 23 ));
System.out.println( exponent * m ); // 123456.7890625
```

Den Exponenten (ohne Bias) einer Fließkommazahl liefert `Math.getExponent()`; auf unsere Zahl angewendet, ist das also 16.

Der Exponent einer Fließkommazahl liefert auch die Methode `getExponent()` der Klassen `Double` und `Float`.

Zugang zum Bitmuster liefern die Methoden `long doubleToLongBits(double)` beziehungsweise `int Float.floatToIntBits(float)`. Die Umkehrung ist `double Double.longBitsToDouble(long)` beziehungsweise `float Float.intBitsToFloat(int)`.

18.3 Die Eigenschaften der Klasse Math

Die Klasse `java.lang.Math` ist eine typische Utility-Klasse, die nur statische Methoden (beziehungsweise Attribute als Konstanten) deklariert. Mit dem privaten Konstruktor lassen sich (so leicht) keine Exemplare von `Math` erzeugen.



Abbildung 18.1: UML-Diagramm der Klasse »Math«

18.3.1 Attribute

Die `Math`-Klasse besitzt zwei statische Attribute:

```
class java.lang.Math
■ static final double E
    Die Eulersche Zahl4 e = 2.7182818284590452354
■ static final double PI
    Die Kreiszahl Pi5 = 3.14159265358979323846
```

18.3.2 Absolutwerte und Vorzeichen

Die zwei statischen `abs()`-Methoden liefern den Betrag des Arguments (mathematische Betragsfunktion: $y = |x|$). Sollte ein negativer Wert als Argument übergeben werden, wandelt ihn `abs()` in einen positiven Wert um.

Eine spezielle Methode ist auch `copySign()`. Sie ermittelt das Vorzeichen einer Fließkommazahl und setzt dieses Vorzeichen bei einer anderen.

```
class java.lang.Math
■ static int abs(int x)
■ static long abs(long x)
■ static float abs(float x)
■ static double abs(double x)
■ static double copySign(double magnitude, double sign)
■ static float copySign(float magnitude, float sign)
    Liefert magnitude als Rückgabe, aber mit dem Vorzeichen von sign.
```

⁴ Die irrationale Zahl e ist nach dem schweizerischen Mathematiker Leonhard Euler (1707–1783) benannt.

⁵ Wer noch auf der Suche nach einer völlig unsinnigen Information ist: Die einmilliardste Stelle hinter dem Komma von Pi ist eine Neun.



Hinweis

Es gibt genau einen Wert, auf den `Math.abs(int)` keine positive Rückgabe liefern kann: `-2147483648`. Dies ist die kleinste darstellbare int-Zahl (`Integer.MIN_VALUE`), während `+2147483648` gar nicht in ein int passt! Die größte darstellbare int-Zahl ist `2147483647` (`Integer.MAX_VALUE`). Was sollte `abs(-2147483648)` auch ergeben?

Vorzeichen erfragen

Die statische Methode `signum(value)` liefert eine numerische Rückgabe für das Vorzeichen von `value`, und zwar `+1` für positive, `-1` für negative Zahlen, und `0` für `0`. Die Methode ist nicht ganz logisch auf die Klassen `Math` für Fließkommazahlen und `Integer/Long` für Ganzzahlen verteilt:

- `java.lang.Integer.signum(int i)`
- `java.lang.Long.signum(long i)`
- `java.lang.Math.signum(double d)`
- `java.lang.Math.signum(float f)`

18.3.3 Maximum/Minimum

Die statischen `max()`-Methoden liefern den größeren der übergebenen Werte. Die statischen `min()`-Methoden liefern den kleineren von zwei Werten als Rückgabewert.

```
class java.lang.Math
```

- `static int max(int x, int y)`
- `static long max(long x, long y)`
- `static float max(float x, float y)`
- `static double max(double x, double y)`
- `static int min(int x, int y)`
- `static long min(long x, long y)`
- `static float min(float x, float y)`
- `static double min(double x, double y)`

18.3.4 Runden von Werten

Für die Rundung von Werten bietet die Klasse `Math` fünf statische Methoden:

```
class java.lang.Math
    static double ceil(double a)
    static double floor(double a)
    static int round(float a)
    static long round(double a)
    static double rint(double a)
```

Auf- und Abrunden mit `ceil()` und `floor()`

Die statische Methode `ceil()` dient zum Aufrunden und liefert die nächsthöhere Ganzzahl (jedoch als `double`, nicht als `long`), wenn die Zahl nicht schon eine ganze Zahl ist; die statische Methode `floor()` rundet auf die nächstniedrigere Ganzzahl ab:

Listing 18.1: RoundingDemo.java, Ausschnitt

```
System.out.println( Math.ceil(-99.1) );      // -99.0
System.out.println( Math.floor(-99.1) );     // -100.0
System.out.println( Math.ceil(-99) );         // -99.0
System.out.println( Math.floor(-99) );        // -99.0
System.out.println( Math.ceil(-.5) );          // -0.0
System.out.println( Math.floor(-.5) );         // -1.0
System.out.println( Math.ceil(-.01) );         // -0.0
System.out.println( Math.floor(-.01) );        // -1.0
System.out.println( Math.ceil(0.1) );          // 1.0
System.out.println( Math.floor(0.1) );         // 0.0
System.out.println( Math.ceil(.5) );           // 1.0
System.out.println( Math.floor(.5) );          // 0.0
System.out.println( Math.ceil(99) );           // 99.0
System.out.println( Math.floor(99) );          // 99.0
```

Die genannten statischen Methoden haben auf ganze Zahlen keine Auswirkung.

Kaufmännisches Runden mit round()

Die statischen Methoden `round(double)` und `round(float)` runden kaufmännisch auf die nächste Ganzzahl vom Typ `long` beziehungsweise `int`. Ganze Zahlen werden nicht aufgerundet. Wir können `round()` als Gegenstück zur Typumwandlung (`long`) `doublevalue` einsetzen:

Listing 18.2: RoundingDemo.java, Ausschnitt

```
System.out.println( Math.round(1.01) );      // 1
System.out.println( Math.round(1.4) );        // 1
System.out.println( Math.round(1.5) );        // 2
System.out.println( Math.round(1.6) );        // 2
System.out.println( (int) 1.6 );              // 1
System.out.println( Math.round(30) );          // 30
System.out.println( Math.round(-2.1) );        // -2
System.out.println( Math.round(-2.9) );        // -3
System.out.println( (int) -2.9 );             // -2
```

Beispiel

zB

Die `Math.round()`-Methode ist in Java ausprogrammiert. Sie addiert auf den aktuellen Parameter 0,5 und übergibt das Ergebnis der statischen `floor()`-Methode:

```
public static long round( double a ) {
    return (int) floor( a + 0.5f );
}
```

18

Gerechtes Runden rint()

`rint()` ist mit `round()` vergleichbar, nur ist es im Gegensatz zu `round()` gerecht, was bedeutet, dass `rint()` bei 0,5 in Abhängigkeit davon, ob die benachbarte Zahl ungerade oder gerade ist, auf- oder abrundet:

Listing 18.3: RoundingDemo.java, Ausschnitt

```
System.out.println( Math.round(-1.5) );      // -1
System.out.println( Math.rint( -1.5 ) );       // -2.0
System.out.println( Math.round(-2.5) );        // -2
System.out.println( Math.rint( -2.5 ) );       // -2.0
System.out.println( Math.round( 1.5 ) );        // 2
```

```
System.out.println( Math.rint( 1.5) );      // 2.0
System.out.println( Math.round( 2.5 ) );     // 3
System.out.println( Math.rint( 2.5 ) );      // 2.0
```

Mit einem konsequenten Auf- oder Abrunden pflanzen sich natürlich auch Fehler ungeschickter fort als mit dieser 50/50-Strategie.

zB Beispiel

Die statische `rint()`-Methode lässt sich auch einsetzen, wenn Zahlen auf zwei Nachkommastellen gerundet werden sollen. Ist `d` vom Typ `double`, so ergibt der Ausdruck `Math.rint(d*100.0)/100.0` die gerundete Zahl.

Listing 18.4: Round2Scales.java

```
class Round2Scales
{
    public static double roundScale2( double d )
    {
        return Math.rint( d * 100 ) / 100.;

    }

    public static void main( String[] args )
    {
        System.out.println( roundScale2(+1.341) );      // 1.34
        System.out.println( roundScale2(-1.341) );      // -1.34
        System.out.println( roundScale2(+1.345) );      // 1.34
        System.out.println( roundScale2(-1.345) );      // -1.34

        System.out.println( roundScale2(+1.347) );      // 1.35
        System.out.println( roundScale2(-1.347) );      // -1.35
    }
}
```

Arbeiten wir statt mit `rint()` mit `round()`, wird die Zahl 1,345 nicht auf 1,34, sondern auf 1,35 gerundet. Wer nun Lust hat, etwas auszuprobieren, darf testen, wie der Formatstring »%.2f« bei `printf()` rundet.

18.3.5 Wurzel- und Exponentialmethoden

Die Math-Klasse bietet weiterhin Methoden zum Berechnen der Wurzel und weitere Exponentialmethoden.

```
class java.lang.Math
```

- static double sqrt(double x)
Liefert die Quadratwurzel von x; sqrt steht für *square root*.
- static double cbrt(double a)
Berechnet die dritte Wurzel aus a.
- static double hypot(double x, double y)
Berechnet die Wurzel aus $x^2 + y^2$, also den euklidischen Abstand. Könnte als $\sqrt{x*x + y*y}$ umgeschrieben werden, doch hypot() bietet eine bessere Genauigkeit und Performance.
- static double scalb(double d, double scaleFactor)
Liefert d mal 2 hoch scaleFactor. Kann prinzipiell auch als d * Math.pow(2, scaleFactor) geschrieben werden, doch scalb() bietet eine bessere Performance.
- static double exp(double x)
Liefert den Exponentialwert von x zur Basis e (der Eulerschen Zahl e = 2,71828...), also e^x .
- static double expm1(double x)
Liefert den Exponentialwert von x zur Basis e minus 1, also $e^x - 1$. Berechnungen nahe null kann expm1(x) + 1 präziser ausdrücken als exp(x).
- static double pow(double x, double y)
Liefert den Wert der Potenz x^y . Für ganzzahlige Werte gibt es keine eigene Methode.

Die Frage nach dem 0.0/0.0 und 0.0^{0.0} *

Wie wir wissen, ist 0.0/0.0 ein glattes NaN. Im Unterschied zu den Ganzzahlwerten bekommen wir hier allerdings keine Exception, denn dafür ist extra die Spezialzahl NaN eingeführt worden. Interessant ist die Frage, was denn (long)(double)(0.0/0.0) ergibt. Die Sprachdefinition sagt hier in §5.1.3, dass die Konvertierung eines Fließkommawerts NaN in ein int 0 oder long 0 ergibt.⁶

⁶ Leider gab es in den ersten Versionen der JVM einen Fehler, sodass Long.MAX_VALUE anstelle von 0.0 produziert wurde. Dieser Fehler ist aber inzwischen behoben.

Eine weitere spannende Frage ist das Ergebnis von $0.0^{0.0}$. Um allgemeine Potenzen zu berechnen, wird die statische Funktion `Math.pow(double a, double b)` eingesetzt. Wir erinnern uns aus der Schulzeit daran, dass wir die Quadratwurzel einer Zahl ziehen, wenn der Exponent b genau $1/2$ ist. Doch jetzt wollen wir wissen, was denn gilt, wenn $a = b = 0$ gilt. §20.11.13 der Sprachdefinition fordert, dass das Ergebnis immer 1.0 ist, wenn der Exponent b gleich -0.0 oder 0.0 ist. Es kommt also in diesem Fall überhaupt nicht auf die Basis a an. In einigen Algebra-Büchern wird 0^0 als undefiniert behandelt. Es macht aber durchaus Sinn, 0^0 als 1 zu definieren, da es andernfalls viele Sonderbehandlungen für 0 geben müsste.⁷

18.3.6 Der Logarithmus *

Der Logarithmus ist die Umkehrfunktion der Exponentialfunktion. Die Exponentialfunktion und der Logarithmus hängen durch folgende Beziehung zusammen: Ist $y = a^x$, dann ist $x = \log_a(y)$. Der Logarithmus, den `Math.log()` berechnet, ist der natürliche Logarithmus zur Basis e . In der Mathematik wird dieser mit »ln« angegeben (*logarithmus naturalis*). Logarithmen mit der Basis 10 heißen *dekadische* oder *briggsche Logarithmen* und werden mit »lg« abgekürzt; der Logarithmus zur Basis 2 (*binärer Logarithmus, dualer Logarithmus*) wird mit »lb« abgekürzt. In Java gibt es die statische Methode `log10()` für den briggschen Logarithmus lg, nicht aber für den binären Logarithmus lb, der weiterhin nachgebildet werden muss. Allgemein gilt folgende Umrechnung: $\log_b(x) = \log_a(x) / \log_a(b)$.

zB Beispiel

Eine eigene statische Methode soll den Logarithmus zur Basis 2 berechnen:

```
public static double lb( double x )
{
    return Math.log( x ) / Math.log( 2.0 );
}
```

⁷ Hier schreiben die Autoren R. Graham, D. Knuth, O. Patashnik des Buchs *Concrete Mathematics*: »Some textbooks leave the quantity 0^0 undefined, because the functions x^0 and 0^x have different limiting values when x decreases to 0 . But this is a mistake. We must define $x^0 = 1$ for all x , if the binomial theorem is to be valid when $x=0$, $y=0$, and/or $x=-y$. The theorem is too important to be arbitrarily restricted! By contrast, the function 0^x is quite unimportant.«, Addison-Wesley, 1994, ISBN 0-201-55802-5.

zB

Beispiel (Forts.)

Da `Math.log(2)` konstant ist, sollte dieser Wert aus Performance-Gründen in einer Konstanten gehalten werden.

```
class java.lang.Math
```

- static double log(double a)
Berechnet von a den Logarithmus zur Basis e.
- static double log10(double a)
Liefert von a den Logarithmus zur Basis 10.
- static double log1p(double x)
Liefert $\log(x) + 1$.

18.3.7 Rest der ganzzahligen Division *

Neben dem Restwert-Operator `%`, der den Rest der Division berechnet, gibt es auch eine statische Methode `Math.IEEEremainder()`.

Listing 18.5: IEEEremainder.java, main()

```
double a = 44.0;
double b = 2.2;
System.out.println( a / b );           // 20.0
System.out.println( a % b );           // 2.199999999999996
System.out.println( Math.IEEEremainder( a, b ) ); // -3.552713678800501E-15
```

18

Das zweite Ergebnis ist mit der mathematischen Ungenauigkeit fast 2,2, aber etwas kleiner, sodass der Algorithmus nicht noch einmal 2,2 abziehen konnte. Die statische Methode `IEEEremainder()` liefert ein Ergebnis nahe null ($-0.000000000000035527136788005$), was besser ist, denn 44,0 lässt sich ohne Rest durch 2,2 teilen, also wäre der Rest eigentlich 0.

```
class java.lang.Math
```

- static double IEEEremainder(double dividend, double divisor)
Liefert den Rest der Division von Dividend und Divisor, so wie es der IEEE 754-Standard vorschreibt.

Eine eigene statische Methode, die mitunter bessere Ergebnisse liefert – mit den Werten 44 und 2,2 wirklich 0,0 –, ist die folgende:

```
public static double remainder( double a, double b )
{
    return Math.signum(a) *
        (Math.abs(a) - Math.abs(b) * Math.floor(Math.abs(a)/Math.abs(b)));
}
```

18.3.8 Winkelmethoden *

Die `Math`-Klasse stellt einige winkelbezogene Methoden und ihre Umkehrungen zur Verfügung. Im Gegensatz zur bekannten Schulmathematik werden die Winkel für `sin()`, `cos()`, `tan()` im Bogenmaß (2π entspricht einem Vollkreis) und nicht im Gradmaß (360 Grad entspricht einem Vollkreis) übergeben.

■	<code>static double sin(double x)</code>
■	<code>static double cos(double x)</code>
■	<code>static double tan(double x)</code>
	Liefert den Sinus/Kosinus/Tangens von <code>x</code> .

Arcus-Methoden

Die Arcus-Methoden realisieren die Umkehrfunktionen zu den trigonometrischen Methoden. Das Argument ist kein Winkel, sondern zum Beispiel bei `asin()` der Sinuswert zwischen -1 und 1 . Das Ergebnis ist dann ein Winkel im Bogenmaß, etwa zwischen $-\pi/2$ und $\pi/2$.

■	<code>static double asin(double x)</code>
■	<code>static double acos(double x)</code>
■	<code>static double atan(double x)</code>
	Liefert den Arcus-Sinus, Arcus-Kosinus beziehungsweise Arcus-Tangens von <code>x</code> .
■	<code>static atan2(double x, double y)</code>
	Liefert bei der Konvertierung von Rechteckkoordinaten in Polarkoordinaten den Winkel <i>theta</i> , also eine Komponente des Polarkoordinaten-Tupels. Die statische Methode

berücksichtigt das Vorzeichen der Parameter x und y , und der freie Schenkel des Winkels befindet sich im richtigen Quadranten.

Hyperbolicus-Methoden bietet Java über `sinh()`, `tanh()` und `cosh()`.

Umrechnungen von Gradmaß in Bogenmaß

Zur Umwandlung eines Winkels von Gradmaß in Bogenmaß und umgekehrt existieren zwei statische Methoden:

- ```
class java.lang.Math
```
- static double toRadians(double angdeg)  
Wandelt Winkel von Gradmaß in Bogenmaß um.
  - static double toDegrees(double angrad)  
Wandelt Winkel von Bogenmaß in Gradmaß um.

### 18.3.9 Zufallszahlen

Positive Gleitkomma-Zufallszahlen zwischen größer gleich 0,0 und echt kleiner 1,0 liefert die statische Methode `Math.random()`. Die Rückgabe ist `double`, und eine Typanpassung auf `int` führt immer zum Ergebnis 0.

Möchten wir Werte in einem anderen Wertebereich haben, ist es eine einfache Lösung, die Zufallszahlen von `Math.random()` durch Multiplikation (Skalierung) auf den gewünschten Wertebereich auszudehnen und per Addition (ein Offset) geeignet zu verschieben. Um ganzzahlige Zufallszahlen zwischen `min` (inklusiv) und `max` (inklusiv) zu erhalten, schreiben wir:

**Listing 18.6:** RandomIntInRange.java

```
public static long random(long min, long max)
{
 return min + Math.round(Math.random() * (max - min));
}
```

Eine Alternative bietet der direkte Einsatz der Klasse `Random` und der Objektmethode `nextInt(n)`.

## 18.4 Genauigkeit, Wertebereich eines Typs und Überlaufkontrolle \*

Für jeden primitiven Datentyp gibt es in Java eine eigene Klasse mit diversen Methoden und Konstanten. Die Klassen `Byte`, `Short`, `Integer`, `Long`, `Float` und `Double` besitzen die Konstanten `MIN_VALUE` und `MAX_VALUE` für den minimalen und maximalen Wertebereich. Die Klassen `Float` und `Double` verfügen zusätzlich über die wichtigen Konstanten `NEGATIVE_INFINITY` und `POSITIVE_INFINITY` für minus und plus unendlich und `NaN` (Not a Number, undefiniert).



### Hinweis

`Integer.MIN_VALUE` steht mit `-2147483648` für den kleinsten Wert, den die Ganzzahl annehmen kann. `Double.MIN_VALUE` steht jedoch für die kleinste *positive* Zahl (beste Näherung an 0), die ein `Double` darstellen kann (`4.9E-324`).

Wenn uns beim Wort `double` im Vergleich zu `float` eine »doppelte Genauigkeit« über die Lippen kommt, müssen wir mit der Aussage vorsichtig sein, denn `double` bietet zumindest nach der Anzahl der Bits eine mehr als doppelt so präzise Mantisse. Über die Anzahl der Nachkommastellen sagt das jedoch direkt nichts aus.



### Bastelaufgabe

Warum sind die Ausgaben so, wie sie sind?

**Listing 18.7:** DoubleFloatEqual.java, main()

```
double d1 = 0.02d;
float f1 = 0.02f;
System.out.println(d1 == f1); // false
System.out.println((float) d1 == f1); // true

double d2 = 0.02f;
float f2 = 0.02f;
System.out.println(d2 == f2); // true
```

### 18.4.1 Behandlung des Überlaufs

Bei einigen mathematischen Fragestellungen müssen Sie feststellen können, ob Operationen wie die Addition, Subtraktion oder Multiplikation den Zahlenbereich sprengen,

also etwa den Ganzzahlbereich eines Integers von 32 Bit verlassen. Passt das Ergebnis einer Berechnung nicht in den Wertebereich einer Zahl, so wird dieser Fehler nicht von Java angezeigt; weder der Compiler noch die Laufzeitumgebung melden dieses Problem. Es gibt auch keine Ausnahme.

Mathematisch gilt  $a \times a / a = a$ , also zum Beispiel  $100\,000 \times 100\,000 / 100\,000 = 100\,000$ . In Java ist das anders, da wir bei  $100\,000 \times 100\,000$  einen Überlauf im `int` haben.

```
System.out.println(100000 * 100000 / 100000); // 14100
```

liefert daher 14100. Wenn wir den Datentyp auf `long` erhöhen, indem wir hinter ein 100 000 ein `L` setzen, sind wir bei dieser Multiplikation noch sicher, da ein `long` das Ergebnis aufnehmen kann.

```
System.out.println(100000L * 100000 / 100000); // 100000
```

## Überlauf erkennen

Für die Operationen Addition und Subtraktion lässt sich das noch ohne allzu großen Aufwand implementieren. Wir vergleichen dazu zunächst das Ergebnis mit den Konstanten `Integer.MAX_VALUE` und `Integer.MIN_VALUE`. Natürlich muss der Vergleich so umgeformt werden, dass dabei kein Überlauf auftritt, also  $a + b > \text{Integer.MAX\_VALUE}$  ist. Überschreiten die Werte diese maximalen Werte, ist die Operation nicht ohne Fehler möglich, und wir setzen das Flag `canAdd` auf `false`. Hier die Programmzeilen für die Addition:

```
if (a >= 0 && b >= 0)
 if (! (b <= Integer.MAX_VALUE - a))
 canAdd = false;
if (a < 0 && b < 0)
 if (! (b >= Integer.MIN_VALUE - a))
 canAdd = false;
```

Bei der Multiplikation gibt es zwei Möglichkeiten: Zunächst einmal lässt sich die Multiplikation als Folge von Additionen darstellen. Dann ließe sich wiederum der Test mit der Konstanten `Integer.XXX_VALUE` durchführen. Diese Lösung scheidet jedoch wegen der Geschwindigkeit aus. Der andere Weg sieht eine Umwandlung nach `long` vor. Das Ergebnis wird zunächst als `long` berechnet und anschließend mit dem Ganzzahlwert vom Typ `int` verglichen.

Dies funktioniert jedoch nur mit Datentypen, die kleiner als long sind. long selbst fällt heraus, da es keinen Datentyp gibt, der größer ist. Mit ein wenig Rechenengenauigkeit würde ein double jedoch weiterhelfen, und bei präziserer Berechnung kann BigInteger helfen. Bei der Multiplikation im Wertebereich int lässt sich ähnlich wie bei der Addition auch  $b > \text{Integer.MAX\_VALUE} / a$  schreiben. Bei  $b == \text{Integer.MAX\_VALUE} / a$  muss ein Test genau zeigen, ob das Ergebnis in den Wertebereich passt.

Die eigene statische Methode `canMulLong()` soll bei der Frage nach dem Überlauf helfen:

**Listing 18.8:** Overflow.java

```
import java.math.BigInteger;

public class Overflow
{
 private final static BigInteger MAX = BigInteger.valueOf(Long.MAX_VALUE);

 public static boolean canMulLong(long a, long b)
 {
 BigInteger bigA = BigInteger.valueOf(a);
 BigInteger bigB = BigInteger.valueOf(b);

 return bigB.multiply(bigA).compareTo(MAX) <= 0;
 }

 public static void main(String[] args)
 {
 System.out.println(canMulLong(Long.MAX_VALUE/2, 2)); // true
 System.out.println(Long.MAX_VALUE/2 * 2); // 9223372036854775806
 System.out.println(canMulLong(Long.MAX_VALUE/2 + 1, 2)); // false
 System.out.println((Long.MAX_VALUE/2 + 1) * 2); // -9223372036854775808
 }
}
```



### Hinweis

Wenn der Compiler beziehungsweise die JVM genau das macht, was wir uns wünschen, so würde auch ein `return a * b / b == a;` reichen. Doch eine JVM kann das zu `return a == a;` optimieren und somit zu `return true;` machen, sodass der Test nicht funktioniert.

### 18.4.2 Was bitte macht ein ulp?

Die Math-Klasse bietet sehr spezielle Methoden, die für diejenigen interessant sind, die sich sehr genau mit Rechen(un)genauigkeiten beschäftigen und mit numerischen Näherungen arbeiten.

Der Abstand von einer Fließkommazahl zur nächsten ist durch den internen Aufbau nicht immer gleich. Wie groß genau der Abstand einer Zahl zur nächstmöglichen ist, zeigt `ulp(double)` beziehungsweise `ulp(float)`. Der lustige Methodenname ist eine Abkürzung für *Unit in the Last Place*. Was genau denn die nächsthöhere/-niedrigere Zahl ist, ermitteln die Methoden `nextUp(double)/nextUp(float)`, die auf `nextAfter()` zurückgreifen.

#### Beispiel

zB

Was kommt nach und vor 1:

```
System.out.printf("%.16f%n", Math.nextUp(1));
System.out.printf("%.16f%n", Math.nextAfter(1, Double.POSITIVE_INFINITY));
System.out.printf("%.16f%n", Math.nextAfter(1, Double.NEGATIVE_INFINITY));
```

Die Ausgabe ist:

```
1,0000001192092896
1,0000001192092896
0,999999403953552
```

`nextUp()` ist eine Abkürzung wie für den Ausdruck in der zweiten Zeile. Ist das zweite Argument von `Math.nextAfter()` größer als das erste, dann wird die nächstgrößere Zahl zurückgegeben, ist sie kleiner, dann die nächstkleinere Zahl. Bei Gleichheit kommt die gleiche Zahl zurück.

18

Dazu ein weiteres Beispiel: Je größer die Zahlen werden, desto größer werden auch die Sprünge.

| Methode                          | Rückgabe des ulp                 |
|----------------------------------|----------------------------------|
| <code>Math.ulp( 0.00001 )</code> | 0,000000000000000000001694065895 |
| <code>Math.ulp( -1 )</code>      | 0,00000011920928955078125        |
| <code>Math.ulp( 1 )</code>       | 0,00000011920928955078125        |

**Tabelle 18.11:** Da das Vorzeichen in einem extra Bit gespeichert ist, haben negative oder positive Zahlen keine anderen Genauigkeiten.

| Methode           | Rückgabe des ulp         |
|-------------------|--------------------------|
| Math.ulp( 2 )     | 0,0000002384185791015625 |
| Math.ulp( 10E30 ) | 1125899906842624         |

Tabelle 18.11: Da das Vorzeichen in einem extra Bit gespeichert ist, haben negative oder positive Zahlen keine anderen Genauigkeiten. (Forts.)

Die üblichen mathematischen Fließkommaoperationen haben eine ulp von  $\frac{1}{2}$ . Das ist so genau wie möglich. Um wie viel ulp die Math-Methoden vom echten Resultat abweichen können, steht in der JavaDoc. Rechenfehler lassen sich insbesondere bei komplexen Methoden nicht vermeiden. So darf sin() eine mögliche Ungenauigkeit von 1 ulp haben, atan2() von maximal 2 ulp und sinh(), chosh(), tanh() von 2,5 ulp.

Die ulp()-Methode ist für das Testen interessant, denn mit ihr lassen sich Abweichungen immer in der passenden Größenordnung realisieren. Bei kleinen Zahlen ergibt eine Differenz von vielleicht 0,001 einen Sinn, bei größeren Zahlen kann die Toleranz größer sein.

Java deklariert in den Klassen Double und Float drei besondere Konstanten. Sie lassen sich gut mit nextAfter() erklären. Am Beispiel von Double:

- MIN\_VALUE = nextUp(0.0) = Double.longBitsToDouble(0x0010000000000000L)
- MIN\_NORMAL = MIN\_VALUE/(nextUp(1.0)-1.0) = Double.longBitsToDouble(0x1L)
- MAX\_VALUE = nextAfter(POSITIVE\_INFINITY, 0.0) =  
Double.longBitsToDouble(0x7fffffffffffffL)

## 18.5 Mathe bitte strikt \*

Bei der Berechnung mit Fließkommazahlen schreibt die Definition des IEEE 754-Standards vor, wie numerische Berechnungen durchgeführt werden. Damit soll die CPU/FPU für float und double mit 32 beziehungsweise 64 Bit rechnen. In Wirklichkeit rechnet jedoch so gut wie kein mathematischer Prozessor mit diesen Größen, außer vielleicht AMD mit der 3Dnow!-Technologie. Auf der PC-Seite kommen Intel und AMD mit internen Rechengenauigkeiten von 80 Bit, also 10 Byte, zum Zuge. Dieses Dilemma betrifft aber nur 80x86- und andere CISC-Prozessoren. Bei RISC sind 32 Bit und 64 Bit das Übliche. Die 80-Bit-Lösung bringt in Java zwei Nachteile mit sich:

- Diese Genauigkeit kann Java bisher nicht nutzen.
- Wegen der starren IEEE 754-Spezifikation kann der Prozessor weniger Optimierungen durchführen, weil er sich immer eng an die Norm halten muss. Das kostet Zeit. Gegebenenfalls können aber die mathematischen Ergebnisse auf unterschiedlichen Maschinen anders aussehen.

### 18.5.1 Strikte Fließkommaberechnungen mit strictfp

Damit zum einen die Vorgaben der Norm erfüllt werden und zum anderen die Geschwindigkeit gewährleistet werden kann, lässt sich vor Klassen und Methoden der Modifizierer `strictfp` setzen, damit Operationen strikt nach der IEEE-Norm vorgehen. Ohne dieses Schlüsselwort (wie es also für die meisten unserer Programme der Fall ist) nimmt die JVM eine interne Optimierung vor. Nach außen bleiben die Datentypen 32 Bit und 64 Bit lang, das heißt: Bei den Konstanten in `double` und `float` ändert sich nichts. Zwischenergebnisse bei Fließkommaberechnungen werden aber eventuell mit größerer Genauigkeit berechnet.

### 18.5.2 Die Klassen Math und StrictMath

Für strikte mathematische Operationen gibt es eine eigene Klasse `StrictMath`. An der Klassendeklaration für `StrictMath` lässt sich ablesen, dass alle Methoden sich an die IEEE-Norm halten.

**Listing 18.9:** `java.lang.StrictMath.java`, `StrictMath`

```
public final strictfp class StrictMath {
 // ...
}
```

Allerdings gibt es nicht zwei Implementierungen der mathematischen Methoden – einmal strikt und genau beziehungsweise einmal nicht strikt, dafür potenziell schneller. Bisher delegiert die Implementierung für `Math` direkt an `StrictMath`:

**Listing 18.10:** `java.lang.Math.java`, Ausschnitt

```
public final strictfp class Math
{
 public static double tan(double a) {
 return StrictMath.tan(a);
 }
}
```

```

 // default impl. delegates to StrictMath
}
// ...
}

```

Die Konsequenz ist, dass alle Methoden wie `Math.pow()` strikt nach IEEE-Norm rechnen. Das ist zwar aus Sicht der Präzision und Übertragbarkeit der Ergebnisse wünschenswert, aber die Performance ist nicht optimal.

## 18.6 Die Random-Klasse

Neben der Zufallsmethode `Math.random()` in der Klasse `Math` gibt es einen flexibleren Generator für Zufallszahlen im `java.util`-Paket. Dies ist die Klasse `Random`, die aber im Gegensatz zu `Math.random()` keine statischen Funktionen besitzt. Die statische Funktion `Math.random()` nutzt jedoch intern ein `Random`-Objekt.

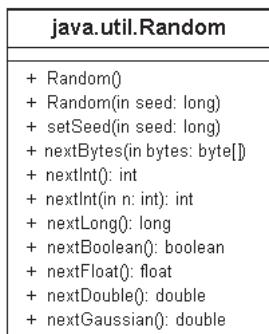


Abbildung 18.2: UML-Diagramm der Klasse Random

### 18.6.1 Objekte mit dem Samen aufbauen

Der Startwert für jede Zufallszahl ist ein 48-Bit-Seed. »Seed« ist das englische Wort für »Samen« und deutet an, dass es bei der Generierung von Zufallszahlen wie bei Pflanzen einen Samen gibt, der zu Nachkommen führt. Aus diesem Startwert ermittelt der Zufallszahlengenerator anschließend die folgenden Zahlen durch lineare Kongruenzen. (Dadurch sind die Zahlen nicht wirklich zufällig, sondern gehorchen einem mathematischen Verfahren. Kryptografisch bessere Zufallszahlen liefert die Klasse `java.security.SecureRandom`, die eine Unterklasse von `Random` ist.)

Am Anfang steht ein Exemplar der Klasse Random. Dieses Exemplar wird mit einem Zufallswert (Datentyp long) initialisiert, der dann für die weiteren Berechnungen verwendet wird. Dieser Startwert prägt die ganze Folge von erzeugten Zufallszahlen, obwohl nicht ersichtlich ist, wie sich die Folge verhält. Doch eines ist gewiss: Zwei mit gleichen Startwerten erzeugte Random-Objekte liefern auch dieselbe Folge von Zufallszahlen. Der parameterlose Standard-Konstruktor von Random initialisiert den Startwert mit der Summe aus einem magischen Startwert und System.nanoTime().

```
class java.util.Random
 implements Serializable
```

- Random()
 

Erzeugt einen neuen Zufallszahlengenerator.
- Random(long seed)
 

Erzeugt einen neuen Zufallszahlengenerator und benutzt den Parameter seed als Startwert.
- void setSeed(long seed)
 

Setzt den Seed neu. Der Generator verhält sich anschließend genauso wie ein mit diesem Seed-Wert frisch erzeugter Generator.

### 18.6.2 Zufallszahlen erzeugen

Die Random-Klasse erzeugt Zufallszahlen für vier verschiedene Datentypen: int (32 Bit), long (64 Bit), double und float. Dafür stehen vier Methoden zur Verfügung:

- int nextInt(), long nextLong()
 

Liefert die nächste Pseudo-Zufallszahl aus dem gesamten Wertebereich, also zwischen Integer.MIN\_VALUE und Integer.MAX\_VALUE beziehungsweise Long.MIN\_VALUE und Long.MAX\_VALUE.
- float nextFloat(), double nextDouble()
 

Liefert die nächste Pseudo-Zufallszahl zwischen 0,0 und 1,0.
- int nextInt(int range)
 

Liefert eine int-Pseudo-Zufallszahl im Bereich von 0 bis range.

Die Klasse Random verfügt über eine besondere Methode, mit der sich eine Reihe von Zufallszahlen erzeugen lässt. Dies ist die Methode nextBytes(byte[]). Der Parameter ist ein Byte-Feld, und dieses wird komplett mit Zufallszahlen gefüllt:

- void nextBytes(byte[] bytes)  
Füllt das Feld mit Zufallsbytes auf.

Hinter allen Methoden zur Erzeugung von Zufallszahlen steckt die Methode `next()`. Sie ist in `Random` implementiert, aber durch die Sichtbarkeit `protected` nur von einer erben-den Klasse sichtbar.

### 18.6.3 Pseudo-Zufallszahlen in der Normalverteilung \*

Über eine spezielle Methode können wir Zufallszahlen erhalten, die einer Normalverteilung genügen: `nextGaussian()`. Diese Methode arbeitet intern nach der sogenannten Polar-Methode und erzeugt aus zwei unabhängigen Pseudo-Zufallszahlen zwei normal-verteilte Zahlen. Der Mittelpunkt liegt bei 0, und die Standardabweichung ist 1. Die Werte, die `nextGaussian()` gibt, sind `double`-Zahlen und häufig in der Nähe von 0. Größere Zahlen sind der Wahrscheinlichkeit nach seltener.

```
class java.util.Random
 implements Serializable
```

- double `nextGaussian()`  
Liefert die nächste Zufallszahl in einer Gaußschen Normalverteilung mit der Mitte 0,0 und der Standardabweichung 1,0.

## 18.7 Große Zahlen \*

Die feste Länge der primitiven Datentypen `int`, `long` für Ganzzahlwerte und `float`, `double` für Fließkommawerte reicht für diverse numerische Berechnungen nicht aus. Besonders wünschenswert sind beliebig große Zahlen in der Kryptografie und präzise Auflösungen in der Finanzmathematik. Für solche Anwendungen gibt es im `math`-Paket zwei Klassen: `BigInteger` für Ganzzahlen und `BigDecimal` für Gleitkommazahlen.

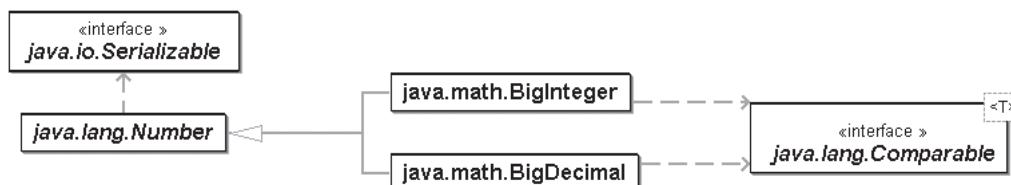


Abbildung 18.3: Vererbungsbeziehung von `BigInteger` und `BigDecimal`

### 18.7.1 Die Klasse BigInteger

Mit der Klasse `BigInteger` ist es uns möglich, beliebig genaue Zahlen anzulegen, zu verwalten und damit zu rechnen. Die `BigInteger`-Objekte werden dabei immer so lang, wie die entsprechenden Ergebnisse Platz benötigen (engl. *infinite word size*). Die Berechnungsmöglichkeiten gehen dabei weit über die der primitiven Typen hinaus und bieten des Weiteren viele statische Methoden der `Math`-Klasse. Zu den Erweiterungen gehören modulare Arithmetik, Bestimmung des größten gemeinsamen Teilers (ggT), Pseudo-Primzahltests, Bitmanipulation und Weiteres.

Die Implementierung stellt ein `BigInteger`-Objekt intern wie auch die primitiven Datentypen `byte`, `short`, `int`, `long` im Zweierkomplement da. Auch die weiteren Operationen entsprechen den Ganzzahl-Operationen der primitiven Datentypen, wie etwa die Division durch null, die eine `ArithmeticException` auslöst.

Intern vergrößert ein `BigInteger`, wenn nötig, den Wertebereich, sodass einige Operationen nicht übertragbar sind. So kann der Verschiebe-Operator `>>>` nicht übernommen werden, denn bei einer Rechtsverschiebung haben wir kein Vorzeichen-Bit im `BigInteger`. Da die Größe des Datentyps bei Bedarf immer ausgedehnt wird und durch diese interne Anpassung des internen Puffers kein Überlauf möglich ist, muss ein Anwender gegebenenfalls einen eigenen Überlauftest in sein Programm einbauen, wenn er den Wertebereich beschränken will.

Auch bei logischen Operatoren muss eine Interpretation der Werte vorgenommen werden. Bei Operationen auf zwei `BigInteger`-Objekten mit unterschiedlicher Bitlänge wird der kleinere Wert dem größeren durch Replikation (Wiederholung) des Vorzeichen-Bits angepasst. Über spezielle Bitoperatoren können einzelne Bits gesetzt werden. Wie bei der Klasse `BitSet` lassen sich durch die »unendliche« Größe Bits setzen, auch wenn die Zahl nicht so viele Bits benötigt. Durch die Bitoperationen lässt sich das Vorzeichen einer Zahl nicht verändern; gegebenenfalls wird vor der Zahl ein neues Vorzeichen-Bit mit dem ursprünglichen Wert ergänzt.

#### **BigInteger-Objekte erzeugen**

Zur Erzeugung stehen uns verschiedene Konstruktoren zur Verfügung. Einen Standard-Konstruktor gibt es nicht. Neben Konstruktoren, die das Objekt mit Werten aus einem Byte-Feld oder String initialisieren, lässt sich auch ein Objekt mit einer zufälligen Belegung erzeugen. Die Klasse `BigInteger` bedient sich dabei der Klasse `java.util.Random`. Ebenso lassen sich `BigInteger`-Objekte erzeugen, die Pseudo-Primzahlen sind.

```
class java.math.BigInteger
extends Number
implements Comparable<BigInteger>
```

- `BigInteger(String val)`

Erzeugt ein BigInteger aus einem Ziffern-String mit einem optionalen Vorzeichen.

- `BigInteger(String val, int radix)`

Ein String mit einem optionalen Vorzeichen wird zu einem BigInteger-Objekt übersetzt. Der Konstruktor verwendet die angegebene Basis radix, um die Zeichen des Strings als Ziffern zu interpretieren. Für radix > 10 werden die Buchstaben A-Z beziehungsweise a-z als zusätzliche »Ziffern« verwendet.

- `BigInteger(byte[] val)`

Ein Byte-Feld mit einer Zweierkomplement-Repräsentation einer BigInteger-Zahl im Big-Endian-Format (Array-Element mit Index 0, enthält die niederwertigsten Bits) initialisiert das neue BigInteger-Objekt.

- `BigInteger(int signum, byte[] magnitude)`

Erzeugt aus einem Big-Endian-Betrag beziehungsweise einer Vorzeichen-Repräsentation ein BigInteger-Objekt. signum gibt das Vorzeichen an und kann mit -1 (negative Zahlen), 0 (Null) und 1 (positive Zahlen) belegt werden.

- `BigInteger(int bitLength, int certainty, Random rnd)`

Erzeugt eine BigInteger-Zahl mit der Bitlänge bitLength (>1), bei der es sich mit gewisser Wahrscheinlichkeit um eine Primzahl handelt. Der Wert certainty bestimmt, wie wahrscheinlich ein Fehlurteil ist. Mit der Wahrscheinlichkeit  $1/(2^{certainty})$  handelt es sich bei der erzeugten Zahl fälschlicherweise doch um keine Primzahl. Je größer certainty (und je unwahrscheinlicher ein Fehlurteil) ist, desto mehr Zeit nimmt sich der Konstruktor.

- `BigInteger(int numbits, Random rnd)`

Liefert eine Zufallszahl aus dem Wertebereich 0 bis  $2^{\text{numBits}-1}$ . Alle Werte sind gleich wahrscheinlich.

- `static BigInteger valueOf(long val)`

Statische Fabrikmethode, die aus einem long ein BigInteger konstruiert.

Bei falschen Zeichenfolgen löst der Konstruktor mit String-Parameter eine NumberFormatException aus.

zB

**Beispiel**

Gegeben sei eine Zeichenkette, die eine Binärfolge aus Nullen und Einsen kodiert. Dann lässt sich ein Objekt der Klasse BigInteger nutzen, um diese Zeichenkette in ein Byte-Array zu konvertieren:

```
String s = "11011101 10101010 0010101 00010101".replace(" ", "");

byte[] bs = new BigInteger(s, 2).toByteArray(); // [158,261,69,69]

for (byte b : bs)
 System.out.println(Integer.toBinaryString(b & 0xFF));
```

Die Schleife erzeugt die vier Ausgaben 1101110, 11010101, 10101 und 10101.

Leider existiert noch immer kein Konstruktor, der auch den long-Datentyp annimmt. Seltsam – denn es gibt die statische Fabrikmethode valueOf(long), die BigInteger-Objekte erzeugt. Dies ist sehr verwirrend, da viele Programmierer diese Methoden übersehen und ein String-Objekt verwenden. Besonders ärgerlich ist es dann, einen privaten Konstruktor zu sehen, der mit einem long arbeitet. Genau diesen Konstruktor nutzt auch valueOf().

Neben den Konstruktoren und dem valueOf() gibt es drei Konstanten für die Werte 0, 1 und 10.

```
class java.math.BigInteger
extends Number
implements Comparable<BigInteger>
```

- static final BigInteger ZERO
- static final BigInteger ONE
- static final BigInteger TEN

18

### 18.7.2 Methoden von BigInteger

Die erste Kategorie von Methoden bildet arithmetische Operationen nach, für die es sonst ein Operatorzeichen oder eine Methode aus Math gäbe.

```
class java.math.BigInteger
extends Number
implements Comparable<BigInteger>
```

- `BigInteger abs()`  
Liefert den Absolutwert, ähnlich wie `Math.abs()` für primitive Datentypen.
- `BigInteger add(BigInteger val)`
- `BigInteger and(BigInteger val)`
- `BigInteger andNot(BigInteger val)`
- `BigInteger divide(BigInteger val)`
- `BigInteger mod(BigInteger m)`
- `BigInteger multiply(BigInteger val)`
- `BigInteger or(BigInteger val)`
- `BigInteger remainder(BigInteger val)`
- `BigInteger subtract(BigInteger val)`
- `BigInteger xor(BigInteger val)`  
Bildet ein neues BigInteger-Objekt mit der Summe, Und-Verknüpfung, Und-Nicht-Verknüpfung, Division, dem Modulo, Produkt, Oder, Restwert, der Differenz, dem Xor dieses Objekts und des anderen.
- `BigInteger[] divideAndRemainder(BigInteger val)`  
Liefert ein Feld mit zwei BigInteger-Objekten. Im Feld, dem Rückgabeobjekt, steht an der Stelle 0 der Wert für `this / val`, und an der Stelle 1 folgt `this % val`.
- `BigInteger modInverse(BigInteger m)`  
Bildet ein neues BigInteger, indem es vom aktuellen BigInteger 1 subtrahiert und es dann Modulo m nimmt.
- `BigInteger modPow(BigInteger exponent, BigInteger m)`  
Nimmt den aktuellen BigInteger hoch exponent Modulo m.
- `BigInteger negate()`  
Negiert das Objekt, liefert also ein neues BigInteger mit umgekehrtem Vorzeichen.
- `BigInteger not()`  
Liefert ein neues BigInteger, das die Bits negiert hat.
- `BigInteger pow(int exponent)`  
Bildet `this` hoch exponent.
- `int signum()`  
Liefert das Vorzeichen des eigenen BigInteger-Objekts.

Die nächste Kategorie von Methoden ist eng mit den Bits der Zahl verbunden:

- `int bitCount()`  
Zählt die Anzahl gesetzter Bits der Zahl, die im Zweierkomplement vorliegt.
- `int bitLength()`  
Liefert die Anzahl der Bits, die nötig sind, um die Zahl im Zweierkomplement ohne Vorzeichen-Bit darzustellen.
- `BigInteger clearBit(int n)`
- `BigInteger flipBit(int n)`
- `BigInteger setBit(int n)`  
Liefert ein neues BigInteger-Objekt mit gelöschtem/gekipptem/gesetztem *n*-tem Bit.
- `BigInteger shiftLeft(int n)`
- `BigInteger shiftRight(int n)`  
Schiebt die Bits um *n* Stellen nach links/rechts.
- `int getLowestSetBit()`  
Liefert die Position eines Bits, das in der Repräsentation der Zahl am weitesten rechts gesetzt ist.
- `boolean testBit(int n)`  
true, wenn das Bit *n* gesetzt ist.

Folgende Methoden sind besonders für kryptografische Verfahren interessant:

- `BigInteger gcd(BigInteger val)`  
Liefert den größten gemeinsamen Teiler vom aktuellen Objekt und *val*.
- `boolean isProbablePrime(int certainty)`  
Ist das BigInteger-Objekt mit der Wahrscheinlichkeit *certainty* eine Primzahl?
- `BigInteger nextProbablePrime()`  
Liefert die nächste Ganzzahl hinter dem aktuellen BigInteger, die wahrscheinlich eine Primzahl ist.
- `static BigInteger probablePrime(int bitLength, Random rnd)`  
Liefert mit einer bestimmten Wahrscheinlichkeit eine Primzahl der Länge *bitLength*.

Die letzte Gruppe bilden die Vergleichs- und Konvertierungsmethoden:

- `int compareTo(Object o)`
- `int compareTo(BigInteger o)`  
Da die Klasse BigInteger die Schnittstelle java.lang.Comparable implementiert, lässt sich jedes BigInteger-Objekt mit einem anderen vergleichen. Die Methode mit dem

Datentyp `BigInteger` ist natürlich nicht von `Comparable` vorgeschrieben, aber beide Methoden sind identisch.

- `double doubleValue()`
- `float floatValue()`
- `int intValue()`
- `long longValue()`  
Konvertiert den `BigInteger` in ein `double/float/int/long`. Es handelt sich um implementierte Methoden der abstrakten Oberklasse `Number`.
- `boolean equals(Object x)`  
Vergleicht, ob `x` und das eigene `BigInteger`-Objekt den gleichen Wert annehmen.
- `BigInteger max(BigInteger val)`
- `BigInteger min(BigInteger val)`  
Liefert das größere/kleinere der `BigInteger`-Objekte als Rückgabe.
- `byte[] toByteArray()`  
Liefert ein Byte-Feld mit dem `BigInteger` als Zweierkomplement.
- `String toString()`
- `String toString(int radix)`  
Liefert die String-Repräsentation von diesem `BigInteger` zur Basis 10 beziehungsweise einer beliebigen Basis.
- `static BigInteger valueOf(long val)`  
Erzeugt ein `BigInteger`, das den Wert `val` annimmt.

### 18.7.3 Ganz lange Fakultäten

Unser Beispielprogramm soll die Fakultät einer natürlichen Zahl berechnen. Die Zahl muss positiv sein:

**Listing 18.11:** Factorial.java

```
import java.math.*;

class Factorial
{
 static BigInteger factorial(int n)
 {
 BigInteger result = BigInteger.ONE;
```

```
if (n == 0 || n == 1)
 return result;

if (n > 1)
 for (int i = 1; i <= n; i++)
 result = result.multiply(BigInteger.valueOf(i));

return result;
}

static public void main(String[] args)
{
 System.out.println(factorial(100));
}
}
```

Neben dieser iterativen Variante ist eine rekursive denkbar. Sie ist allerdings aus zwei Gründen nicht wirklich gut. Zuerst aufgrund des hohen Speicherplatzbedarfs: Für die Berechnung von  $n!$  sind  $n$  Objekte nötig. Im Gegensatz zur iterativen Variante müssen jedoch alle Zwischenobjekte bis zum Auflösen der Rekursion im Speicher gehalten werden. Dadurch ergibt sich die zweite Schwäche: die längere Laufzeit. Aus akademischen Gründen soll dieser Weg hier allerdings aufgeführt werden. Es ist interessant zu beobachten, wie diese rekursive Implementierung den Speicher aufzehrt. Dabei ist es nicht einmal der Heap, der keine neuen Objekte mehr aufnehmen kann, sondern vielmehr der Stack des aktuellen Threads:

**Listing 18.12:** Factorial.java, factorial2()

```
public static BigInteger factorial2(int i)
{
 if (i <= 1)
 return BigInteger.ONE;

 return BigInteger.valueOf(i).multiply(factorial2(i - 1));
}
```

#### 18.7.4 Große Fließkommazahlen mit BigDecimal

Während sich BigInteger um die beliebig genauen Ganzzahlen kümmert, übernimmt BigDecimal die Fließkommazahlen.

## BigDecimal aufbauen

Der Konstruktor nimmt unterschiedliche Typen an, unter anderem double und String.  
Bei double ist Obacht geboten, denn während

das Literal auf den für double gültigen Bereich bringt (1), ist Folgendes präzise:

Das gleiche Phänomen ist bei System.out.println(new BigDecimal(Math.PI)); zu beobachten; die Ausgabe suggeriert eine hohe Genauigkeit:

3.141592653589793115997963468544185161590576171875

Richtig ist jedoch:

3.141592653589793~~238462643383279502884197169399375~~

## Methoden statt Operatoren

Mit den `BigDecimal`-Objekten lässt sich nun rechnen, wie von `BigInteger` bekannt. Die wichtigsten Methoden sind:

```
class java.math.BigDecimal
extends Number
implements Comparable<BigDecimal>
```

- `BigDecimal add(BigDecimal augend)`
  - `BigDecimal subtract(BigDecimal subtrahend)`
  - `BigDecimal divide(BigDecimal divisor)`
  - `BigDecimal multiply(BigDecimal multiplicand)`
  - `BigDecimal remainder(BigDecimal divisor)`
  - `BigDecimal abs()`

- BigDecimal negate()
- BigDecimal plus()
- BigDecimal max(BigDecimal val)
- BigDecimal min(BigDecimal val)
- BigDecimal pow(int n)

Des Weiteren gibt es drei Konstanten für die Zahlen `BigDecimal.ZERO`, `BigDecimal.ONE` und `BigDecimal.TEN`.

## Rundungsmodus

Eine Besonderheit stellt jedoch die Methode `divide()` dar, die zusätzlich einen Rundungsmodus und optional auch eine Anzahl gültiger Nachkommastellen bekommen kann.

```
BigDecimal a = new BigDecimal("10");
BigDecimal b = new BigDecimal("2");
System.out.println(a.divide(b)); // 5
```

Es ist kein Problem, wenn das Ergebnis eine Ganzzahl oder das Ergebnis exakt ist.

```
System.out.println(new BigDecimal(1).divide(b)); // 0.5
```

Wenn das Ergebnis aber nicht exakt ist, lässt sich `divide()` nicht einsetzen. Die Anweisung

```
new BigDecimal(1).divide(new BigDecimal(3))
```

ergibt den Fehler:

```
java.lang.ArithmaticException: Non-
terminating decimal expansion; no exact representable decimal result.
```

An dieser Stelle kommen die Rundungsmodi `ROUND_UP`, `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_FLOOR`, `ROUND_HALF_UP`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN` ins Spiel. `ROUND_UNNECESSARY` ist auch einer davon, darf aber nur dann verwendet werden, wenn die Division exakt ist:

```
System.out.println(c.divide(d, BigDecimal.ROUND_UP)); // 1
System.out.println(c.divide(d, BigDecimal.ROUND_DOWN)); // 0
```

Jetzt kann noch die Anzahl der Nachkommastellen bestimmt werden:

```
System.out.println(c.divide(d, 6, BigDecimal.ROUND_UP)); // 0.333334
System.out.println(c.divide(d, 6, BigDecimal.ROUND_DOWN)); // 0.333333
```

zB

### Beispiel

`BigDecimal` bietet die praktische Methode `setScale()` an, mit der sich die Anzahl der Nachkommastellen setzen lässt. Das ist zum Runden sehr gut. In unserem Beispiel sollen 45 Liter Benzin zu 1,399 bezahlt werden:

**Listing 18.13:** RoundWithSetScale.java, main()

```
BigDecimal petrol = new BigDecimal("1.399").multiply(new BigDecimal(45));
System.out.println(petrol.setScale(3, BigDecimal.ROUND_HALF_UP));
System.out.println(petrol.setScale(2, BigDecimal.ROUND_HALF_UP));
```

Die Ausgaben sind 62.955 und 62.96.

### 18.7.5 Mit MathContext komfortabel die Rechengenauigkeit setzen

Die Klasse `java.math.MathContext` wurde in Java 5 eingeführt, um für `BigDecimal` komfortabel die Rechengenauigkeit (nicht die Nachkommastellen) und den Rundungsmodus setzen zu können. Vorher wurde diese Information, wie das vorangehende Beispiel gezeigt hat, den einzelnen Berechnungsmethoden mitgegeben. Jetzt kann dieses eine Objekt einfach an alle berechnenden Methoden weitergegeben werden.

Die Eigenschaften werden mit den Konstruktoren gesetzt, denn `MathContext`-Objekte sind anschließend immutable.

```
class java.math.MathContext
implements Serializable
```

- `MathContext(int setPrecision)`  
Baut ein neues `MathContext` mit angegebener Präzision als Rundungsmodus `HALF_UP`.
- `MathContext(int setPrecision, RoundingMode setRoundingMode)`  
Baut ein neues `MathContext` mit angegebener Präzision und einem vorgegebenen Rundungsmodus vom Typ `RoundingMode`. Deklarierte Konstanten der Aufzählung sind `CEILING`, `DOWN`, `FLOOR`, `HALF_DOWN`, `HALF_EVEN`, `HALF_UP`, `UNNECESSARY` und `UP`.

- `MathContext(String val)`

Baut ein neues `MathContext` aus einem String. Der Aufbau des Strings ist wie von `toString()` der Klasse, etwa `precision=34 roundingMode=HALF_EVEN`.

Für die üblichen Fälle stehen vier vorgefertigte `MathContext`-Objekte als Konstanten der Klasse zur Verfügung: `DECIMAL128`, `DECIMAL32`, `DECIMAL64` und `UNLIMITED`.

**Listing 18.14:** `MathContextDemo.java`, `main()`

```
out.println(MathContext.DECIMAL128); // precision=34 roundingMode=HALF_EVEN
```

Nach dem Aufbau des `MathContext`-Objekts wird es im Konstruktor von `BigDecimal` übergeben.

```
class java.math.BigDecimal
 extends Number
 implements Comparable<BigInteger>
```

- `BigDecimal(BigInteger unscaledVal, int scale, MathContext mc)`
- `BigDecimal(BigInteger val, MathContext mc)`
- `BigDecimal(char[] in, int offset, int len, MathContext mc)`
- `BigDecimal(char[] in, MathContext mc)`
- `BigDecimal(double val, MathContext mc)`
- `BigDecimal(int val, MathContext mc)`
- `BigDecimal(long val, MathContext mc)`
- `BigDecimal(String val, MathContext mc)`

Auch bei jeder Berechnungsmethode lässt sich nun das `MathContext`-Objekt übergeben:

- `BigDecimal abs(MathContext mc)`
- `BigDecimal add(BigDecimal augend, MathContext mc)`
- `BigDecimal divide(BigDecimal divisor, MathContext mc)`
- `BigDecimal divideToIntegralValue(BigDecimal divisor, MathContext mc)`
- `BigDecimal plus(MathContext mc)`
- `BigDecimal pow(int n, MathContext mc)`
- `BigDecimal remainder(BigDecimal divisor, MathContext mc)`
- `BigDecimal round(MathContext mc)`
- `BigDecimal subtract(BigDecimal subtrahend, MathContext mc)`

## 18.8 Zum Weiterlesen

Die Java-Bibliothek bietet, abgesehen von den Klassen zur Unterstützung großer Wertebereiche, kaum weitere Algorithmen, wie sie oft für mathematische Probleme benötigt werden. Zu den wenigen Methoden gehören `solveCubic()` und `solveQuadratic()` aus den Klassen `CubicCurve2D` und `QuadCurve2D`, und selbst die sind nicht fehlerfrei.<sup>8</sup>

Auf dem (freien) Markt gibt es aber eine große Anzahl an Erweiterungen, etwa für Brüche, Polynome, Matrizen und so weiter. Eine kleine Auswahl:

- *Commons-Math: Jakarta Mathematics Library* (<http://jakarta.apache.org/commons/math/>). Enthält unter anderem Statistik, lineare Algebra, komplexe Zahlen, Brüche.
- *JScience* (<http://jscience.org/>). Enthält lineare Algebra mit Matrizen und Vektoren sowie LU-Zerlegung, Brüche, Polynome.
- *JAMA: A Java Matrix Package* (<http://math.nist.gov/javanumerics/jama/>). Bietet Eigenwerte berechnen, Lösen nichtsingulärer Systeme, Determinante ...

Eine Liste weiterer Bibliotheken bietet <http://math.nist.gov/javanumerics/>.

Ein weiteres Problemfeld sind die Rechenungenuigkeiten, die jedoch einfach in der Natur der Sache liegen. Doch auch andere Probleme bestehen, die das Paper »How Java's Floating-Point Hurts Everyone Everywhere« (<http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf>) sehr genau behandelt – allerdings wurde es vor der Einführung des Schlüsselworts `strict` veröffentlicht, sodass nicht mehr jeder Punkt gültig ist.

---

<sup>8</sup> [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=4645692](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4645692)



## Kapitel 19

# Die Werkzeuge des JDK

»Erfolg sollte stets nur die Folge, nie das Ziel des Handelns sein.«  
– Gustave Flaubert (1821–1880)

Dieses Kapitel stellt die wichtigsten Programme des JDK vor. Da die meisten Programme kommandozeilenorientiert arbeiten, werden sie zusammen mit ihrer Aufrufsyntax vorgestellt. Bei den JDK-Programmen handelt es sich unter anderem um folgende Tools:

- **javac**: Java-Compiler zum Übersetzen von `.java` in `.class`-Dateien
- **java**: Java-Interpreter zum Ausführen der Java-Applikationen
- **appletviewer**: Applet-Viewer zum Ausführen von Java-Applets, die in eine HTML-Datei eingebettet sind
- **javadoc**: Generator für Header- und Quellcode-Rümpfe zum nativen Zugriff
- **javap**: Anzeiger vom Bytecode einer Klassendatei
- **jdb**: Debugger zum Durchlaufen eines Programms
- **jar**: Archivierungswerkzeug, um Dateien in einem Archiv zusammenzufassen
- **jconsole**: Java-Monitoring- und Management-Konsole
- **pack200, unpack200**: Starke (De-)Kompression von Jar-Dateien
- **serialver**: Generiert serialVersionUID.
- **keytool, jarsigner und policytool**: Programme zum Einstellen der Sicherheitseigenschaften

Obwohl es versionsabhängig noch weitere Aufrufparameter gibt, sind nur diejenigen aufgeführt, die offiziell in der aktuellen Dokumentation genannt sind.

## 19.1 Java-Quellen übersetzen

### 19.1.1 Java-Compiler vom JDK

Der Compiler *javac* übersetzt den Quellcode einer Datei in Java-Bytecode. Jede in einer Datei deklarierte Klasse übersetzt der Compiler in eine eigene Klassendatei. Wenn bei einer Klasse (nennen wir sie A) eine Abhängigkeit zu einer anderen Klasse (nennen wir sie B) besteht – wenn zum Beispiel A von B erbt – und B nicht als Bytecode-Datei vorliegt, dann verarbeitet der Compiler B automatisch mit. Der Compiler überwacht also automatisch die Abhängigkeiten der Quelldateien. Der allgemeine Aufruf des Compilers ist:

```
$ javac [Optionen] Dateiname(n).java
```

| Option                             | Bedeutung                                                                                                                                                                                                                                                                        |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-cp classpath</code>         | Eine Liste von Pfaden, auf denen der Compiler die Klassendateien finden kann. Diese Option überschreibt die unter Umständen gesetzte Umgebungsvariable CLASSPATH und ergänzt sie nicht. Ein Semikolon (Windows) beziehungsweise Doppelpunkt (Unix) trennt mehrere Verzeichnisse. |
| <code>-d Verzeichnis</code>        | Gibt an, wo die übersetzten <code>.class</code> -Dateien gespeichert werden. Ohne Angabe legt der Compiler sie in das gleiche Verzeichnis wie das mit den Quelldateien.                                                                                                          |
| <code>-deprecation</code>          | Zeigt veraltete Methoden an.                                                                                                                                                                                                                                                     |
| <code>-g</code>                    | Erzeugt Debug-Informationen. Die Option muss gesetzt sein, damit der Debugger alle Informationen hat. <code>-g:none</code> erzeugt keine Debug-Informationen, was die Klassendatei etwas kleiner macht.                                                                          |
| <code>-nowarn</code>               | Deaktiviert die Ausgabe von Warnungen. Fehler ( <i>errors</i> ) werden noch angezeigt.                                                                                                                                                                                           |
| <code>-source Version</code>       | Erzeugt Bytecode für eine bestimmte Java-Version.                                                                                                                                                                                                                                |
| <code>-sourcepath Quellpfad</code> | Ähnlich wie <code>-classpath</code> , nur sucht der Compiler im Quellpfad nach Quelldateien.                                                                                                                                                                                     |
| <code>-verbose</code>              | Ausgabe von Meldungen über geladene Quell- und Klassendateien während der Übersetzung.                                                                                                                                                                                           |

Tabelle 19.1: Optionen des Compilers *javac*

### 19.1.2 Native Compiler

Eine in Java geschriebene Applikation lässt sich erst einmal nur mit einer Java-Laufzeitumgebung ausführen. Einige Hersteller haben jedoch Compiler entwickelt, die direkt unter Windows oder einem anderen Betriebssystem ausführbare Programme erstellen. Die Compiler, die aus Java-Quelltext – oder Java-Bytecode – Maschinencode der jeweiligen Architektur erzeugen, nennen sich *native* oder *Ahead-of-Time Compiler*. Das Ergebnis ist eine direkt ausführbare Datei, die keine Java-Laufzeitumgebung nötig macht. Je nach Anwendungsfall kann das Programm performanter sein, eine Garantie dafür gibt es allerdings nicht. Die Startzeiten sind im Allgemeinen geringer, und das Programm ist viel schwieriger zu entschlüsseln, was das Reverse Engineering<sup>1</sup> angeht.

Ein freier Compiler unter der GNU-Lizenz ist *gcj* (<http://gcc.gnu.org/java/>). Für den gcj integriert das (allerdings nicht mehr so aktive) Open-Source-Projekt *JavaCompiler* (<http://jnc.mtsystems.ch/>) diverse Zusätze, um natives Übersetzen zu vereinfachen.

Ein kommerzieller Vertreter ist *Excelsior JET* (<http://www.excelsior-usa.com/jet.html>). Dass viele Hersteller ihre Produkte eingestellt haben, ist sicherlich ein Zeichen dafür, dass die existierenden Laufzeitumgebungen mittlerweile eine ausreichende Geschwindigkeit, einen vertretbaren Speicherverbrauch und annehmbare Startzeiten zeigen.

### 19.1.3 Java-Programme in ein natives ausführbares Programm einpacken

Wer Java-Programme vertreibt, weiß um das Problem der JVM-Versionen, Pfade, Start-Icons, Splash-Screens und so weiter Bescheid. Eine Lösung besteht darin, einen Wrapper zu bemühen, der sich als ausführbares Programm wie eine Schale um das Java-Programm legt. Der Wrapper ruft die virtuelle Maschine auf und über gibt ihr die Klassen. Es ist also immer noch eine Laufzeitumgebung nötig, doch lassen sich den Java-Programmen Icons und Startparameter setzen.

Die Open-Source-Software *launch4j* (<http://launch4j.sourceforge.net/>) kapselt ein Java-Archiv mit Klassen und Ressource-Dateien in ein komprimiertes, ausführbares Programm für Windows, Linux, Mac OS X und Solaris. *launch4j* setzt Eigenschaften wie ein assoziiertes Icon oder Startvariablen mit einer angenehmen grafischen Oberfläche. Ein weiteres quelloffenes und freies Programm ist *JSmooth* (<http://jsmooth.sourceforge.net/>). Für beide gibt es Ant-Skripte.

---

<sup>1</sup> Das Zurückverwandeln von unstrukturiertem Binär code in Quellcode.

## 19.2 Die Java-Laufzeitumgebung

Der Java-Interpreter `java` führt den Java-Bytecode in der Laufzeitumgebung aus. Dazu sucht der Interpreter in der als Parameter übergebenen Klassendatei nach der speziellen statischen `main()`-Methode. Der allgemeine Aufruf ist:

```
$ java [Optionen] Klassenname [Argumente]
```

Ist die Klasse in einem Paket deklariert, muss der Name der Klasse voll qualifiziert sein. Liegt die Klasse Main etwa im Paket `com.tutego`, also im Unterverzeichnis `com/tutego`, muss der Klassenname `com.tutego.Main` lauten. Die benötigten Klassen muss die Laufzeitumgebung finden können. Die JVM wertet wie der Compiler die Umgebungsvariable `CLASSPATH` aus und erlaubt die Angabe des Klassenpfades durch die Option `-classpath`.

| Option                                  | Bedeutung                                                                                                                                                                                                                                                                                                |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-client</code>                    | Wählt die <i>Java HotSpot Client VM</i> , Standard.                                                                                                                                                                                                                                                      |
| <code>-server</code>                    | Wählt die <i>Java HotSpot Server VM</i> .                                                                                                                                                                                                                                                                |
| <code>-cp classpath</code>              | Eine Liste von Pfaden, innerhalb derer der Compiler die Klassendateien finden kann. Diese Option überschreibt die unter Umständen gesetzte Umgebungsvariable <code>CLASSPATH</code> und ergänzt sie nicht. Das Semikolon (Windows) beziehungsweise der Doppelpunkt (Unix) trennen mehrere Verzeichnisse. |
| <code>-D Property=Wert</code>           | Setzt den Wert einer Property, etwa <code>-Dversion=1.2</code> , die später <code>System.getProperty()</code> erfragen kann.                                                                                                                                                                             |
| <code>-help</code> oder <code>-?</code> | Listet alle vorhandenen Optionen auf.                                                                                                                                                                                                                                                                    |
| <code>-ea</code>                        | Ermöglicht Assertions, die standardmäßig ausgeschaltet sind.                                                                                                                                                                                                                                             |
| <code>-jar</code>                       | Startet eine Klasse aus dem Jar-Archiv, falls sie in der Manifest-Datei genannt ist. Die Hauptklasse lässt sich aber immer noch angeben.                                                                                                                                                                 |
| <code>-verbose</code>                   | Informationen über die Laufzeitumgebung: <ul style="list-style-type: none"> <li>■ <code>-verbose:class</code> gibt Informationen über geladene Klassen.</li> <li>■ <code>-verbose:gc</code> informiert über GC-Aufrufe.</li> <li>■ <code>-verbose:jni</code> informiert über native Aufrufe.</li> </ul>  |
| <code>-version</code>                   | Zeigt die aktuelle Version an.                                                                                                                                                                                                                                                                           |
| <code>-X</code>                         | Zeigt nicht standardisierte Optionen an.                                                                                                                                                                                                                                                                 |
| <code>-Xdebug</code>                    | Startet mit Debugger.                                                                                                                                                                                                                                                                                    |

Tabelle 19.2: Optionen des Interpreters `java`

| Option                   | Bedeutung                                                                                                                                                                                                                                        |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-Xincgc</code>     | Schaltet den inkrementellen GC ein.                                                                                                                                                                                                              |
| <code>-Xmsn</code>       | Anfangsgröße des Speicherbereichs für die Allokation von Objekten (n MiB), voreingestellt sind 2 MiB.                                                                                                                                            |
| <code>-Xmxn</code>       | Maximal verfügbarer Speicherbereich für die Allokation von Objekten. Voreingestellt sind 64 MiB. n beschreibt als einfache Zahl die Bytes oder Kilobytes mit einem angefügten k oder Megabytes (angefügtes m). Beispiel: <code>-Xms128m</code> . |
| <code>-Xnoclassgc</code> | Schaltet den GC für geladene, aber nicht mehr benötigte Klassen aus.                                                                                                                                                                             |
| <code>-Xprof</code>      | Der Interpreter schreibt Profiling-Informationen in der Datei <code>java.prof</code> .                                                                                                                                                           |
| <code>-Xrs</code>        | Reduziert intern die Verwendung von Unix-Signalen durch die Laufzeitumgebung. Das ergibt gegebenenfalls eine schlechtere Performance, aber eine bessere Kompatibilität mit diversen Unix-/Solaris-Versionen.                                     |
| <code>-Xssn</code>       | Setzt die Größe des Stacks.                                                                                                                                                                                                                      |

Tabelle 19.2: Optionen des Interpreters java (Forts.)

### Zusatzoptionen

Mit der Option `-X` lassen sich weitere Schalter setzen und dann der Laufzeitumgebung Zusatzanweisungen geben, etwas über den maximal zu verwendenden Speicher. Ein interessanter Schalter ab Java 7 ist `-XshowSettings`, der die Zustände der Standardeigenschaften ausgibt. Das ist sehr nützlich, um etwa abzulesen, welche Pfade gesetzt sind. Angewendet auf das Quadrat-Programm aus dem Kapitel 1 ergibt sich dann:

```
$ java -XshowSettings Quadrat
VM settings:
 Max. Heap Size (Estimated): 247.50M
 Ergonomics Machine Class: client
 Using VM: Java HotSpot(TM) Client VM
```

Property settings:

```
awt.toolkit = sun.awt.windows.WToolkit
file.encoding = Cp1252
file.encoding.pkg = sun.io
file.separator = \
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
```

```
java.awt.printerjob = sun.awt.windows.WPrinterJob
java.class.path = .
 C:\Program Files\Java\jre7\lib\ext\QTJava.zip
java.class.version = 51.0
java.endorsed.dirs = C:\Program Files\Java\jre7\lib\endorsed
java.ext.dirs = C:\Program Files\Java\jre7\lib\ext
 C:\windows\Sun\Java\lib\ext
java.home = C:\Program Files\Java\jre7
java.io.tmpdir = C:\Users\CHRIST~1\AppData\Local\Temp\
java.library.path = C:\windows\SYSTEM32
.
 C:\windows\Sun\Java\bin
 C:\windows\system32
 C:\windows
 C:\windows\SYSTEM32
 C:\windows
 C:\windows\SYSTEM32\WBEM
 C:\windows\SYSTEM32\WINDOWSPOWERSHELL\V1.0\
 C:\Program Files\QuickTime\QTSystem\
java.runtime.name = Java(TM) SE Runtime Environment
java.runtime.version = 1.7.0-ea-b143
java.specification.name = Java Platform API Specification
java.specification.vendor = Oracle Corporation
java.specification.version = 1.7
java.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
java.vendor.url.bug = http://bugreport.sun.com/bugreport/
java.version = 1.7.0-ea
java.vm.info = mixed mode, sharing
java.vm.name = Java HotSpot(TM) Client VM
java.vm.specification.name = Java Virtual Machine Specification
java.vm.specification.vendor = Oracle Corporation
java.vm.specification.version = 1.7
java.vm.vendor = Oracle Corporation
java.vm.version = 21.0-b13
line.separator = \r \n
```

```
os.arch = x86
os.name = Windows 7
os.version = 6.1
path.separator = ;
sun.arch.data.model = 32
sun.boot.class.path = C:\Program Files\Java\jre7\lib\resources.jar
C:\Program Files\Java\jre7\lib\rt.jar
C:\Program Files\Java\jre7\lib\sunrsasign.jar
C:\Program Files\Java\jre7\lib\jsse.jar
C:\Program Files\Java\jre7\lib\jce.jar
C:\Program Files\Java\jre7\lib\charsets.jar
C:\Program Files\Java\jre7\classes
sun.boot.library.path = C:\Program Files\Java\jre7\bin
sun.cpu.endian = little
sun.cpu.isalist = pentium_pro+mmx pentium_pro pentium+mmx pentium i486 i386 i86
sun.desktop = windows
sun.io.unicode.encoding = UnicodeLittle
sun.java.command = Quadrat
sun.java.launcher = SUN_STANDARD
sun.jnu.encoding = Cp1252
sun.management.compiler = HotSpot Client Compiler
sun.os.patch.level = Service Pack 1
user.country = DE
user.dir = C:\Users\Christian\Documents\My Dropbox\Insel\programme\1_01_Intro
user.home = C:\Users\Christian
user.language = de
user.name = Christian
user.script =
user.timezone =
user.variant =
```

#### Locale settings:

```
default locale = Deutsch
default display locale = Deutsch (Deutschland)
default format locale = Deutsch (Deutschland)
available locales = ar, ar_AE, ar_BH, ar_DZ, ar_EG, ar_IQ, ar_JO, ar_KW,
```

```

ar_LB, ar LY, ar MA, ar OM, ar QA, ar SA, ar SD, ar SY,
ar TN, ar YE, be, be BY, bg, bg BG, ca, ca ES,
cs, cs CZ, da, da DK, de, de AT, de CH, de DE,
de LU, el, el CY, el GR, en, en AU, en CA, en GB,
en IE, en IN, en MT, en NZ, en PH, en SG, en US, en ZA,
es, es AR, es BO, es CL, es CO, es CR, es DO, es EC,
es ES, es GT, es HN, es MX, es NI, es PA, es PE, es PR,
es PY, es SV, es US, es UY, es VE, et, et EE, fi,
fi FI, fr, fr BE, fr CA, fr CH, fr FR, fr LU, ga,
ga IE, hi IN, hr, hr HR, hu, hu HU, in, in ID,
is, is IS, it, it CH, it IT, iw, iw IL, ja,
ja JP, ja JP JP #u-ca-japanese, ko, ko KR, lt, lt LT, lv, lv LV,
mk, mk MK, ms, ms MY, mt, mt MT, nl, nl BE,
nl NL, no, no NO, no NO NY, pl, pl PL, pt, pt BR,
pt PT, ro, ro RO, ru, ru RU, sk, sk SK, sl,
sl SI, sq, sq AL, sr, sr BA, sr BA #Latn, sr CS, sr ME,
sr ME #Latn, sr RS, sr RS #Latn, sr #Latn, sv, sv SE, th, th TH,
th TH TH #u-nu-thai, tr, tr TR, uk, uk UA, vi, vi VN, zh,
zh CN, zh HK, zh SG, zh TW

Quadrat(1) = 1
Quadrat(2) = 4
Quadrat(3) = 9
Quadrat(4) = 16

```

### Class-Path-Wildcard

Die Option `-cp` erweitert den Klassenpfad durch Java-Archive (`.jar`-Dateien) und einzelne Klassen-Dateien (`.class`-Dateien). Seit Java 6 ermöglicht eine Class-Path-Wildcard über `*` eine noch einfachere Angabe von Java-Archiven. So fügt folgende Angabe alle Java-Archive im Verzeichnis `lib` dem Klassenpfad hinzu:

```
$ java -cp lib/* Main
```



#### Hinweis

Je länger es die JVM von Oracle gibt, desto länger wurde die Liste der Optionen. <http://blogs.sun.com/watt/resource/jvm-options-list.html> listet diese je nach Version kurz auf.

### Der Unterschied zwischen `java.exe` und `javaw.exe`

Unter einer Windows-Installation gibt es im Java-JDK für den Interpreter zwei ausführbare Dateien: `java.exe` und `javaw.exe` – `java.exe` stellt die Regel dar. Der Unterschied besteht darin, dass eine über die grafische Oberfläche gestartete Applikation mit `java.exe` im Unterschied zu `javaw.exe` ein Konsolenfenster anzeigt. Ohne Konsolenfenster sind mit `javaw` dann auch Ausgaben über `System.out/err` nicht sichtbar.

In der Regel nutzt ein Programm mit grafischer Oberfläche während der Entwicklung `java` und im Produktivbetrieb dann `javaw`.

## 19.3 Dokumentationskommentare mit JavaDoc

Die Dokumentation von Softwaresystemen ist ein wichtiger, aber oft vernachlässigter Teil der Softwareentwicklung. Leider, denn Software wird im Allgemeinen öfter gelesen als geschrieben. Während des Entwicklungsprozesses müssen die Entwickler Zeit in Beschreibungen der einzelnen Komponenten investieren, besonders dann, wenn weitere Entwickler diese Komponenten in einer öffentlichen Bibliothek anderen Entwicklern zur Wiederverwendung zur Verfügung stellen. Um die Klassen, Schnittstellen, Aufzählungen und Methoden sowie Attribute gut zu finden, müssen sie sorgfältig beschrieben werden. Wichtig bei der Beschreibung sind der Typname, der Methodename, die Art und die Anzahl der Parameter, die Wirkung der Methoden und das Laufzeitverhalten. Da das Erstellen einer externen Dokumentation (also einer Beschreibung außerhalb der Quellcodedatei) fehlerträchtig und deshalb nicht gerade motivierend für die Beschreibung ist, werden spezielle Dokumentationskommentare in den Java-Quelltext eingeführt. Ein spezielles Programm generiert aus den Kommentaren Beschreibungsdateien (im Allgemeinen HTML) mit den gewünschten Informationen.<sup>2</sup>

### 19.3.1 Einen Dokumentationskommentar setzen

In einer besonders ausgezeichneten Kommentarumgebung werden die *Dokumentationskommentare* (»*Doc Comments*«) eingesetzt. Die Kommentarumgebung erweitert einen Blockkommentar und ist vor allen Typen (Klassen, Schnittstellen, Aufzählungen)

---

<sup>2</sup> Die Idee ist nicht neu. In den 1980er-Jahren verwendete Donald E. Knuth das WEB-System zur Dokumentation von TeX. Das Programm wurde mit den Hilfsprogrammen *weave* und *tangle* in ein Pascal-Programm und eine TeX-Datei umgewandelt.

sowie Methoden und Variablen üblich. Im folgenden Beispiel gibt JavaDoc Kommentare für die Klasse, Attribute und Methoden an:

**Listing 19.1:** com/tutego/insel/javadoc/Room.java

```
package com.tutego.insel.javadoc;
```

```
/**
 * This class models a room with a given number of players.
 */
public class Room
{
 /** Number of players in a room. */
 private int numberOfPersons;

 /**
 * A person enters the room.
 * Increments the number of persons.
 */
 public void enterPerson() {
 numberOfPersons++;
 }

 /**
 * A person leaves the room.
 * Decrement the number of persons.
 */
 public void leavePerson() {
 if (numberOfPersons > 0)
 numberOfPersons--;
 }

 /**
 * Gets the number of persons in this room.
 * This is always greater equals 0.
 */
```

```

 * @return Number of persons.
 */
public int getNumberOfPersons() {
 return numberOfPersons;
}
}

```

| Kommentar            | Beschreibung                                                                           | Beispiel                                                 |
|----------------------|----------------------------------------------------------------------------------------|----------------------------------------------------------|
| @param               | Beschreibung der Parameter                                                             | @param a A Value.                                        |
| @see                 | Verweis auf ein anderes Paket, einen anderen Typ, eine andere Methode oder Eigenschaft | @see java.util.Date<br>@see<br>java.lang.String#length() |
| @version             | Version                                                                                | @version 1.12                                            |
| @author              | Schöpfer                                                                               | @author Christian Ullenboom                              |
| @return              | Rückgabewert einer Methode                                                             | @return Number of elements.                              |
| @exception/@throws   | Ausnahmen, die ausgelöst werden können                                                 | @exception<br>NumberFormatException                      |
| {@link Verweis}      | Ein eingebauter Verweis im Text im Code-Font. Parameter wie bei @see                   | {@link java.io.File}                                     |
| {@linkplain Verweis} | Wie {@link}, nur im normalen Font                                                      | {@linkplain java.io.File}                                |
| {@code Code}         | Quellcode im Code-Zeichensatz – auch mit HTML-Sonderzeichen                            | {@code 1 ist < 2}                                        |
| {@literal Literale}  | Maskiert HTML-Sonderzeichen. Kein Code-Zeichensatz                                     | {@literal 1 < 2 && 2 > 1}                                |
| @category            | Für Java 7 oder 8 geplant:<br>Vergabe einer Kategorie                                  | @category Setter                                         |

Tabelle 19.3: Die wichtigsten Dokumentationskommentare im Überblick



### Hinweis

Die Dokumentationskommentare sind so aufgebaut, dass der erste Satz in der Auflistung der Methoden und Attribute erscheint und der Rest in der Detailansicht:

```
/*
 * Ein kurzer Satz, der im Abschnitt "Method Summary" stehen wird.
 * Es folgt die ausführliche Beschreibung, die später im
 * Abschnitt "Method Detail" erscheint, aber nicht in der Übersicht.
 */
public void foo() { }
```

Weil ein Dokumentationskommentar `/**` mit `/*` beginnt, ist er für den Compiler ein normaler Blockkommentar. Die JavaDoc-Kommentare werden oft optisch aufgewertet, indem am Anfang jeder Zeile ein Sternchen steht – dieses ignoriert JavaDoc.

### 19.3.2 Mit dem Werkzeug javadoc eine Dokumentation erstellen

Aus dem mit Kommentaren versehenen Quellcode generiert ein externes Programm die Zieldokumente. Das JDK liefert das Konsolen-Programm *javadoc* mit aus, dem als Parameter ein Dateiname der zu kommentierenden Klasse übergeben wird; aus kompliierten Dateien können natürlich keine Beschreibungsdateien erstellt werden. Wir starten *javadoc* im Verzeichnis, in dem auch die Klassen liegen, und erhalten unsere HTML-Dokumente.



### Beispiel

Möchten wir Dokumentationen für das gesamte Verzeichnis erstellen, so geben wir alle Dateien mit der Endung *.java* an:

```
$ javadoc *.java
```

JavaDoc geht durch den Quelltext, parst die Deklarationen und zieht die Dokumentation heraus. Daraus generiert das Tool eine Beschreibung, die in der Regel als HTML-Seite zu uns kommt.



In Eclipse lässt sich eine Dokumentation mit JavaDoc sehr einfach erstellen: Im Menü FILE • EXPORT ist der Eintrag JAVADOC zu wählen, und nach einigen Einstellungen ist die Dokumentation generiert.

**Hinweis**

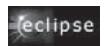
Die Sichtbarkeit spielt bei JavaDoc eine wichtige Rolle. Standardmäßig nimmt JavaDoc nur öffentliche Dinge in die Dokumentation auf.



### 19.3.3 HTML-Tags in Dokumentationskommentaren \*

In den Kommentaren können HTML-Tags verwendet werden, beispielsweise `<b>bold</b>` und `<i>italic</i>`, um Textattribute zu setzen. Sie werden direkt in die Dokumentation übernommen und müssen korrekt geschachtelt sein, damit die Ausgabe nicht falsch dargestellt wird. Die Überschriften-Tags `<h1>..</h1>` und `<h2>..</h2>` sollten jedoch nicht verwendet werden. JavaDoc verwendet sie zur Gliederung der Ausgabe und weist ihnen Formatvorlagen zu.

In Eclipse zeigt die Ansicht JAVADOC in einer Vorschau das Ergebnis des Dokumentationskommentars an.



### 19.3.4 Generierte Dateien

Für jede öffentliche Klasse erstellt JavaDoc eine HTML-Datei. Sind Klassen nicht öffentlich, muss ein Schalter angegeben werden. Die HTML-Dateien werden zusätzlich mit Querverweisen zu den anderen dokumentierten Klassen versehen. Daneben erstellt JavaDoc weitere Dateien:

- `index-all.html` liefert eine Übersicht aller Klassen, Schnittstellen, Ausnahmen, Methoden und Felder in einem Index.
- `overview-tree.html` zeigt in einer Baumstruktur die Klassen an, damit die Vererbung deutlich sichtbar ist.
- `allclasses-frame.html` zeigt alle dokumentierten Klassen in allen Unterpaketen auf.
- `deprecated-list.html` bietet eine Liste der veralteten Methoden und Klassen.
- `serialized-form.html` listet alle Klassen auf, die Serializable implementieren. Jedes Attribut erscheint mit einer Beschreibung in einem Absatz.
- `help-doc.html` zeigt eine Kurzbeschreibung von JavaDoc.
- `index.html`: JavaDoc erzeugt eine Ansicht mit Frames. Das ist die Hauptdatei, die die rechte und linke Seite referenziert. Die linke Seite ist die Datei `allclasses-frame.html`. Rechts im Frame wird bei fehlender Paketbeschreibung die erste Klasse angezeigt.

- *stylesheet.css* ist eine Formatvorlage für HTML-Dateien, in der sich Farben und Zeichensätze einstellen lassen, die dann alle HTML-Dateien nutzen.
- *packages.htm* ist eine veraltete Datei. Sie verweist auf die neuen Dateien.

### 19.3.5 Dokumentationskommentare im Überblick \*

Einige JavaDoc-Kommentare kann der Entwickler in den Block setzen, so wie `@param` oder `@return` zur Beschreibung der Parameter oder Rückgaben, andere auch in den Text, wie `{@link}` zum Setzen eines Verweises auf einen anderen Typ oder eine andere Methode. Tags der ersten Gruppe heißen *Block-Tags*, die anderen *Inline-Tags*. Bisher erkennt das JavaDoc-Tool die folgenden Tags (ab welcher Version, steht in Klammern):<sup>3</sup>

- *Block-Tags*: `@author` (1.0), `@deprecated` (1.0), `@exception` (1.0), `@param` (1.0), `@return` (1.0), `@see` (1.0), `@serial` (1.2), `@serialData` (1.2), `@serialField` (1.2), `@since` (1.1), `@throws` (1.2), `@version` (1.0)
- *Inline-Tags*: `{@code}` (1.5), `{@docRoot}` (1.3), `{@inheritDoc}` (1.4), `{@link}` (1.2), `{@linkplain}` (1.4), `{@literal}` (1.5), `{@value}` (1.4)

In Java 6 und Java 7 ist kein Tag hinzugekommen. Es gab einiges auf der Liste, was als JavaDoc-Tag in Java 8 hinzukommen soll.

## Beispiele

Eine externe Zusatzquelle geben wir wie folgt an:

```
@see Java Spec.
```

Verweis auf eine Methode, die mit der beschriebenen Methode verwandt ist:

```
@see String#equals(Object) equals
```

Von `@see` gibt es mehrere Varianten:

```
@see #field
@see #method(Type, Type,...)
@see #method(Type argname, Type argname,...)
@see #constructor(Type, Type,...)
@see #constructor(Type argname, Type argname,...)
```

---

<sup>3</sup> <http://tutego.de/go/javadoctags>

```

@see Class#field
@see Class#method(Type, Type,...)
@see Class#method(Type argname, Type argname,...)
@see Class#constructor(Type, Type,...)
@see Class#constructor(Type argname, Type argname,...)
@see Class.NestedClass
@see Class
@see package.Class#field
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type argname, Type argname,...)
@see package.Class#constructor(Type, Type,...)
@see package.Class#constructor(Type argname, Type argname,...)
@see package.Class.NestedClass
@see package.Class
@see package

```

Dokumentiere eine Variable. Gib einen Verweis auf eine Methode an:

```

/**
 * The X-coordinate of the component.
 *
 * @see #getLocation()
 */
int x = 1263732;

```

19

Eine veraltete Methode, die auf eine Alternative zeigt:

```

/**
 * @deprecated As of JDK 1.1,
 * replaced by {@link #setBounds(int,int,int,int)}
 */

```

Anstatt HTML-Tags wie `<tt>` oder `<code>` für den Quellcode zu nutzen, ist `{@code}` viel einfacher.

```

/**
 * Compares this current object with another object.
 * Uses {@code equals()} an not {@code ==}.
 */

```

### 19.3.6 JavaDoc und Doclets \*

Die Ausgabe von JavaDoc kann den eigenen Bedürfnissen angepasst werden, indem Doclets eingesetzt werden. Ein Doclet ist ein Java-Programm, das auf der Doclet-API aufbaut und die Ausgabedatei schreibt. Das Programm liest dabei wie das bekannte JavaDoc-Tool die Quelldateien ein und erzeugt daraus ein beliebiges AusgabefORMAT. Dieses Format kann selbst gewählt und implementiert werden. Wer also neben dem von JavaSoft beigefügten Standard-Doclet für HTML-Dateien Framemaker-Dateien (MIF) oder RTF-Dateien erzeugen möchte, der muss ein eigenes Doclet programmieren oder kann auf Doclets unterschiedlicher Hersteller zurückgreifen. Die Webseite <http://www.doclet.com/> listet zum Beispiel Doclets auf, die Docbook generieren oder UML-Diagramme mit aufnehmen.

Daneben dient ein Doclet aber nicht nur der Schnittstellendokumentation. Ein Doclet kann auch aufzeigen, ob es zu jeder Methode eine Dokumentation gibt oder ob jeder Parameter und jeder Rückgabewert korrekt beschrieben ist. Vor dem Durchbruch der Annotationen in Java 5 waren Doclets zur Generierung zusätzlicher Programmdateien und XML-Deskriptoren populär.<sup>4</sup>

### 19.3.7 Veraltete (deprecated) Typen und Eigenschaften

Während der Entwicklungsphase einer Software ändern sich immer wieder Methodensignaturen, oder Methoden kommen hinzu oder fallen weg. Gründe gibt es viele:

- Methoden können nicht wirklich plattformunabhängig programmiert werden, wurden aber einmal so angeboten. Nun soll die Methode nicht mehr unterstützt werden (ein Beispiel ist die Methode `stop()` eines Threads).
- Die Java-Namenskonvention soll eingeführt, ältere Methodennamen sollen nicht mehr verwendet werden. Das betrifft in erster Linie spezielle `setXXX()/getXXX()`-Methoden, die seit Version 1.1 zur Verfügung standen. So finden wir beim AWT viele Beispiele dafür. Nun heißt es zum Beispiel statt `size()` bei einer grafischen Komponente `getSize()`.
- Entwickler haben sich beim Methodennamen verschrieben. So hieß es in `FontMetrics` vorher `getMaxDecent()` und nun heißt es `getMaxDescent()`, und im `HTMLEditorKit` wird `insertAtBoundary()` zu `insertAtBoundary()`.

---

<sup>4</sup> XDoclet (<http://xdoclet.sourceforge.net/>) generiert aus Anmerkungen in den JavaDoc-Tags Mappings-Dateien für relationale Datenbanken oder Dokumente für Enterprise JavaBeans.

Es ist ungünstig, die Methoden jetzt einfach zu löschen, weil es dann zu Compilerfehlern kommt. Eine Lösung wäre daher, die Methode beziehungsweise den Konstruktor für *deprecated* zu deklarieren. @deprecated ist ein eigener Dokumentationskommentar. Sein Einsatz sieht dann etwa folgendermaßen aus (Ausschnitt aus der Klasse `java.util.Date`):

```
/*
 * Sets the day of the month of this <tt>Date</tt> object to the
 * specified value. ...
 *
 * @param date the day of the month value between 1-31.
 * @see java.util.Calendar
 * @deprecated As of JDK version 1.1,
 * replaced by <code>Calendar.set(Calendar.DAY_OF_MONTH, int date)</code>.
 */
public void setDate(int date) {
 setField(Calendar.DATE, date);
}
```

Die Kennung `@deprecated` gibt an, dass die Methode beziehungsweise der Konstruktor nicht mehr verwendet werden soll. Ein guter Kommentar zeigt auch Alternativen auf, sofern welche vorhanden sind. Die hier genannte Alternative ist die Methode `set()` aus dem `Calendar`-Objekt. Da der Kommentar in die generierte API-Dokumentation übernommen wird, erkennt der Entwickler, dass eine Methode veraltet ist.

### Hinweis

Wenn eine Methode als »veraltet« markiert ist, heißt das noch nicht, dass es sie nicht mehr geben muss. Es ist nur ein Hinweis darauf, dass die Methoden nicht mehr verwendet werden sollten und Unterstützung nicht mehr gegeben ist.

### Compilermeldungen bei veralteten Methoden

Der Compiler gibt bei veralteten Methoden eine kleine Meldung auf dem Bildschirm aus. Testen wir das an der Klasse `OldSack`:

**Listing 19.2:** OldSack.java

```
public class OldSack
{
 java.util.Date d = new java.util.Date(62, 3, 4);
}
```

Jetzt rufen wir ganz normal den Compiler auf:

```
$ javac OldSack.java
Note: OldSack.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.
```

Der Compiler sagt uns, dass der Schalter `-deprecation` weitere Hinweise gibt:

```
$ javac -deprecation OldSack.
OldSack.java:5: warning: Date(int,int,int) in java.util.Date has been deprecated
 Date d = new Date(62, 3, 4);
 ^
1 warning
```

Die Ausgabe gibt genau die Zeile mit der veralteten Anweisung an; Alternativen nennt er nicht. Allerdings ist schon interessant, dass der Compiler in die Dokumentationskommentare sieht. Eigentlich hat er mit den auskommentierten Blöcken ja nichts zu tun und überliest jeden Kommentar. Zur Auswertung der speziellen Kommentare gibt es schließlich das Extra-Tool *javadoc*, das wiederum mit dem Java-Compiler nichts zu tun hat.

**Hinweis**

Auch Klassen lassen sich als deprecated kennzeichnen (siehe etwa `java.io.LineNumberInputStream`). Dies finden wir jedoch selten in der Java-Bibliothek, und bei eigenen Typen sollte es vermieden werden.

**Die Annotation »@Deprecated«**

Seit Java 5 gibt es Annotationen, die zusätzliche Modifizierer sind. Eine Annotation `@Deprecated` (großgeschrieben) ist vorgegeben und ermöglicht es ebenfalls, Dinge als veraltet zu kennzeichnen. Dazu wird die Annotation wie ein üblicher Modifizierer etwa

für Methoden vor den Rückgabetyp gestellt. Oracle hat die oben genannte Methode `setDate()` mit dieser Annotation gekennzeichnet, wie der folgende Ausschnitt zeigt:

```
/** ...
 * @deprecated As of JDK version 1.1,
 * replaced by <code>Calendar.set(Calendar.DAY_OF_MONTH, int date)</code>.
 */
@Deprecated
public void setDate(int date) { ... }
```

Der Vorteil der Annotation `@Deprecated` gegenüber dem JavaDoc-Tag besteht darin, dass die Annotation auch zur Laufzeit sichtbar ist. Liegt vor einem Methodenaufruf ein `@Deprecated`-Tester, so kann dieser die veralteten Methoden zur Laufzeit melden. Bei dem JavaDoc-Tag übersetzt der Compiler das Programm in Bytecode und gibt zur Compilezeit eine Meldung aus, im Bytecode selbst gibt es aber keinen Hinweis.

### Veraltete Bibliotheken

Veraltetes hat sich im Laufe der Zeit genug angesammelt. In der aktuellen Version, Java 7, sind 20 Klassen (zuzüglich 4 Exceptions), 17 Schnittstellen (viele aus CORBA), 3 Annotationstypen und ein Annotationselement, über 360 Methoden, 21 Konstruktoren sowie fast 60 Variablen/Konstanten als veraltet eingestuft.<sup>5</sup>



## 19.4 Das Archivformat Jar

Die Jar-Dateien (von Java-Archiv) bilden ein Archivformat, das Zip ähnelt. Wie für ein Archivformat üblich, packt auch Jar mehrere Dateien zusammen. »Gepackt« heißt aber nicht zwingend, dass die Dateien komprimiert sein müssen, sie können einfach nur in einem Jar gebündelt sein. Ein Auspackprogramm wie WinZip kann Jar-Dateien entpacken. Hier bleibt zu überlegen, ob ein Programm wie WinZip mit der Dateiendung `.jar` verbunden werden soll oder ob das Standardverhalten bei installiertem JRE beibehalten wird: Unter Windows ist mit der Dateiendung `.jar` das JRE verbunden, das die Hauptklasse des Archivs startet.

---

<sup>5</sup> <http://download.oracle.com/javase/7/docs/jre/api/security/smardcardio/spec/javax/smardcardio/package-summary.html>

## Signieren und Versionskennungen

Microsoft vertraut bei seinen ActiveX-Controls vollständig auf Zertifikate und glaubt an eine Zurückverfolgung der Übeltäter in dem Fall, dass das Control Unsinn anstellt. Leider ist in dieser Gedankenkette ein Fehler enthalten, weil jeder sich Zertifikate ausstellen lassen kann, auch unter dem Namen Mickey Mouse.<sup>6</sup>

Überlegt angewendet, ist das Konzept jedoch gut zu verwenden, und Jar-Archive nutzen das gleiche Konzept. Sie lassen sich durch eine Signatur schützen, und die Laufzeitumgebung räumt Java-Programmen Extrarechte ein, die ein normales Programm sonst nicht hätte. Dies ist bei Programmen aus dem Intranet interessant.

Des Weiteren können Hersteller Informationen über Version und Kennung hinzufügen wie auch eine Versionskontrolle, damit nur solche Klassen eines Archivs verwendet werden, die den Verbleib in der gleichen Version gewährleisten. Ferner kam ein Archivformat hinzu, das Pakete zur Core-Plattform API hinzunehmen kann. Ein Beispiel ist etwa die 3D- und Java-Mail-API. Eigene Pakete sehen also so aus, als gehörten sie zum Standard.

### 19.4.1 Das Dienstprogramm jar benutzen

*jar* ist ein Kommandozeilenprogramm und verfügt über verschiedene Optionen, um Archive zu erzeugen, sie auszupacken und anzusehen. Die wichtigsten Formen für das Kommandozeilenprogramm sind:

- Anlegen: *jar c[Optionen] Jar-Datei Eingabedateien*
- Aktualisieren: *jar u[Optionen] Jar-Datei Eingabedateien*
- Auspacken: *jar x[Optionen] Jar-Datei*
- Inhalt anzeigen: *jar t[Optionen] Jar-Datei*
- Indexdatei INDEX.LIST erzeugen: *jar i Jar-Datei*

Je nach Aktion sind weitere Optionen möglich.

Daneben gibt es eine API im Paket `java.util.jar`, mit der alles programmiert werden kann, was auch das Dienstprogramm leistet.

---

<sup>6</sup> Obwohl dieser schon vergeben ist; doch vielleicht ist Darkwing Duck ja noch frei.

## Jar-Dateien anlegen

Die notwendige Option für das Anlegen eines neuen Archivs ist *c* (für engl. *create*). Da wir häufig die Ausgabe (das neue Archiv) in einer Datei haben wollen, geben wir zusätzlich *f* (für engl. *file*) an. Somit können wir schon unser erstes Archiv erstellen. Nehmen wir dazu an, es gibt ein Verzeichnis *images* für Bilder und die Klasse *Slider.class*. Dann packt folgende Zeile die Klasse und alle Bilder in das Archiv *slider.jar*:

```
$ jar cvf slider.jar Slider.class images
```

Während des Komprimierens geht *jar* alle angegebenen Verzeichnisse und Unterverzeichnisse durch und gibt, da zusätzlich zu *cf* der Schalter *v* gesetzt ist, auf dem Bildschirm die Dateien mit einem Kompressionsfaktor an:

```
adding: Slider.class (in=2790) (out=1506) (deflated 46%)
adding: images/ (in=0) (out=0) (stored 0%)
adding: images/darkwing.gif (in=1065) (out=801) (deflated 24%)
adding: images/volti.gif (in=173) (out=154) (deflated 10%)
adding: images/superschurke.gif (in=1076)(out=926)(deflated 13%)
adding: images/aqua.gif (in=884) (out=568) (deflated 35%)
```

Statt der Dateinamen können wir auch \* oder andere Wildcards angeben. Diese Expansionsfähigkeit ist ohnehin Aufgabe der Shell.

Möchten wir die Dateien nicht komprimiert haben, sollten wir den Schalter *O* angeben.

*jar* behält bei den zusammengefassten Dateien standardmäßig die Verzeichnisstruktur bei. In der oberen Ausgabe ist abzulesen, dass *jar* für *images* ein eigenes Verzeichnis im Archiv erstellt und die Bilder dort hineinsetzt. Der Schalter *C* (genau wie -*C* beim Kompressionsprogramm GZip) bildet diese hierarchische Struktur flach ohne Verzeichnisstruktur ab. Wenn wir mehrere Verzeichnisse zusammenpacken, lässt sich für jedes Verzeichnis bestimmen, ob die Struktur erhalten bleiben soll oder nicht. Nehmen wir zu unserem *sliders*-Archiv noch ein weiteres Verzeichnis mit Sound-Dateien hinzu, und beobachten wir die Ausgabe bei:

```
$ jar cfv0 slider.jar Slider.class images -C sounds
```

Zweierlei ist neu: Zum einen komprimiert *jar* nicht mehr (der Schalter *O* ist gesetzt), und die Option *C* erreicht, dass *jar* in das *sound*-Verzeichnis geht und dort alle Sound-Dateien in das Basisverzeichnis setzt.

Einer angelegten Archiv-Datei lassen sich später mit *u* (für engl. *update*) noch Dateien hinzufügen. Nehmen wir an, es kommt eine Bilddatei hinzu, so schreiben wir:

```
$ jar vuf slider.jar images/buchsbaum.gif
```

### **Jar-Dateien betrachten**

Die zusammengepackten Dateien zeigt die Option *tf* an:

```
$ jar tf slider.jar
META-INF/MANIFEST.MF
Slider.class
images/volti.gif
```

Zusätzlich zu unseren Dateien sehen wir eine von *jar* eigenständig hinzugefügte Manifest-Datei, die wir in Abschnitt 19.4.2, »Das Manifest«, besprechen wollen.

Fehlt die Endung, oder ist der Dateiname falsch angegeben, folgt eine etwas ungewöhnliche Fehlermeldung: `java.io.FileNotFoundException` – das heißt: ein Dateiname und dann ein Stack-Trace. Dies wirkt etwas unprofessionell.

Zum Anzeigen der Archive kommt der Schalter *t* (für engl. *table of contents*) zum Einsatz. Wir geben im Beispiel *f* an, weil wir den Dateinamen auf der Kommandozeile eintragen und nicht von der Standardeingabe etwa über eine Pipe lesen. Zusätzlich gibt uns der Schalter *v* (für engl. *verbose*) noch den Zeitpunkt der letzten Änderung und die Dateigröße aus:

```
291 Fri Dec 17 14:51:08 GMT 1999 META-INF/MANIFEST.MF
2790 Thu Dec 16 14:54:06 GMT 1999 Slider.class
173 Mon Oct 14 00:38:00 GMT 1996 images/volti.gif
```

### **Dateien aus dem Archiv extrahieren**

Der wichtigste Schalter beim Entpacken ist *x* (für engl. *extract*). Zusätzlich gilt für den Schalter *f* (*file*) das Gleiche wie beim Anzeigen: Ohne den Schalter erwartet *jar* die Archiv-Datei in der Standardeingabe. Als Parameter ist zusätzlich das Archiv erforderlich. Sind optional Dateien oder Verzeichnisse angegeben, packt *jar* nur diese aus. Nötige Verzeichnisse für die Dateien erzeugt *jar* automatisch. Hier ist Vorsicht geboten, denn *jar* überschreibt alle Dateien, die schon mit dem gleichen Namen auf dem Datenträger existieren. Das Archiv bleibt nach dem Auspacken erhalten. Wir wollen jetzt nur die Grafiken aus unserem Archiv *slider.jar* auspacken. Dazu schreiben wir:

```
$ jar vxf slider.jar images*
extracted: images\volti.gif
```

Die Option *v* haben wir eingesetzt, damit wir sehen, was *jar* genau packt. Sonst erfolgt keine Ausgabe auf der Konsole.

### 19.4.2 Das Manifest

Ohne dass die Ausgabe es zeigt, fügt *jar* beim Erzeugen eines Archivs automatisch eine Manifest-Datei namens *META-INF/MANIFEST.MF* ein. Ein Manifest enthält für ein Archiv wichtige Zusatzinformationen, wie die Signatur, die für jede Datei aufgeführt ist. Sehen wir uns einmal die Manifest-Datei an, die sich für

```
$ jar cfv slider.jar Slider.class images/volti.gif
```

ergibt. Die Einträge im Manifest erinnern an eine Property-Datei, denn auch hier gibt es immer Schlüssel und Werte, die durch einen Doppelpunkt getrennt sind:

```
Manifest-Version: 1.0
Name: Slider.class
Digest-Algorithms: SHA MD5
SHA-Digest: /RD8BF1mwd3bYXcaYYkqLjCkYdw=
MD5-Digest: WcnCNJbo08PH/ATqMHqZDw==
Name: images/volti.gif
Digest-Algorithms: SHA MD5
SHA-Digest: 9zehlViDyOfpfv0KkPECiMYvHO=
MD5-Digest: qv913KlZFi5tdPr2BjatIg==
```

19

### 19.4.3 Applikationen in Jar-Archiven starten

Dass die Dateien zusammen in einem Archiv gebündelt sind, hat den Vorteil, dass Entwickler ihren Kunden nicht mehr ein ganzes Bündel von Klassen- und Ressourcen-Dateien ausliefern müssen, sondern nur eine einzige Datei. Ein anderer Vorteil ist, dass ein Betriebssystem wie Windows oder Mac OS X standardmäßig mit der Endung *.jar* das JRE (Java Runtime Environment) verbunden hat, sodass ein Doppelklick auf einer Jar-Datei das Programm gleich startet.

## Main-Class im Manifest

Damit die Laufzeitumgebung weiß, welches `main()` welcher Klasse sie aufrufen soll, ist eine kleine Notiz mit dem Schlüssel Main-Class in der Manifest-Datei nötig:

```
Main-Class: voll.qualifieder.Klassenname.der.Klasse.mit.main
```

Dies ist sehr angenehm für den Benutzer eines Archivs, denn nun ist der Hersteller für den Eintrag des Einstiegspunkts im Manifest verantwortlich.

## Manifest-Dateien mit Main-Class-Einträgen erstellen

Wir können das *m*-Flag (für engl. *merge*) beim Dienstprogramm `jar` nutzen, um Einträge zum Manifest hinzuzufügen und auf diese Weise dem Jar-Archiv die Klasse mit der statischen `main()`-Methode mitzuteilen. Vor der Erzeugung eines Archivs erstellen wir eine Textdatei, die wir hier `MainfestMain.txt` nennen wollen, mit dem Eintrag `Main-Class`:

**Listing 19.3: MainfestMain.txt**

```
Main-Class: Main
```

Unser Slider-Programm soll die Hauptklasse `Main.class` besitzen.

Nun lässt sich die Datei `MainfestMain.txt` mit der Manifest-Datei zusammenbinden und anschließend benutzen:

```
$ jar cmf MainfestMain.txt slider.jar Main.class
$ java -jar slider.jar
$ java -jar slider.jar Main
```



### Hinweis

Seit Java 6 ermöglicht der Schalter `-e` (für *endpoint*) direkt die Angabe der ausführbaren Klasse in der Manifest-Datei des Java-Archivs:

```
$ jar cfe application.jar com.tutego.Main com/tutego/Main.class
```

## Von der Kommandozeile oder mit Doppelklick starten

Starten wir den Interpreter `java` von der Kommandozeile, gibt die Option `-jar` das Archiv an, und der Interpreter sucht nach dem Startprogramm, das durch die Manifest-Datei gegeben ist.

```
$ java -jar JarDatei.jar
```

Ausführbare Java-Archive starten wir unter Windows mit einem Doppelklick, da die Dateiendung *.jar* dazu führt, dass *javaw -jar* mit dem Dateinamen ausgeführt wird. Auch Solaris ab 2.6 erkennt Jar-Dateien in der Konsole oder dem Desktop als ausführbare Programme und startet sie selbstständig mit *java -jar*.

#### Hinweis

*java* (oder *javaw*) ignoriert die Angaben über *-cp* beziehungsweise Einträge in der Umgebungsvariable CLASSPATH, wenn ein Java-Programm mit *-jar* gestartet wird.

Das *Fat Jar Eclipse Plug-In* (<http://fjep.sourceforge.net/>) entpackt etwaige referenzierte Java-Archive und bündelt sie zu einem neuen großen Jar, das *java -jar* starten kann.





## Anhang A

# Die Klassenbibliothek

»Einer der Vorteile der Unordentlichkeit liegt darin, dass man dauernd tolle Entdeckungen macht.«

– Alan Alexander Milne (1882–1956)

Es folgt eine Übersicht über die mehr als 200 Pakete, die Java 7 deklariert. Sie beschreiben zusammen 3.777 Typen, davon 2.457 Klassen, 972 Schnittstellen, 49 Aufzählungen, 473 Ausnahmeklassen und 32 Errorklassen. Insgesamt gibt es 1.482 Objektvariablen, 4.408 statische Variablen bzw. Konstanten, 21.881 Objektmethoden in Klassen und 5.226 aus Schnittstellen, 3.039 Klassenmethoden sowie 4.973 Konstruktoren. (In Java 1.0 verteilten sich 212 Klassen auf 8 Pakete.) Die wichtigen Pakete sind fett hervorgehoben.

|                             |                                                                                                                                                             |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>java.lang</b>            | ein Paket, das automatisch eingebunden ist und unverzichtbare Klassen wie String-, Thread- oder Wrapper-Klassen enthält                                     |
| <b>java.lang.annotation</b> | Unterstützung für die in Java 5 hinzugekommenen Annotationen                                                                                                |
| <b>java.lang.instrument</b> | Bezieht der Klassenlader eine Klasse, so wird der Bytecode nicht direkt der JVM übergeben, sondern vorher modifiziert. Das nennt sich Instrumentalisierung. |
| <b>java.lang.management</b> | Überwachung der virtuellen Maschine                                                                                                                         |
| <b>java.lang.ref</b>        | Behandelt Referenzen.                                                                                                                                       |
| <b>java.lang.reflect</b>    | Mit Reflection ist es möglich, Klassen und Objekte über sich erzählen zu lassen.                                                                            |
| <b>java.applet</b>          | Stellt Klassen für Java-Applets bereit, damit diese auf Webseiten ihr Leben führen können.                                                                  |

Tabelle A.1: Pakete in Java 7

|                                        |                                                                                                                                                                  |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>java.awt</code>                  | Das Paket AWT ( <i>Abstract Windowing Toolkit</i> ) bietet Klassen zur Grafikausgabe und zur Nutzung von grafischen Bedienoberflächen.                           |
| <code>java.awt.color</code>            | Unterstützung von Farbräumen und Farbmodellen                                                                                                                    |
| <code>java.awt.datatransfer</code>     | Informationsaustausch zwischen (Java-)Programmen über die Zwischenablage des Betriebssystems                                                                     |
| <code>java.awt.dnd</code>              | Drag&Drop, um unter grafischen Oberflächen Informationen zu übertragen oder zu manipulieren                                                                      |
| <code>java.awt.event</code>            | Schnittstellen für die verschiedenen Ereignisse unter grafischen Oberflächen                                                                                     |
| <code>java.awt.font</code>             | Klassen, mit denen Zeichensätze genutzt und modifiziert werden können                                                                                            |
| <code>java.awt.geom</code>             | Paket für die Java 2D-API, um ähnlich wie im Grafikmodell von PostScript beziehungsweise PDF affine Transformationen auf beliebige 2D-Objekte anwenden zu können |
| <code>java.awt.im</code>               | Klassen für alternative Eingabegeräte                                                                                                                            |
| <code>java.awt.image</code>            | Erstellen und Manipulieren von Rastergrafiken                                                                                                                    |
| <code>java.awt.image.renderable</code> | Klassen und Schnittstellen zum allgemeinen Erstellen von Grafiken                                                                                                |
| <code>java.awt.print</code>            | Bietet Zugriff auf Drucker und kann Druckaufträge erzeugen.                                                                                                      |
| <code>java.beans</code>                | JavaBeans definieren wiederverwendbare Komponenten auf der Client-Seite, die beim Programmieren visuell konfiguriert werden können.                              |
| <code>java.beans.beancontext</code>    | Beans lassen sich in einem Bean-Kontext zusammenbringen.                                                                                                         |
| <code>java.io</code>                   | Möglichkeiten zur Ein- und Ausgabe. Dateien werden als Objekte repräsentiert. Datenströme erlauben den sequentiellen Zugriff auf die Dateiinhalte.               |
| <code>java.lang.invoke</code>          | Unterstützung für dynamische Programmiersprachen                                                                                                                 |
| <code>java.math</code>                 | beliebig lange Ganzzahlen oder Fließkommazahlen                                                                                                                  |

Tabelle A.1: Pakete in Java 7 (Forts.)

|                                       |                                                                                                                                                                 |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>java.net</code>                 | Kommunikation über Netzwerke. Bietet Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP beziehungsweise IP mit dem Internet verbinden lassen. |
| <code>java.nio</code>                 | neue IO-Implementierung (daher NIO <sup>1</sup> ) für performante Ein- und Ausgabe                                                                              |
| <code>java.nio.channels</code>        | Datenkanäle für nicht blockierende Ein- und Ausgabeoperationen                                                                                                  |
| <code>java.nio.charset</code>         | Kodierungen für die Übersetzung zwischen Byte- und Unicode-Zeichen                                                                                              |
| <code>java.nio.file</code>            | NIO.2-Dateisysteme, Datei- und Pfadzugriff                                                                                                                      |
| <code>java.nio.file.attribute</code>  | Datei- und Dateisystemattribute                                                                                                                                 |
| <code>java.rmi</code>                 | Aufruf von Methoden auf entfernten Rechnern                                                                                                                     |
| <code>java.rmi.activation</code>      | Unterstützung für die RMI-Aktivierung, wenn Objekte auf ihren Aufruf warten                                                                                     |
| <code>java.rmi.dgc</code>             | der verteilte Garbage-Collector DGC ( <i>Distributed Garbage Collection</i> )                                                                                   |
| <code>java.rmi.registry</code>        | Zugriff auf den Namensdienst unter RMI, die Registry                                                                                                            |
| <code>java.rmi.server</code>          | die Serverseite von RMI                                                                                                                                         |
| <code>java.security</code>            | Klassen und Schnittstellen für Sicherheit                                                                                                                       |
| <code>java.security.acl</code>        | unwichtig, da sie durch Klassen in <code>java.security</code> ersetzt wurden                                                                                    |
| <code>java.security.cert</code>       | Analysieren und Verwalten von Zertifikaten, Pfaden und Rückruf (Verfall) von Zertifikaten                                                                       |
| <code>java.security.interfaces</code> | Schnittstellen für RSA- und DSA-Schlüssel                                                                                                                       |
| <code>java.security.spec</code>       | Parameter der Schlüssel und Algorithmen für die Verschlüsselung                                                                                                 |
| <code>java.sql</code>                 | Zugriff auf relationale Datenbanken über SQL                                                                                                                    |

**Tabelle A.1:** Pakete in Java 7 (Forts.)

<sup>1</sup> NIO steht ja für *New IO*. Eine Aktualisierung ist NIO.2.

|                                          |                                                                                                                          |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>java.text</code>                   | Unterstützung für internationalisierte Programme.<br>Behandlung von Text, Formatierung von Datums-<br>werten und Zahlen. |
| <code>java.util</code>                   | Datenstrukturen, Raum und Zeit sowie Teile der<br>Internationalisierung, Zufallszahlen                                   |
| <code>java.util.concurrent</code>        | Hilfsklassen für nebenläufiges Programmieren,<br>etwa Thread-Pools                                                       |
| <code>java.util.concurrent.atomic</code> | atomare Operationen auf Variablen                                                                                        |
| <code>java.util.concurrent.locks</code>  | Lock-Objekte zum Sperren kritischer Bereiche                                                                             |
| <code>java.util.jar</code>               | Zugriffe auf Dateien im Archiv-Format JAR (Java<br>Archive)                                                              |
| <code>java.util.logging</code>           | Protokollieren von Programmabläufen                                                                                      |
| <code>java.util.prefs</code>             | Verwalten von Benutzer- und Systemeigenschaften                                                                          |
| <code>java.util.regex</code>             | Unterstützung von regulären Ausdrücken                                                                                   |
| <code>java.util.zip</code>               | Zugriff auf komprimierte Daten mit GZIP und<br>Archive (ZIP)                                                             |
| <code>javax.accessibility</code>         | Schnittstellen zwischen Eingabegeräten und<br>Benutzerkomponenten                                                        |
| <code>javax.activation</code>            | JavaBeans Activation Framework. Findet unter<br>anderem MIME-Typen.                                                      |
| <code>javax.activity</code>              | drei Ausnahmen im Fall von CORBA-Fehlern                                                                                 |
| <code>javax.annotation</code>            | zentrale Annotationen, etwa für Injizierung oder<br>Lebenszyklus                                                         |
| <code>javax.annotation.processing</code> | Schnittstellen für Annotation Processors                                                                                 |
| <code>javax.crypto</code>                | Klassen und Schnittstellen für kryptografische<br>Operationen                                                            |
| <code>javax.crypto.interfaces</code>     | Schnittstellen für Diffie-Hellman-Schlüssel                                                                              |
| <code>javax.crypto.spec</code>           | Klassen und Schnittstellen für Schlüssel und Para-<br>meter zur Verschlüsselung                                          |
| <code>javax.imageio</code>               | Schnittstellen zum Lesen und Schreiben von Bild-<br>dateien in verschiedenen Formaten                                    |
| <code>javax.imageio.event</code>         | Ereignisse, die während des Ladens und Speicherns<br>bei Grafiken auftauchen                                             |

Tabelle A.1: Pakete in Java 7 (Forts.)

|                                          |                                                                                             |
|------------------------------------------|---------------------------------------------------------------------------------------------|
| <code>javax.imageio.metadata</code>      | Unterstützung für beschreibende Metadaten in Bilddateien                                    |
| <code>javax.imageio.plugins.bmp</code>   | Klassen, die das Lesen und Schreiben von BMP-Bilddateien unterstützen                       |
| <code>javax.imageio.plugins.jpeg</code>  | Klassen, die das Lesen und Schreiben von JPEG-Bilddateien unterstützen                      |
| <code>javax.imageio.stream</code>        | Unterstützt das Einlesen und Schreiben von Bildern durch die Behandlung der unteren Ebenen. |
| <code>javax.jws</code>                   | Annotationen für Web-Services                                                               |
| <code>javax.jws.soap</code>              | nur Annotation und Aufzählungen für SOAP-Binding                                            |
| <code>javax.lang.model</code>            | eine Aufzählung für Java-Versionen                                                          |
| <code>javax.lang.model.element</code>    | Repräsentiert Elemente der Java-Sprache (Methode, Annotation ...).                          |
| <code>javax.lang.model.type</code>       | Repräsentiert Java-Typen (Fehlertyp, Referenztyp usw.).                                     |
| <code>javax.lang.model.util</code>       | Utility-Klassen für Programmelemente und Typen                                              |
| <code>javax.management</code>            | Management-API (JMX) mit einigen Unterpaketen                                               |
| <code>javax.management.loading</code>    | Unterstützt dynamisch geladene Klassen.                                                     |
| <code>javax.management.modelmbean</code> | Beschreibung von Model-MBeans                                                               |
| <code>javax.management.monitor</code>    | zur Beobachtung von MBeans                                                                  |
| <code>javax.management.openmbean</code>  | Definition der Open MBean als spezielle MBean                                               |
| <code>javax.management.relation</code>   | Verbindet MBeans im MBean-Server.                                                           |
| <code>javax.management.remote</code>     | Remote-Zugriff auf einen JMX-MBean Server                                                   |
| <code>javax.management.remote.rmi</code> | über RMI Remote-Zugriff auf MBean-Server                                                    |
| <code>javax.management.timer</code>      | Timer MBean meldet nach Zeitablauf Ereignisse.                                              |
| <code>javax.naming</code>                | Zugriff auf Namensdienste                                                                   |
| <code>javax.naming.directory</code>      | Zugriff auf Verzeichnisdienste, erweitert das javax.naming-Paket.                           |
| <code>javax.naming.event</code>          | Ereignisse, wenn sich etwas beim Verzeichnisdienst ändert                                   |
| <code>javax.naming.ldap</code>           | Unterstützung von LDAPv3-Operationen                                                        |
| <code>javax.net</code>                   | Klassen mit einer Socket-Fabrik                                                             |

Tabelle A.1: Pakete in Java 7 (Forts.)

|                                |                                                                        |
|--------------------------------|------------------------------------------------------------------------|
| javax.net.ssl                  | SSL-Verschlüsselung                                                    |
| javax.print                    | Java Print Service API                                                 |
| javax.print.attribute          | Attribute (wie Anzahl der Seiten, Ausrichtung) beim Java Print Service |
| javax.print.attribute.standard | Standard für einige Drucker-Attribute                                  |
| javax.print.event              | Ereignisse beim Drucken                                                |
| javax.rmi                      | Nutzen von RMI über das CORBA-Protokoll RMI-IIOP                       |
| javax.rmi.CORBA                | Unterstützt Portabilität von RMI-IIOP.                                 |
| javax.rmi.ssl                  | mit SSL mehr Sicherheit bei RMI-Verbindungen                           |
| javax.script                   | Scripting-API zum Einbinden von Skriptsprachen                         |
| javax.security.auth            | Framework für Authentifizierung und Autorisierung                      |
| javax.security.auth.callback   | Informationen wie Benutzernamen oder Passwort vom Server beziehen      |
| javax.security.auth.kerberos   | Unterstützung von Kerberos zur Authentifizierung in Netzwerken         |
| javax.security.auth.login      | Framework für die Authentifizierungsdienste                            |
| javax.security.auth.x500       | Für X.509-Zertifikate, X.500 Principal und X500 PrivateCredential      |
| javax.security.cert            | Public-Key-Zertifikate                                                 |
| javax.security.sasl            | Unterstützung für SASL (Simple Authentication and Security Layer)      |
| javax.sound.midi               | Ein- und Ausgabe, Synthesisierung von MIDI-Daten                       |
| javax.sound.sampled            | Schnittstellen zur Ausgabe und Verarbeitung von Audio-Daten            |
| javax.sql                      | Datenquellen auf Serverseite                                           |
| javax.sql.rowset               | Implementierung von RowSet                                             |
| javax.sql.rowset.serial        | Mappt SQL-Typen auf serialisierbare Java-Typen.                        |
| javax.swing                    | einfache Swing-Komponenten                                             |
| javax.swing.border             | grafische Rahmen für die Swing-Komponenten                             |
| javax.swing.colorchooser       | Anzeige vom JColorChooser, einer Komponente für die Farbauswahl        |

Tabelle A.1: Pakete in Java 7 (Forts.)

|                              |                                                                                                                                                                                                        |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| javax.swing.event            | Ereignisse der Swing-Komponenten                                                                                                                                                                       |
| javax.swing.filechooser      | Dateiauswahldialog unter Swing: JFileChooser                                                                                                                                                           |
| javax.swing.plaf             | Unterstützt auswechselbares Äußeres bei Swing durch abstrakte Klassen.                                                                                                                                 |
| javax.swing.plaf.basic       | Basisimplementierung vom Erscheinungsbild der Swing-Komponenten                                                                                                                                        |
| javax.swing.plaf.metal       | plattformunabhängiges Standarderscheinungsbild von Swing-Komponenten                                                                                                                                   |
| javax.swing.plaf.multi       | Benutzerschnittstellen, die mehrere Erscheinungsbilder kombinieren                                                                                                                                     |
| javax.swing.plaf.nimbus      | neuer Nimbus-Look-And-Feel                                                                                                                                                                             |
| javax.swing.plaf.synth       | Swing-Look-and-Feel aus XML-Dateien                                                                                                                                                                    |
| javax.swing.table            | Typen rund um die grafische Tabellenkomponente javax.swing.JTable                                                                                                                                      |
| javax.swing.text             | Unterstützung für Textkomponenten                                                                                                                                                                      |
| javax.swing.text.html        | HTMLEditorKit zur Anzeige und Verwaltung eines HTML-Texteditors                                                                                                                                        |
| javax.swing.text.html.parser | Einlesen, Visualisieren und Strukturieren von HTML-Dateien                                                                                                                                             |
| javax.swing.text.rtf         | Editorkomponente für Texte im Rich-Text-Format (RTF)                                                                                                                                                   |
| javax.swing.tree             | Zubehör für die grafische Baumansicht javax.swing.JTree                                                                                                                                                |
| javax.swing.undo             | Undo- oder Redo-Operationen, etwa für einen Texteditor                                                                                                                                                 |
| javax.tools                  | Ansprachen von Java-Tool; im Moment nur der Compiler                                                                                                                                                   |
| javax.transaction            | Ausnahmen bei Transaktionen                                                                                                                                                                            |
| javax.transaction.xa         | Beziehung zwischen Transactions-Manager und Resource-Manager für Java Transaction API (JTA), besonders für verteilte Transaktionen ( <i>Distributed Transaction Processing: The XA Specification</i> ) |
| javax.xml                    | Konstanten aus der XML-Spezifikation                                                                                                                                                                   |

Tabelle A.1: Pakete in Java 7 (Forts.)

|                                    |                                                                                  |
|------------------------------------|----------------------------------------------------------------------------------|
| javax.xml.bind                     | JAXB-Typen zum Binden von XML-Strukturen an Objekte                              |
| javax.xml.bind.annotation          | im Wesentlichen JAXB 2.0-Annotationen                                            |
| javax.xml.bind.annotation.adapters | zur Behandlung von beliebigen Java-Klassen, die JAXB auf XML-Strukturen abbilden |
| javax.xml.bind.attachment          | Attachements von JAXB                                                            |
| javax.xml.bind.helpers             | nur für Implementierer eines JAXB-Providers                                      |
| javax.xml.bind.util                | Utility-Klassen für JAXB                                                         |
| javax.xml.crypto                   | Klassen für XML-Signaturen oder zur Verschlüsselung von XML-Daten                |
| javax.xml.crypto.dom               | DOM-spezifische Klassen                                                          |
| javax.xml.crypto.dsig              | Unterstützt digitale XML-Signaturen. Hat Unterpakete.                            |
| javax.xml.datatype                 | Schema-Datentypen für Dauer und gregorianischen Kalender                         |
| javax.xml.namespace                | QName für den Namensraum                                                         |
| javax.xml.parsers                  | Einlesen von XML-Dokumenten                                                      |
| javax.xml.soap                     | Aufbau von SOAP-Nachrichten                                                      |
| javax.xml.stream                   | StAX-API für XML Pull-Parser                                                     |
| javax.xml.stream.events            | nur Schnittstellen für StAX-Event-Modus                                          |
| javax.xml.stream.util              | für einen StAX-Parser                                                            |
| javax.xml.transform                | allgemeine Schnittstellen zur Transformation von XML-Dokumenten                  |
| javax.xml.transform.dom            | Quelle oder Ziel der Transformation ist DOM.                                     |
| javax.xml.transform.sax            | Quelle oder Ziel der Transformation ist SAX.                                     |
| javax.xml.transform.stax           | Quelle oder Ziel der Transformation ist StAX.                                    |
| javax.xml.transform.stream         | Transformationen auf der Basis von linearisierten XML-Dokumenten                 |
| javax.xml.validation               | Validation nach einem Schema                                                     |
| javax.xml.ws                       | JAX-WS API mit diversen Unterpaketen                                             |
| javax.xml.xpath                    | XPath API                                                                        |

Tabelle A.1: Pakete in Java 7 (Forts.)

|                     |                                                                                     |
|---------------------|-------------------------------------------------------------------------------------|
| org.ietf.jgss       | Framework für Sicherheitsdienste wie Authentifizierung, Integrität, Vertraulichkeit |
| org.w3c.dom         | Klassen für die Baumstruktur eines XML-Dokuments nach DOM-Standard                  |
| org.w3c.dom.events  | DOM-Events                                                                          |
| org.w3c.dom.ls      | Laden und speichern von XML-Strukturen                                              |
| org.xml.sax         | Ereignisse, die beim Einlesen eines XML-Dokuments nach dem SAX-Standard auftreten   |
| org.xml.sax.ext     | zusätzliche Behandlungsroutinen für SAX2-Ereignisse                                 |
| org.xml.sax.helpers | Adapterklassen und Standardimplementierungen                                        |

Tabelle A.1: Pakete in Java 7 (Forts.)

Daneben deklariert das Paket `org.omg` eine Reihe von Unterpaketen mit CORBA-Diensten, die für unsere Betrachtung jedoch zu speziell sind. SPI-Pakete (SPI steht für *Service Provider Implementation*) definieren Verhalten für Anbieter bestimmter Funktionalitäten und sind ebenfalls nicht genannt. Nicht in der Tabelle ist auch `javax.smartcardio`, was seit Java 6 mit an Bord ist, aber nicht in der Standard-Dokumentation genannt wird. Es enthält die *Java Smart Card I/O API* und dient zum Ansprechen von Kartenlesern.<sup>2</sup>

## A.1 java.lang-Paket

### A.1.1 Schnittstellen

|               |                                                                                       |
|---------------|---------------------------------------------------------------------------------------|
| Appendable    | An die Typen lassen sich Zeichen oder Zeichenketten anhängen.                         |
| AutoCloseable | Ressourcen, die über einen speziellen try-Block automatisch geschlossen werden können |
| CharSequence  | Repräsentiert Typen, die lesenden Zugriff auf Zeichen- und Zeichenfolgen erlauben.    |
| Cloneable     | Markiert Klassen, deren Exemplare sich klonen lassen.                                 |
| Comparable    | Erlaubt das Vergleichen.                                                              |

<sup>2</sup> <http://java.sun.com/javase/6/docs/jre/api/security/smardcardio/spec/javax/smardcardio/package-summary.html>

|                                 |                                                         |
|---------------------------------|---------------------------------------------------------|
| Iterable                        | Kann einen Iterator liefern.                            |
| Readable                        | Liefert aus einer Ressource Zeichen bzw. Zeichenfolgen. |
| Runnable                        | Programmcode, den ein Thread starten kann               |
| Thread.UncaughtExceptionHandler | An den Thread gehängt, fängt es Laufzeitfehler ab.      |

### A.1.2 Klassen

|                         |                                            |
|-------------------------|--------------------------------------------|
| Boolean                 | Wrapper-Klasse für boolean                 |
| Byte                    | Wrapper-Klasse für byte                    |
| Character               | Wrapper-Klasse für char                    |
| Character.Subset        | Unicode-Zeichenbereich                     |
| Character.UnicodeBlock  | rund 200 konkrete Unicode-Zeichenbereiche  |
| Class                   | Typen in der Laufzeitumgebung              |
| ClassLoader             | Klassenlader                               |
| ClassValue              | Verbindet einen Wert mit einem Klassentyp. |
| Compiler                | nur für den JIT-Compiler nötig             |
| Double                  | Wrapper-Klasse für double                  |
| Enum                    | Basisklasse für Aufzählungen               |
| Float                   | Wrapper-Klasse für float                   |
| InheritableThreadLocal  | Verbindet Werte mit einem Thread.          |
| Integer                 | Wrapper-Klasse für int                     |
| Long                    | Wrapper-Klasse für long                    |
| Math                    | Utility-Klasse für numerische Operationen  |
| Number                  | Basisklasse für numerische Typen           |
| Object                  | absolute Basisklasse aller Java-Klassen    |
| Package                 | Informationen eines Java-Pakets            |
| Process                 | Kontrolle extern gestarteter Programme     |
| ProcessBuilder          | Optionen für externes Programm bestimmen   |
| ProcessBuilder.Redirect | Umlenkung für externes Programm definieren |
| Runtime                 | Klasse mit diversen Systemmethoden         |
| RuntimePermission       | Rechte mit Laufzeiteigenschaften           |
| SecurityManager         | Sicherheitsmanager                         |

|                   |                                                |
|-------------------|------------------------------------------------|
| Short             | Wrapper-Klasse für short                       |
| StackTraceElement | Element für den Stack-Trace                    |
| StrictMath        | numerische Operationen strikt gerechnet        |
| String            | immutable Zeichenketten                        |
| StringBuffer      | veränderbare nicht threadsichere Zeichenketten |
| StringBuilder     | veränderbare threadsichere Zeichenketten       |
| System            | Utility-Klasse mit diversen Klassenmethoden    |
| Thread            | nebenläufige Programme                         |
| ThreadGroup       | Gruppiert Threads.                             |
| ThreadLocal       | Verbindet Werte mit einem Thread.              |
| Throwable         | Basistyp für Ausnahmen                         |
| Void              | spezieller Typ für void-Rückgabe               |



# Index

|                                   |                    |
|-----------------------------------|--------------------|
| !, logischer Operator .....       | 165                |
| #ifdef .....                      | 62                 |
| #IMPLIED .....                    | 1143               |
| #REQUIRED .....                   | 1142               |
| \$, innere Klasse .....           | 694, 703           |
| %, Format-Spezifizierer .....     | 449                |
| %, Modulo-Operator .....          | 156                |
| %, Operator .....                 | 1227               |
| %b, Format-Spezifizierer .....    | 449                |
| %c, Format-Spezifizierer .....    | 450                |
| %d, Format-Spezifizierer .....    | 450                |
| %e, Format-Spezifizierer .....    | 450                |
| %f, Format-Spezifizierer .....    | 450                |
| %n, Format-Spezifizierer .....    | 449                |
| %s, Format-Spezifizierer .....    | 449                |
| %t, Format-Spezifizierer .....    | 450                |
| %x, Format-Spezifizierer .....    | 450                |
| &&, logischer Operator .....      | 165                |
| &, Generics .....                 | 813                |
| &amp; .....                       | 1139               |
| &apos .....                       | 1139               |
| &gt .....                         | 1139               |
| &lt .....                         | 1139               |
| &quot .....                       | 1139               |
| *, Multiplikationsoperator .....  | 154                |
| *, regulärer Ausdruck .....       | 411                |
| *7 .....                          | 48                 |
| +, Additionsoperator .....        | 154                |
| +, regulärer Ausdruck .....       | 411                |
| -, Subtraktionsoperator .....     | 154                |
| -, regulärer Ausdruck .....       | 411                |
| ..., variable Argumentliste ..... | 311                |
| .class .....                      | 738                |
| /, Divisionsoperator .....        | 154                |
| //, Zeilenkommentar .....         | 123                |
| =, Zuweisungsoperator .....       | 152                |
| == .....                          | 284                |
| ==, Referenzvergleich .....       | 740                |
| ? , Generics .....                | 821                |
| ? , regulärer Ausdruck .....      | 411                |
| @author, JavaDoc .....            | 1261               |
| @category, JavaDoc .....          | 1261               |
| @code, JavaDoc .....              | 1261               |
| @Deprecated .....                 | 1268               |
| @Deprecated, Annotation .....     | 334                |
| @deprecated, JavaDoc .....        | 1267               |
| @exception, JavaDoc .....         | 1261               |
| @link, JavaDoc .....              | 1261               |
| @linkplain, JavaDoc .....         | 1261               |
| @literal, JavaDoc .....           | 1261               |
| @Override .....                   | 334, 569, 596, 739 |
| @param, JavaDoc .....             | 1261               |
| @return, JavaDoc .....            | 1261               |
| @SafeVarargs .....                | 789                |
| @see, JavaDoc .....               | 1261               |
| @SuppressWarnings .....           | 335                |
| @throws, JavaDoc .....            | 1261               |
| @version, JavaDoc .....           | 1261               |
| @XmlElement .....                 | 1156               |
| @XmlRootElement .....             | 1154               |
| \, Ausmaskierung .....            | 383                |
| ^, logischer Operator .....       | 165                |
| ^, regulärer Ausdruck .....       | 412                |
| , logischer Operator .....        | 165                |

## A

---

|                                     |                     |
|-------------------------------------|---------------------|
| Abrunden .....                      | 1222                |
| abs(), Math .....                   | 1220                |
| Absolutwert .....                   | 185                 |
| Abstract Window Toolkit .....       | 1014                |
| abstract, Schlüsselwort .....       | 587, 589            |
| Abstrakte Klasse .....              | 587                 |
| Abstrakte Methode .....             | 589                 |
| Absturz der Ariane 5 .....          | 1206                |
| Accessibility .....                 | 1016                |
| ActionListener, Schnittstelle ..... | 1040, 1051,<br>1054 |
| Adapterklasse .....                 | 1046                |
| add(), Container .....              | 1037                |
| addActionListener(), JButton .....  | 1054                |
| Addition .....                      | 154                 |

|                                                    |                         |
|----------------------------------------------------|-------------------------|
| addPropertyChangeListener(),                       |                         |
| PropertyChangeSupport                              | 864                     |
| addWindowListener()                                | 1044                    |
| Adjazenzmatrix                                     | 309                     |
| Adobe Flash                                        | 69                      |
| Aggregationsfunktion                               | 1181                    |
| Ahead-Of-Time Compiler                             | 1253                    |
| Aktor                                              | 246                     |
| Al-Chwârîzmî, Ibn Mûsâ                             | 971                     |
| Algorithmus                                        | 971                     |
| Alias                                              | 278                     |
| Allgemeiner Konstruktor                            | 517                     |
| AM_PM, Calendar                                    | 889                     |
| American Standard Code for Information Interchange | 341                     |
| Amigos                                             | 245                     |
| Android                                            | 76                      |
| Anführungszeichen                                  | 149                     |
| Angepasster Exponent                               | 1217                    |
| Annotation                                         | 333                     |
| Anonyme innere Klasse                              | 701                     |
| Anpassung                                          | 861                     |
| Antialiasing                                       | 1073                    |
| Anweisung                                          | 122                     |
| <i>elementare</i>                                  | 126                     |
| <i>geschachtelte</i>                               | 183                     |
| <i>leere</i>                                       | 126                     |
| Anweisungssequenz                                  | 126                     |
| Anwendungsfall                                     | 246                     |
| Anwendungsfalldiagramm                             | 246                     |
| ANY                                                | 1141                    |
| Anzahl Einträge                                    | 1181                    |
| Aonix Perc Pico                                    | 77                      |
| Apache Commons CLI                                 | 330                     |
| Apache Commons Codec                               | 373, 447                |
| Apache Commons Lang                                | 734, 740                |
| Apache Harmony                                     | 71                      |
| append(), StringBuffer/StringBuilder               | 400                     |
| Appendable, Schnittstelle                          | 402                     |
| appendReplacement(), Matcher                       | 424                     |
| appendTail(), Matcher                              | 424                     |
| Applet                                             | 49, 68                  |
| appletviewer                                       | 1251                    |
| Applikations-Klassenlader                          | 895                     |
| APRIL, Calendar                                    | 886                     |
| Äquivalenz                                         | 165                     |
| Arcus-Funktion                                     | 1228                    |
| Arcus-Funktionen                                   | 1228                    |
| Argument                                           | 125                     |
| <i>der Funktion</i>                                | 220                     |
| Argumentanzahl, variable                           | 311                     |
| ArithmeticeException                               | 155, 634, 1239          |
| Arithmetischer Operator                            | 154                     |
| ARM-Block                                          | 668                     |
| Array                                              | 287                     |
| arraycopy(), System                                | 314                     |
| Array-Grenze                                       | 59                      |
| ArrayIndexOutOfBoundsException                     | 634                     |
| ArrayList, Klasse                                  | 545, 972, 979, 981, 984 |
| ArrayStoreException                                | 577                     |
| Array-Typ                                          | 252                     |
| ASCII                                              | 341                     |
| ASCII-Zeichen                                      | 115                     |
| asin(), Math                                       | 1228                    |
| asList(), Arrays                                   | 323, 1006               |
| assert, Schlüsselwort                              | 687                     |
| Assertion                                          | 687                     |
| AssertionError                                     | 687                     |
| Assignment                                         | 152                     |
| Assoziation                                        | 541                     |
| <i>reflexive</i>                                   | 543                     |
| <i>rekursive</i>                                   | 543                     |
| <i>zirkuläre</i>                                   | 543                     |
| Assoziativer Speicher                              | 980                     |
| atomar                                             | 960                     |
| Attribut                                           | 243–244                 |
| Attribute, XML                                     | 1137                    |
| Aufgeschobene Initialisierung                      | 233                     |
| Aufrufstapel                                       | 682                     |
| Aufrunden                                          | 1222                    |
| AUGUST, Calendar                                   | 886                     |
| Ausdruck                                           | 130                     |
| Ausdrucksanweisung                                 | 131, 153                |
| Ausführungsstrang                                  | 934                     |
| Ausnahme                                           | 59                      |
| Ausprägung                                         | 243                     |
| Ausprägungsspezifikation                           | 247                     |
| Ausprägungsvariable                                | 252                     |
| Äußere Schleife                                    | 200                     |
| Auszeichnungssprache                               | 1135                    |

|                                           |                |
|-------------------------------------------|----------------|
| Autoboxing .....                          | 732            |
| Automatic Resource Management (ARM) ..... | 668            |
| Automatische Typanpassung .....           | 559            |
| AWT .....                                 | 1014           |
| AWT-Event-Thread .....                    | 1051           |
| <b>B</b>                                  |                |
| Base64 .....                              | 447            |
| BASE64Decoder, Klasse .....               | 447            |
| BASE64Encoder, Klasse .....               | 447            |
| Baseline .....                            | 1076           |
| Basic Multilingual Plane .....            | 349            |
| BD-J .....                                | 69             |
| Bedingte Compilierung .....               | 61             |
| Bedingung, zusammengesetzte .....         | 179            |
| Bedingungsoperator .....                  | 168, 184       |
| Behinderung, Accessibility .....          | 1016           |
| Beispielprogramme der Insel .....         | 41             |
| Benutzerdefinierter Klassenlader .....    | 896            |
| Beobachter-Pattern .....                  | 849            |
| Betrag .....                              | 1220           |
| Betriebssystemunabhängigkeit .....        | 52             |
| Bezeichner .....                          | 115            |
| Bias .....                                | 1217           |
| Biased exponent .....                     | 1217           |
| Bidirektionale Beziehung .....            | 542            |
| BigDecimal, Klasse .....                  | 1238, 1246     |
| Big-Endian .....                          | 1240           |
| BigInteger, Klasse .....                  | 1239           |
| Binärer Operator .....                    | 152            |
| Binärrepräsentation .....                 | 393            |
| Binärsystem .....                         | 1203           |
| Binary Code License .....                 | 63             |
| Binary Floating-Point Arithmetic .....    | 1213           |
| binarySearch(), Arrays .....              | 321, 1005      |
| Binnenmajuskel .....                      | 117            |
| bin-Pfad .....                            | 87             |
| Bitweises exklusives Oder .....           | 1200           |
| Bitweises Oder .....                      | 1200           |
| Bitweises Und .....                       | 1200           |
| Block .....                               | 134            |
| <i>leerer</i> .....                       | 134            |
| Block-Tag .....                           | 1264           |
| Blu-ray Disc Association (BDA) .....      | 69             |
| Blu-ray Disc Java .....                   | 69             |
| BOM (Byte Order Mark) .....               | 350            |
| boolean, Datentyp .....                   | 136            |
| Boolean, Klasse .....                     | 731            |
| Bootstrap-Klassen .....                   | 893            |
| Bootstrap-Klassenlader .....              | 895            |
| BorderLayout, Klasse .....                | 1056, 1060     |
| Bound properties .....                    | 864            |
| Bound property .....                      | 862            |
| Boxing .....                              | 732            |
| BoxLayout, Klasse .....                   | 1055, 1059     |
| break .....                               | 204–205        |
| BreakIterator, Klasse .....               | 438            |
| Bruch .....                               | 1250           |
| Bruchzahl .....                           | 1213           |
| Brückenmethoden .....                     | 839            |
| BufferedInputStream, Klasse .....         | 1114, 1128     |
| BufferedOutputStream .....                | 1126           |
| BufferedReader .....                      | 1114           |
| BufferedReader, Klasse .....              | 1128           |
| BufferedWriter .....                      | 1126           |
| Byte .....                                | 1199           |
| byte, Datentyp .....                      | 137, 146, 1202 |
| Byte, Klasse .....                        | 722            |
| Bytecode .....                            | 50             |
| <b>C</b>                                  |                |
| C .....                                   | 47             |
| C++ .....                                 | 47, 243        |
| Calendar, Klasse .....                    | 883            |
| Call by Reference .....                   | 282            |
| Call by Value .....                       | 220, 282       |
| Call stack .....                          | 682            |
| CANON_EQ, Pattern .....                   | 414            |
| CardLayout, Klasse .....                  | 1056           |
| CASE_INSENSITIVE, Pattern .....           | 414            |
| CASE_INSENSITIVE_ORDER, String .....      | 1006           |
| Cast .....                                | 167            |
| Cast, casten .....                        | 170            |
| catch, Schlüsselwort .....                | 616            |
| CDATA .....                               | 1142           |
| ceil(), Math .....                        | 1222           |
| char, Datentyp .....                      | 136, 149       |

|                                     |                 |
|-------------------------------------|-----------------|
| Character, Klasse .....             | 350             |
| charAt(), String .....              | 362             |
| CharSequence, Schnittstelle .....   | 358, 407        |
| Charset, Klasse .....               | 443             |
| Checked exception .....             | 635             |
| ChoiceFormat, Klassse .....         | 466             |
| Class literal .....                 | 738             |
| Class Loader .....                  | 892             |
| Class, Klasse .....                 | 737             |
| class, Schlüsselwort .....          | 475             |
| ClassCastException .....            | 561, 634        |
| ClassLoader, Klasse .....           | 896             |
| CLASSPATH .....                     | 894, 1254, 1275 |
| -classpath .....                    | 894, 1254       |
| Class-Path-Wildcard .....           | 1258            |
| Clip-Bereich .....                  | 1070            |
| clone() .....                       | 746             |
| clone(), Arrays .....               | 313             |
| clone(), Object .....               | 746             |
| Cloneable, Schnittstelle .....      | 747             |
| CloneNotSupportedException .....    | 747, 749        |
| Closeable, Schnittstelle .....      | 1117            |
| Cloudscape .....                    | 1182            |
| cmd.exe .....                       | 918             |
| Code point .....                    | 115             |
| Codepage .....                      | 350             |
| Codepoint .....                     | 341             |
| Codeposition .....                  | 341             |
| CollationKey, Klasse .....          | 471             |
| Collator, Klasse .....              | 467, 710, 1007  |
| Collection, Schnittstelle .....     | 972, 975        |
| Collection-API .....                | 971             |
| Collections, Klasse .....           | 972             |
| Color, Klasse .....                 | 1079            |
| Command Model .....                 | 1040            |
| Command not found .....             | 86              |
| command.com .....                   | 918             |
| Comparable, Schnittstelle .....     | 600, 709, 725   |
| Comparator, Schnittstelle .....     | 709             |
| compare(), Comparator .....         | 711             |
| compare(), Wrapper-Klassen .....    | 725             |
| compareTo(), Comparable .....       | 711             |
| compareTo(), String .....           | 371             |
| compareToIgnoreCase(), String ..... | 371             |
| Compilation Unit .....              | 123, 272        |
| Compilationseinheit .....           | 272             |
| Compiler .....                      | 86              |
| concat(), String .....              | 379             |
| ConcurrentSkipListMap, Klasse ..... | 981             |
| const, Schlüsselwort .....          | 283             |
| const-korrekt .....                 | 283             |
| Constraint property .....           | 862             |
| Container .....                     | 972             |
| contains(), String .....            | 363             |
| containsKey(), Map .....            | 997             |
| contentEquals(), String .....       | 406             |
| Content-Pane .....                  | 1035            |
| continue .....                      | 204, 207        |
| Copy-Constructor .....              | 746             |
| Copy-Konstruktor .....              | 519             |
| copyOf(), Arrays .....              | 320             |
| copyOfRange(), Arrays .....         | 320             |
| CopyOnWriteArrayList, Klasse .....  | 984             |
| cos(), Math .....                   | 1228            |
| cosh(), Math .....                  | 1229            |
| Cosinus .....                       | 1228            |
| -cp .....                           | 894, 1254, 1258 |
| Cp037 .....                         | 442             |
| Cp850 .....                         | 442             |
| CREATE TABLE, SQL .....             | 1181            |
| Crimson .....                       | 1151            |
| currency, Datentyp .....            | 145             |
| Currency, Klasse .....              | 464             |
| currentThread(), Thread .....       | 944             |
| currentTimeMillis(), System .....   | 910, 915–916    |
| Customization .....                 | 861             |

## D

---

|                                  |      |
|----------------------------------|------|
| -D .....                         | 905  |
| Dalvik Virtual Machine .....     | 76   |
| Dämon .....                      | 948  |
| Dangling pointer .....           | 525  |
| Dangling-Else-Problem .....      | 181  |
| Data Hiding .....                | 489  |
| Data Query Language .....        | 1179 |
| Database Management System ..... | 1175 |
| DataInput, Schnittstelle .....   | 1097 |
| DataOutput, Schnittstelle .....  | 1097 |
| Datapoint .....                  | 51   |

|                                            |              |
|--------------------------------------------|--------------|
| DATE, Calendar .....                       | 889          |
| Date, Klasse .....                         | 880          |
| DateFormat, Klasse .....                   | 459–460      |
| Dateinamenendung .....                     | 372          |
| Datenbankausprägung .....                  | 1176         |
| Datenbankschema .....                      | 1176         |
| Datenbankverwaltungssystem .....           | 1175         |
| Datenbasis .....                           | 1175         |
| Datentyp .....                             | 135          |
| <i>ganzzahliger</i> .....                  | 146          |
| Datenzeiger .....                          | 1098         |
| DAY_OF_MONTH, Calendar .....               | 889          |
| DAY_OF_WEEK, Calendar .....                | 889          |
| DAY_OF_WEEK_IN_MONTH, Calendar .....       | 889          |
| DAY_OF_YEAR, Calendar .....                | 889          |
| dBase, JDBC .....                          | 1192         |
| DBMS .....                                 | 1175         |
| Deadlock .....                             | 935          |
| DECEMBER, Calendar .....                   | 886          |
| DecimalFormat, Klasse .....                | 462, 464     |
| Deep copy .....                            | 749          |
| deepEquals(), Arrays .....                 | 318, 755     |
| deepHashCode(), Arrays .....               | 756          |
| default .....                              | 189          |
| Default constructor → Default-Konstruktor  |              |
| Default-Konstruktor .....                  | 261, 515–516 |
| Default-Paket .....                        | 270          |
| Dekonstruktor .....                        | 525          |
| Dekrement .....                            | 167          |
| delegate .....                             | 91           |
| Delegation Model .....                     | 1040         |
| delete() .....                             | 59           |
| delete(), StringBuffer/StringBuilder ..... | 403          |
| Delimiter .....                            | 427, 437     |
| deprecated .....                           | 1252, 1267   |
| -deprecation .....                         | 1268         |
| Deque, Schnittstelle .....                 | 979          |
| Derby .....                                | 1182         |
| Dereferenzierung .....                     | 176          |
| Design-Pattern .....                       | 849          |
| Desktop, Klasse .....                      | 922          |
| Destruktor .....                           | 761          |
| Dezimalpunkt .....                         | 1213         |
| Dezimalsystem .....                        | 1203         |
| Diakritische Zeichen entfernen .....       | 473          |
| Diamantoperator .....                      | 790          |
| Diamanttyp .....                           | 790          |
| DirectX .....                              | 91           |
| Disjunktion .....                          | 165          |
| Dividend .....                             | 155          |
| Division .....                             | 154          |
| <i>Rest</i> .....                          | 1227         |
| Divisionsoperator .....                    | 155          |
| Divisor .....                              | 155          |
| Doc Comment .....                          | 1259         |
| Doclet .....                               | 1266         |
| DOCTYPE .....                              | 1143         |
| Document Object Model .....                | 1150         |
| Document Type Definition .....             | 1140         |
| Document, Klasse .....                     | 1161         |
| DocumentBuilderFactory .....               | 1152         |
| Dokumentationskommentar .....              | 1259         |
| DOM .....                                  | 1150         |
| DOMBuilder, Klasse .....                   | 1163         |
| DOS-Programm .....                         | 918          |
| DOTALL, Pattern .....                      | 414          |
| double, Datentyp .....                     | 137, 1213    |
| Double, Klasse .....                       | 722          |
| doubleToLongBits(), Double .....           | 757, 1218    |
| do-while-Schleife .....                    | 195          |
| DQL .....                                  | 1179         |
| Drag & Drop .....                          | 1016         |
| drawLine(), Graphics .....                 | 1074         |
| drawString(), Graphics .....               | 1075         |
| DST_OFFSET, Calendar .....                 | 889          |
| DTD .....                                  | 1140         |
| Duck-Typing .....                          | 214          |
| Durchschnittswert .....                    | 1181         |
| Dynamische Datenstruktur .....             | 971          |
| <hr/>                                      |              |
| <b>E</b>                                   |              |
| <hr/>                                      |              |
| -ea .....                                  | 688, 1254    |
| EBCDIC .....                               | 1131         |
| EBCDIC-Zeichensatz .....                   | 442          |
| Echtzeit-Java .....                        | 77           |
| Eclipse .....                              | 89           |
| Eclipse Translation Packs .....            | 94           |
| Edit-Distanz .....                         | 374          |
| Eigenschaft .....                          | 861          |

|                                            |          |
|--------------------------------------------|----------|
| Eigenschaften, objektorientierte .....     | 54       |
| Einfache Eigenschaft .....                 | 862      |
| Einfaches Hochkomma .....                  | 149      |
| Einfachvererbung .....                     | 551      |
| Eingeschränkte Eigenschaft .....           | 862      |
| Element, Klasse .....                      | 1164     |
| Element, XML .....                         | 1137     |
| Elementklasse .....                        | 694      |
| else, Schlüsselwort .....                  | 180      |
| Elternklasse .....                         | 548      |
| EmptyStackException .....                  | 634      |
| Enable assertions .....                    | 688      |
| Encoding .....                             | 442      |
| Endlosschleife .....                       | 194      |
| Endorsed-Verzeichnis .....                 | 894, 901 |
| Endrekursion .....                         | 235      |
| endsWith(), String .....                   | 372      |
| ENGLISH, Locale .....                      | 877      |
| Enterprise Edition .....                   | 76       |
| Entität .....                              | 1139     |
| Entity .....                               | 446      |
| Entwurfsmuster .....                       | 849      |
| Enum, Klasse .....                         | 767      |
| enum, Schlüsselwort .....                  | 509, 687 |
| Enumerator .....                           | 999      |
| EOFException .....                         | 1097     |
| equals() .....                             | 740      |
| equals(), Arrays .....                     | 318, 755 |
| equals(), Object .....                     | 285, 740 |
| equals(), String .....                     | 405      |
| equals(), StringBuilder/StringBuffer ..... | 407      |
| equals(), URL .....                        | 745      |
| equalsIgnoreCase(), String .....           | 369–370  |
| ERA, Calendar .....                        | 889      |
| Ereignis .....                             | 861      |
| Ereignisauslöser .....                     | 1039     |
| Ergebnistyp .....                          | 213      |
| Erreichbar, catch .....                    | 681      |
| Erreichbarer Quellcode .....               | 222      |
| Error, Klasse .....                        | 635, 645 |
| Erweiterte for-Schleife .....              | 297      |
| Erweiterungsklasse .....                   | 548      |
| Erweiterungs-Klassenlader .....            | 895      |
| Escape-Sequenz .....                       | 346      |
| Escape-Zeichen .....                       | 364      |
| Escher, Maurits .....                      | 240      |
| Eulersche Zahl .....                       | 1220     |
| Euro-Zeichen .....                         | 348      |
| Event .....                                | 861      |
| Event-Dispatching-Thread .....             | 1051     |
| EventListener, Schnittstelle .....         | 858      |
| EventObject, Klasse .....                  | 857      |
| Eventquelle .....                          | 1039     |
| Event-Source .....                         | 1039     |
| Excelsior JET .....                        | 1253     |
| Exception .....                            | 59       |
| Exception, Klasse .....                    | 635      |
| ExceptionInInitializerError .....          | 533      |
| exec(), Runtime .....                      | 916      |
| Executor, Schnittstelle .....              | 955      |
| ExecutorService, Schnittstelle .....       | 956      |
| Exemplar .....                             | 243      |
| Exemplarinitialisierer .....               | 535      |
| Exemplarinitialisierungsblock .....        | 704      |
| Exemplarvariable .....                     | 252, 499 |
| exit(), System .....                       | 330      |
| EXIT_ON_CLOSE, JFrame .....                | 1035     |
| Explizite Typumwandlung .....              | 561      |
| Explizites Klassenladen .....              | 893      |
| Exponent .....                             | 1217     |
| Exponentialwert .....                      | 1225     |
| Expression .....                           | 130      |
| extends, Schlüsselwort .....               | 548, 605 |
| eXtensible Markup Language .....           | 1136     |
| Extension-Verzeichnis .....                | 894      |
| <b>F</b>                                   |          |
| Fabrik .....                               | 849      |
| Fabrikmethode .....                        | 528      |
| Factory .....                              | 849      |
| Faden .....                                | 934      |
| Fakultät .....                             | 1244     |
| Fall-Through .....                         | 190      |
| FALSE, Boolean .....                       | 731      |
| false, Schlüsselwort .....                 | 136      |
| Farbe .....                                | 1079     |
| FEBRUARY, Calendar .....                   | 886      |
| Fee, die gute .....                        | 233      |
| Fehler .....                               | 636      |

|                                                    |                    |
|----------------------------------------------------|--------------------|
| Fehlercode .....                                   | 615                |
| Fehlermeldung, non-static-method .....             | 218                |
| Feld .....                                         | 287                |
| <i>nichtrechteckiges</i> .....                     | 306                |
| Feldtyp .....                                      | 252                |
| Fencepost error .....                              | 198                |
| Fenster .....                                      | 1033               |
| FIFO-Prinzip .....                                 | 979                |
| File, Klasse .....                                 | 1086               |
| file.encoding .....                                | 1131               |
| File.separatorChar .....                           | 1087               |
| FileInputStream, Klasse .....                      | 1109               |
| FileNotFoundException .....                        | 636                |
| FileOutputStream, Klasse .....                     | 1108               |
| FileReader, Klasse .....                           | 1107               |
| FileSystem, Klasse .....                           | 1100               |
| FileWriter, Klasse .....                           | 1105               |
| fill(), Arrays .....                               | 319                |
| fillInStackTrace(), Throwable .....                | 659–660            |
| final, Schlüsselwort .....                         | 232, 500, 506, 573 |
| Finale Klasse .....                                | 573                |
| Finale Methode .....                               | 573                |
| Finale Werte .....                                 | 539                |
| finalize(), Object .....                           | 761                |
| Finalizer .....                                    | 761                |
| finally, Schlüsselwort .....                       | 629                |
| find(), Matcher .....                              | 419                |
| FindBugs .....                                     | 653                |
| findClass(), ClassLoader .....                     | 896                |
| firePropertyChange(), PropertyChange-Support ..... | 864                |
| fireVetoableChange(), VetoableChange-Support ..... | 868                |
| First in, First out .....                          | 979                |
| First Person, Inc. .....                           | 48                 |
| Fitts's Law .....                                  | 1057               |
| Flache Kopie, clone() .....                        | 749                |
| Flache Objektkopie .....                           | 749                |
| Fließkommazahl .....                               | 135, 144, 1213     |
| Fließpunktzahl .....                               | 1213               |
| float, Datentyp .....                              | 137, 1213          |
| Float, Klasse .....                                | 722                |
| floatToIntBits(), Float .....                      | 757                |
| floor(), Math .....                                | 1222               |
| FlowLayout, Klasse .....                           | 1055, 1057         |
| Fluchtsymbol .....                                 | 346                |
| Flushable, Schnittstelle .....                     | 1118               |
| Font, Klasse .....                                 | 1076               |
| For-Each Loop .....                                | 193                |
| format(), Format .....                             | 459                |
| format(), PrintWriter/PrintStream .....            | 450                |
| format(), String .....                             | 449                |
| Format, Klasse .....                               | 459–460            |
| Format-Spezifizierer .....                         | 449                |
| Format-String .....                                | 449                |
| Formattable, Schnittstelle .....                   | 457                |
| Formatter, Klasse .....                            | 455                |
| for-Schleife .....                                 | 197                |
| Fortschaltausdruck .....                           | 198                |
| Fragezeichen-Operator .....                        | 152                |
| Frame .....                                        | 1033               |
| FRANCE, Locale .....                               | 877                |
| free() .....                                       | 59                 |
| FRENCH, Locale .....                               | 877                |

## G

---

|                                      |                        |
|--------------------------------------|------------------------|
| Ganzzahl .....                       | 135                    |
| Garbage-Collector .....              | 59, 251, 257, 513, 525 |
| Gaußsche Normalverteilung .....      | 1238                   |
| GC .....                             | 59, 513                |
| GC, Garbage-Collector .....          | 525                    |
| gcj .....                            | 1253                   |
| Gebundene Eigenschaft .....          | 862, 864               |
| Gegenseitiger Ausschluss .....       | 960, 964               |
| Geltungsbereich .....                | 230                    |
| Generics .....                       | 784                    |
| Generische Methode .....             | 795                    |
| Geordnete Liste .....                | 978                    |
| Geprüfte Ausnahme .....              | 624, 635               |
| GERMAN, Locale .....                 | 877                    |
| GERMANY, Locale .....                | 877                    |
| Geschachtelte Ausnahme .....         | 665                    |
| Geschachtelte Top-Level-Klasse ..... | 692                    |
| get(), List .....                    | 984                    |
| get(), Map .....                     | 996                    |
| getBoolean(), Boolean .....          | 720                    |
| getBytes(), String .....             | 442                    |
| getChars(), String .....             | 377                    |
| getClass(), Object .....             | 737                    |

getContentPane(), JFrame ..... 1035  
 getInstance(), Calendar ..... 887  
 getInteger(), Integer ..... 720  
 getProperties(), System ..... 903  
 getResource() ..... 1120  
 getResourceAsStream() ..... 1120  
 getStackTrace(), Thread ..... 683  
 Getter ..... 492, 863  
 getText(), JLabel ..... 1038  
 getText(), JTextComponent ..... 1067  
 getTimeInMillis(), Calendar ..... 887  
 ggt ..... 1239  
 GlassFish ..... 77  
 Gleichheit ..... 285  
 Gleitkommazahl ..... 1213  
 Globale Variable ..... 230  
 Glyph ..... 1075  
 GNU Classpath ..... 71  
 Google Guava ..... 652  
 Gosling, James ..... 48  
 goto, Schlüsselwort ..... 208  
 Grafischer Editor ..... 1023  
 Grammatik ..... 113  
 Graphics, Klasse ..... 1069  
 Graphics2D, Klasse ..... 1069  
 Greedy operator, regulärer Ausdruck ..... 420  
 Green-OS ..... 48  
 Green-Projekt ..... 48  
 Green-Team ..... 48  
 GregorianCalendar, Klasse ..... 883, 885  
 Gregorianischer Kalender ..... 883  
 GridBagLayout, Klasse ..... 1056  
 GridLayout, Klasse ..... 1055, 1063  
 Groovy ..... 53  
 Groß-/Kleinschreibung ..... 116, 373, 379  
 Größter gemeinsamer Teiler ..... 1239  
 group(), Matcher ..... 419  
 GroupLayout, Klasse ..... 1056  
 Grundlinie ..... 1076  
 Gruppenfunktion ..... 1181  
 Gültigkeit, XML ..... 1140  
 Gültigkeitsbereich ..... 230

**H**


---

Hangman ..... 367  
 Harmony, Apache ..... 71  
 Hashcode ..... 751  
 hashCode(), Arrays ..... 756  
 hashCode(), Object ..... 751  
 Hash-Funktion ..... 751  
 HashMap, Klasse ..... 981, 993  
 HashSet, Klasse ..... 979  
 Hash-Tabelle ..... 993  
 Hashtable ..... 993  
 Hash-Wert ..... 751  
 hasNextLine(), Scanner ..... 431  
 Hauptklasse ..... 123  
 Header-Datei ..... 61  
 Heap ..... 250  
 Heavyweight component ..... 1015  
 hexadezimale Zahl ..... 1203  
 Hexadezimalrepräsentation ..... 393  
 Hexadezimalsystem ..... 1203  
 Hilfsklasse ..... 527  
 Hoare, C. A. R. ..... 964  
 HotJava ..... 49  
 HotSpot ..... 56  
 HOUR, Calendar ..... 889  
 HOUR\_OF\_DAY, Calendar ..... 889  
 HP ..... 56  
 HSQLDB ..... 1182  
 HTML ..... 1135  
 HTML-Entity ..... 446  
 Hyperbolus-Funktionen ..... 1229

**I**


---

i18n.jar ..... 893  
 IcedTea ..... 63  
 Ich-Ansatz ..... 244  
 IDENTICAL, Collator ..... 469  
 Identifizierer ..... 115  
 Identität ..... 285, 740  
 identityHashCode(), System ..... 754, 758  
 IEEE 754 ..... 144, 157, 1213  
 IEEERemainder(), Math ..... 1227

|                                      |                         |
|--------------------------------------|-------------------------|
| if-Anweisung .....                   | 177                     |
| <i>angehäufte</i> .....              | 183                     |
| IFC .....                            | 1015                    |
| if-Kaskade .....                     | 183                     |
| Ignorierter Statusrückgabewert ..... | 621                     |
| IKVM.NET .....                       | 92                      |
| IllegalArgumentException .....       | 634, 646, 649, 651, 653 |
| IllegalMonitorStateException .....   | 634                     |
| IllegalStateException .....          | 649                     |
| IllegalThreadStateException .....    | 939                     |
| Imagination .....                    | 48                      |
| immutable .....                      | 357                     |
| Imperative Programmiersprache .....  | 122                     |
| Implikation .....                    | 165                     |
| Implizites Klassenladen .....        | 893                     |
| import, Schlüsselwort .....          | 266                     |
| Index .....                          | 287, 291                |
| Indexed property .....               | 862                     |
| Indexierte Variablen .....           | 290                     |
| indexOf(), String .....              | 364                     |
| IndexOutOfBoundsException .....      | 293–294                 |
| IndexOutOfBoundsException .....      | 650                     |
| Indizierte Eigenschaft .....         | 862                     |
| Infinity .....                       | 1213                    |
| Inkrement .....                      | 167                     |
| Inline-Tag .....                     | 1264                    |
| Innere Klasse .....                  | 691                     |
| Innere Schleife .....                | 200                     |
| InputMismatchException .....         | 435                     |
| InputStream, Klasse .....            | 1118                    |
| InputStreamReader, Klasse .....      | 445, 1132               |
| instanceof, Schlüsselwort .....      | 592                     |
| Instanz .....                        | 243                     |
| Instanzinitialisierer .....          | 535                     |
| Instanzvariable .....                | 252                     |
| int, Datentyp .....                  | 137, 146, 1202          |
| Integer, Klasse .....                | 722                     |
| IntelliJ IDEA .....                  | 91                      |
| Interaktionsdiagramm .....           | 247                     |
| Interface .....                      | 66, 587, 593            |
| interface, Schlüsselwort .....       | 594                     |
| Interface-Typ .....                  | 252                     |
| Internet Explorer .....              | 68                      |
| Internet Foundation Classes .....    | 1015                    |
| Interrupt .....                      | 951                     |
| interrupt(), Thread .....            | 951                     |
| interrupted(), Thread .....          | 953                     |
| InterruptedException .....           | 920, 946, 952           |
| Interval .....                       | 203                     |
| Introspection .....                  | 861                     |
| Invarianz .....                      | 819                     |
| IOException .....                    | 623, 636                |
| iPhone .....                         | 51                      |
| isInterrupted(), Thread .....        | 951                     |
| is-Methode .....                     | 492                     |
| isNaN(), Double/Float .....          | 1215                    |
| ISO 8859-1 .....                     | 115, 343                |
| ISO Country Code .....               | 877                     |
| ISO Language Code .....              | 877                     |
| ISO/IEC 8859-1 .....                 | 342                     |
| ISO-639-Code .....                   | 877                     |
| ISO-Abkürzung .....                  | 879                     |
| Ist-eine-Art-von-Beziehung .....     | 587                     |
| ITALIAN, Locale .....                | 877                     |
| Iterable, Schnittstelle .....        | 777, 983                |
| Iterator .....                       | 999                     |
| iterator(), Iterable .....           | 777                     |
| Iterator, Schnittstelle .....        | 777, 999                |

**J**

|                                 |            |
|---------------------------------|------------|
| J/Direct .....                  | 91         |
| J2EE .....                      | 76         |
| J2ME .....                      | 75         |
| Jacobson, Ivar .....            | 245        |
| Jahr .....                      | 889        |
| Jakarta Commons Math .....      | 1250       |
| JamaicaVM .....                 | 77         |
| JANUARY, Calendar .....         | 886        |
| JAPAN, Locale .....             | 877        |
| JAPANESE, Locale .....          | 877        |
| Jar .....                       | 1269       |
| jar, Dienstprogramm .....       | 1251, 1270 |
| -jar, java .....                | 1275       |
| Jaro-Winkler-Algorithmus .....  | 374        |
| jarsigner, Dienstprogramm ..... | 1251       |
| Java .....                      | 49         |
| Java 2D API .....               | 1016       |
| Java API for XML Parsing .....  | 1151       |

|                                        |            |
|----------------------------------------|------------|
| Java Card .....                        | 76         |
| Java Community Process (JCP) .....     | 874        |
| Java Database Connectivity .....       | 1188       |
| Java DB .....                          | 1182       |
| Java Document Object Model .....       | 1150       |
| Java EE .....                          | 76         |
| Java Foundation Classes .....          | 1016       |
| Java ME .....                          | 75         |
| Java Runtime Environment .....         | 1273       |
| Java SE .....                          | 71         |
| Java Virtual Machine .....             | 51         |
| java, Dienstprogramm .....             | 1251, 1254 |
| java, Paket .....                      | 265        |
| java.endorsed.dirs .....               | 901        |
| java.ext.dirs .....                    | 895        |
| java.nio.charset, Paket .....          | 443        |
| java.nio.file, Paket .....             | 1100       |
| java.prof .....                        | 1255       |
| java.text, Paket .....                 | 438        |
| java.util.jar, Paket .....             | 1270       |
| java.util.regex, Paket .....           | 410        |
| JavaBean .....                         | 861        |
| javac, Dienstprogramm .....            | 1251–1252  |
| JavaCompiler .....                     | 1253       |
| JavaDoc .....                          | 1260       |
| avadoc, Dienstprogramm .....           | 1251, 1262 |
| JavaFX Script .....                    | 69         |
| JavaFX-Plattform .....                 | 69         |
| JavaScript .....                       | 66         |
| Java-Security-Model .....              | 57         |
| JavaSoft .....                         | 49         |
| javaw, Dienstprogramm .....            | 1259       |
| javax, Paket .....                     | 265, 875   |
| javax.swing, Paket .....               | 1034       |
| javax.swing.text, Paket .....          | 1065       |
| javax.xml.bind.annotation, Paket ..... | 1154       |
| JAXB .....                             | 1153       |
| JAXBContext, Klasse .....              | 1154       |
| Jaxen .....                            | 1161       |
| JAXP .....                             | 1151–1152  |
| JBuilder .....                         | 112        |
| JButton, Klasse .....                  | 1051       |
| jdb .....                              | 1251       |
| JDBC .....                             | 1188       |
| JDK .....                              | 51         |
| JDOM .....                             | 1150       |
| JEditorPane, Klasse .....              | 1065       |
| JFC .....                              | 1016       |
| JFormattedTextField, Klasse .....      | 1065       |
| JFrame, Klasse .....                   | 1034, 1072 |
| JIT .....                              | 56         |
| JLabel, Klasse .....                   | 1037       |
| JOptionPane, Klasse .....              | 622        |
| JPanel, Klasse .....                   | 1055       |
| JPasswordField, Klasse .....           | 1065       |
| JRE .....                              | 1273       |
| JRuby .....                            | 53         |
| JSmooth .....                          | 1253       |
| JSR (Java Specification Request) ..... | 71         |
| JSR-203 .....                          | 1100       |
| JTextArea, Klasse .....                | 1065       |
| JTextComponent, Klasse .....           | 1067       |
| JTextField, Klasse .....               | 1065–1066  |
| JTextPane, Klasse .....                | 1065       |
| JULY, Calendar .....                   | 886        |
| JUNE, Calendar .....                   | 886        |
| Just-in-Time Compiler .....            | 56         |
| Jython .....                           | 53         |

## K

---

|                                        |         |
|----------------------------------------|---------|
| Kanonischer Pfad .....                 | 1089    |
| Kardinalität .....                     | 542     |
| Kaufmännische Rundung .....            | 1223    |
| Key .....                              | 980     |
| keytool .....                          | 1251    |
| Kindklasse .....                       | 548     |
| Klammerpaar .....                      | 216     |
| Klasse .....                           | 54, 243 |
| Klassendiagramm .....                  | 246     |
| Klasseneigenschaft .....               | 499     |
| Klassenhierarchie .....                | 548     |
| Klasseninitialisierer .....            | 532     |
| Klassenkonzept .....                   | 66      |
| Klassenlader .....                     | 57, 892 |
| Klassen-Literal .....                  | 738     |
| Klassenmethode .....                   | 218     |
| Klassenobjekt .....                    | 737     |
| Klassentyp .....                       | 252     |
| Klassenvariable, Initialisierung ..... | 534     |

|                                 |          |                                        |               |
|---------------------------------|----------|----------------------------------------|---------------|
| Klonen .....                    | 745      | length(), String .....                 | 361           |
| Kodierung, Zeichen .....        | 442      | LESS-Prinzip .....                     | 829           |
| Kommandozeilenparameter .....   | 329      | Levenshtein-Distanz .....              | 374           |
| Komma-Operator .....            | 200      | Lexikalik .....                        | 113           |
| Kommentar .....                 | 123      | Lightweight component .....            | 1018          |
| Kompilationseinheit .....       | 123      | line.separator .....                   | 905           |
| Komplement .....                | 1200     | Lineare Algebra .....                  | 1250          |
| <i>bitweises</i> .....          | 167      | lineare Kongruenzen .....              | 1236          |
| <i>logisches</i> .....          | 167      | Linie .....                            | 1073          |
| Komplexe Zahl .....             | 1250     | LinkedList, Klasse .....               | 979, 981, 984 |
| Konditionaloperator .....       | 184      | Linking .....                          | 892           |
| Konjunktion .....               | 165      | Linksassoziativität .....              | 169           |
| Konkatenation .....             | 359      | Liskov, Barbara .....                  | 562           |
| Konkrete Klasse .....           | 587      | Liskovsches Substitutionsprinzip ..... | 562           |
| Konstantenpool .....            | 387      | List, Schnittstelle .....              | 978, 983      |
| Konstruktor .....               | 261, 513 | Liste .....                            | 983           |
| <i>Vererbung</i> .....          | 552      | Listener .....                         | 856, 1040     |
| Konstruktoraufruf .....         | 249      | Literal .....                          | 117           |
| Konstruktorweiterleitung .....  | 553      | loadClass(), ClassLoader .....         | 896           |
| Kontravalenz .....              | 165      | Locale .....                           | 877           |
| Kontrollstruktur .....          | 177      | Locale, Klasse .....                   | 380, 464, 876 |
| Kopf .....                      | 213      | Lock .....                             | 964           |
| Kopfdefinition .....            | 1139     | lock(), Lock .....                     | 966           |
| KOREA, Locale .....             | 877      | log(), Math .....                      | 1226          |
| KOREAN, Locale .....            | 877      | Logischer Operator .....               | 164           |
| Kovarianter Rückgabetyp .....   | 575      | Lokale Klasse .....                    | 700           |
| Kovariantes Überschreiben ..... | 841      | Lokalisierte Zahl, Scanner .....       | 436           |
| Kovarianz bei Arrays .....      | 576      | long, Datentyp .....                   | 137, 146      |
| Kovarianz, Generics .....       | 819      | Long, Klasse .....                     | 722           |
| Kreiszahl .....                 | 1220     | longBitsToDouble(), Double .....       | 1218          |
| Kritischer Abschnitt .....      | 960      | Lower-bound Wildcard-Typ .....         | 823           |
| Kurzschluss-Operator .....      | 166      | LU-Zerlegung .....                     | 1250          |

**L**

|                                    |           |
|------------------------------------|-----------|
| lastIndexOf(), String .....        | 365       |
| Latin-1 .....                      | 342, 1131 |
| Laufzeitumgebung .....             | 50        |
| launch4j .....                     | 1253      |
| LayoutManager, Schnittstelle ..... | 1056      |
| Lebensdauer .....                  | 230       |
| Leerer String .....                | 384       |
| Leerraum, entfernen .....          | 380       |
| Leer-String .....                  | 388       |
| Leerzeichen .....                  | 437       |

**M**

|                          |               |
|--------------------------|---------------|
| Magic number .....       | 506           |
| Magische Zahl .....      | 506           |
| main() .....             | 88, 124       |
| Main-Class .....         | 1274          |
| Makro .....              | 62            |
| MANIFEST.MF .....        | 1273          |
| Mantelklasse .....       | 718           |
| Mantisse .....           | 1217          |
| Map, Schnittstelle ..... | 972, 980, 992 |
| MARCH, Calendar .....    | 886           |

|                                           |               |
|-------------------------------------------|---------------|
| Marke .....                               | 208           |
| Marker interface .....                    | 597           |
| Markierungsschnittstelle .....            | 597           |
| Marshaller, Schnittstelle .....           | 1155          |
| MaskFormatter, Klasse .....               | 458           |
| Matcher, Klasse .....                     | 410           |
| matches(), Pattern .....                  | 410           |
| matches(), String .....                   | 410           |
| MatchResult, Schnittstelle .....          | 421           |
| Math, Klasse .....                        | 1218          |
| MathContext, Klasse .....                 | 1248          |
| Matisse .....                             | 1023          |
| max(), Collections .....                  | 1008          |
| max(), Math .....                         | 1221          |
| MAX_RADIX .....                           | 355           |
| Maximalwert .....                         | 1181          |
| Maximum .....                             | 185, 1221     |
| MAY, Calendar .....                       | 886           |
| McNealy, Scott .....                      | 48            |
| Megginson, David .....                    | 1150          |
| Mehrdimensionales Array .....             | 301           |
| Mehrzahlvererbung .....                   | 599           |
| Mehrfachverzweigung .....                 | 183           |
| Member class .....                        | 694           |
| Memory leak .....                         | 525           |
| MESA .....                                | 47            |
| MessageFormat, Klasse .....               | 459–460, 465  |
| Metadaten .....                           | 333           |
| META-INF/MANIFEST.MF .....                | 1273          |
| Metaphone-Algorithmus .....               | 373           |
| Methode .....                             | 212           |
| <i>parametrisierte</i> .....              | 219           |
| <i>rekursive</i> .....                    | 233           |
| <i>statische</i> .....                    | 218           |
| <i>überladene</i> .....                   | 127           |
| Methoden überladen .....                  | 227           |
| Methodenaufruf .....                      | 125, 216, 481 |
| Methodenkopf .....                        | 213           |
| Methodenrumpf .....                       | 213           |
| Micro Edition .....                       | 75            |
| Microsoft Development Kit .....           | 91            |
| MILLISECOND, Calendar .....               | 889           |
| Millisekunde .....                        | 889           |
| MimeUtility, Klasse .....                 | 447           |
| min(), Collections .....                  | 1008          |
| min(), Math .....                         | 1221          |
| MIN_RADIX .....                           | 355           |
| MIN_VALUE .....                           | 1230          |
| Minimalwert .....                         | 1181          |
| Minimum .....                             | 185, 1221     |
| Minute .....                              | 889           |
| MINUTE, Calendar .....                    | 889           |
| Mitgliedsklasse .....                     | 694           |
| Model-View-Controller .....               | 849           |
| Modifizierer .....                        | 133           |
| Modulo .....                              | 157           |
| Monat .....                               | 889           |
| Monitor .....                             | 964           |
| monitorenter .....                        | 964           |
| monitorexit .....                         | 964           |
| Mono .....                                | 67            |
| MONTH, Calendar .....                     | 889           |
| MouseListener, Schnittstelle .....        | 1040          |
| MouseMotionListener, Schnittstelle .....  | 1040          |
| multicast .....                           | 91            |
| Multi-catch .....                         | 641           |
| Multilevel continue .....                 | 208           |
| MULTILINE, Pattern .....                  | 414           |
| Multiline-Modus, regulärer Ausdruck ..... | 418           |
| Multiplikation .....                      | 154           |
| Multiplizität .....                       | 542           |
| Multitaskingfähig .....                   | 933           |
| Multithreaded .....                       | 934           |
| Muster, regulärer Ausdruck .....          | 409           |
| Mutex .....                               | 964           |
| MyEclipse .....                           | 112           |

## N

|                                    |                 |
|------------------------------------|-----------------|
| name(), Enum .....                 | 768             |
| Namensraum .....                   | 1147            |
| NaN .....                          | 155, 1213, 1230 |
| NAND-Gatter .....                  | 165             |
| nanoTime(), System .....           | 910             |
| Narrowing conversion .....         | 171             |
| Native Methode .....               | 65              |
| native2ascii, Dienstprogramm ..... | 349, 445        |
| Nativer Compiler .....             | 1253            |
| Nativer Thread .....               | 934             |
| Natural ordering .....             | 709             |

Natürliche Ordnung ..... 709, 1004  
 Naughton, Patrick ..... 48  
 NavigableMap, Schnittstelle ..... 981, 994  
 Nebeneffekt ..... 481  
 Negative Zeichenklassen ..... 412  
 NEGATIVE\_INFINITY ..... 1230  
 Negatives Vorzeichen ..... 152  
 Nested exception ..... 665  
 Nested top-level class ..... 692  
 NetBeans ..... 90  
 Netscape ..... 66, 1015  
 new line ..... 905  
 new, Schlüsselwort ..... 249, 513  
 newLine(), BufferedWriter ..... 1128  
 nextLine(), Scanner ..... 431  
 Nicht ..... 165  
 Nicht geprüfte Ausnahme ..... 635  
 Nicht-primitives Feld ..... 298  
 NIO.2 ..... 1100  
 No-arg-constructor → No-Arg-Konstruktor  
 No-Arg-Konstruktor ..... 261, 514, 516  
 Non-greedy operator, regulärer Ausdruck . 421  
 nonNull(), Objects ..... 652  
 NOR-Gatter ..... 165  
 normalize(), Normalizer ..... 473  
 Normalizer, Klasse ..... 473  
 Normalverteilung ..... 1238  
 NoSuchElementException ..... 999  
 Not a Number ..... 1213, 1230  
 Notation ..... 245  
 notifyObservers(), Observable ..... 850  
 NOVEMBER, Calendar ..... 886  
 nowarn ..... 1252  
 NULL ..... 615  
 null, Schlüsselwort ..... 274  
 Nullary constructor ..... 514  
 NullPointerException ..... 275, 293, 634, 651  
 Null-Referenz ..... 274  
 Null-String ..... 388  
 Number, Klasse ..... 722  
 NumberFormat, Klasse ..... 459–460, 462  
 NumberFormatException ..... 392, 617, 622  
 Numeric promotion ..... 154  
 Numerische Umwandlung ..... 154

**O**


---

Oak ..... 48  
 Oberklasse ..... 548  
 Object Management Group (OMG) .... 246, 875  
 Object, Klasse ..... 551, 737  
 Objective-C ..... 66  
 Objects, Klasse ..... 764  
 Objektansatz ..... 244  
 Objektdiagramm ..... 246  
 Objektgleichheit ..... 740  
 Objektidentifikation ..... 738  
 Objektorientierter Ansatz ..... 66  
 Objektorientierung ..... 54, 132  
 Objekttyp ..... 560  
 Objektvariable ..... 252  
 Objektvariable, Initialisierung ..... 530  
 Observable, Klasse ..... 850  
 Observer, Schnittstelle ..... 850  
 Observer/Observable ..... 849  
 OCTOBER, Calendar ..... 886  
 ODBC ..... 1189  
 Oder ..... 165  
     *ausschließendes* ..... 165  
     *bitweises* ..... 168  
     *exklusives* ..... 165  
     *logisches* ..... 168  
 Off-by-one error ..... 198  
 Oktalsystem ..... 1203  
 Oktalzahlrepräsentation ..... 393  
 OMG ..... 246  
 OO-Methode ..... 245  
 OpenJDK ..... 62  
 OpenJDK 7 ..... 63  
 Operator ..... 152  
     *arithmetischer* ..... 154  
     *binärer* ..... 152  
     *einstelliger* ..... 152  
     *logischer* ..... 164  
     *Rang eines* ..... 167  
     *relationaler* ..... 162  
     *ternärer* ..... 184  
     *trinärer* ..... 184  
     *unärer* ..... 152  
     *zweistelliger* ..... 152

|                                          |                    |
|------------------------------------------|--------------------|
| Operator precedence .....                | 167                |
| Oracle Corporation .....                 | 49                 |
| Oracle JDK .....                         | 51                 |
| ordinal(), Enum .....                    | 770                |
| Ordinalzahl, Enum .....                  | 769                |
| org.jdom, Paket .....                    | 1160               |
| org.omg, Paket .....                     | 1285               |
| OutOfMemoryError .....                   | 250, 645, 747      |
| OutputStream, Klasse .....               | 1116               |
| OutputStreamWriter, Klasse .....         | 445, 1131          |
| <br><b>P</b>                             |                    |
| package, Schlüsselwort .....             | 269                |
| paint(), Frame .....                     | 1068               |
| paintComponent() .....                   | 1072               |
| Paket .....                              | 265                |
| Paketsichtbarkeit .....                  | 494                |
| Palrang, Joe .....                       | 48                 |
| Parameter .....                          | 219                |
| <i>aktueller</i> .....                   | 220                |
| <i>formaler</i> .....                    | 219                |
| Parameterliste .....                     | 213, 216           |
| Parameterloser Konstruktor .....         | 514                |
| Parameterübergabemechanismus .....       | 220                |
| Parametrisierter Konstruktor .....       | 517                |
| Parametrisierter Typ .....               | 786                |
| parseBoolean(), Boolean .....            | 391                |
| parseByte(), Byte .....                  | 391                |
| Parsed Character Data .....              | 1141               |
| parseDouble(), Double .....              | 391                |
| ParseException .....                     | 461                |
| parseFloat(), Float .....                | 391                |
| parseInt(), Integer .....                | 391, 396, 622, 727 |
| parseLong(), Long .....                  | 391, 396           |
| parseObject(), Format .....              | 459                |
| parseShort(), Short .....                | 391                |
| Partiell abstrakte Klasse .....          | 589                |
| PATH .....                               | 87                 |
| Path, Klasse .....                       | 1101               |
| Paths, Klasse .....                      | 1101               |
| Pattern, Klasse .....                    | 410                |
| Pattern, regulärer Ausdruck .....        | 409                |
| Pattern-Flags .....                      | 415                |
| Pattern-Matcher .....                    | 410                |
| Payne, Jonathan .....                    | 49                 |
| PCDATA .....                             | 1141               |
| p-code .....                             | 51                 |
| PDA .....                                | 75                 |
| PECS .....                               | 829                |
| Peer-Klassen .....                       | 1014               |
| Peirce-Funktion .....                    | 165                |
| Persistenz .....                         | 862                |
| phoneMe .....                            | 75                 |
| PicoJava .....                           | 51                 |
| Plattformunabhängigkeit .....            | 52                 |
| Pluggable Look & Feel .....              | 1016               |
| Plugin, Eclipse .....                    | 106                |
| Plus, überladenes .....                  | 175                |
| Plus/Minus, unäres .....                 | 167                |
| Point, Klasse .....                      | 244, 249           |
| Pointer .....                            | 57                 |
| Polar-Methode .....                      | 1238               |
| policytool .....                         | 1251               |
| Polymorphie .....                        | 580                |
| POSITIVE_INFINITY .....                  | 1230               |
| Post-Dekrement .....                     | 161                |
| Post-Inkrement .....                     | 161                |
| Potenz .....                             | 1225               |
| Prä-Dekrement .....                      | 161                |
| Präfix .....                             | 372                |
| Prä-Inkrement .....                      | 161                |
| Preferences, Klasse .....                | 924                |
| PRIMARY, Collator .....                  | 469                |
| print() .....                            | 127, 228           |
| printf() .....                           | 128                |
| printf(), PrintWriter/PrintStream .....  | 450                |
| println() .....                          | 127                |
| printStackTrace(), Throwable .....       | 621                |
| PriorityQueue, Klasse .....              | 979                |
| private, Schlüsselwort .....             | 487                |
| Privatsphäre .....                       | 487                |
| Process, Klasse .....                    | 920                |
| ProcessBuilder, Klasse .....             | 917                |
| Profiler .....                           | 912                |
| Profiling .....                          | 910                |
| Profiling-Informationen .....            | 1255               |
| Programm .....                           | 123                |
| Programmieren gegen Schnittstellen ..... | 599                |
| Programmiersprache, imperative .....     | 122                |

|                                               |                |
|-----------------------------------------------|----------------|
| Properties, Bean .....                        | 862            |
| Properties, Klasse .....                      | 903            |
| Property .....                                | 492, 861       |
| PropertyChangeEvent, Klasse .....             | 864            |
| PropertyChangeListener, Schnittstelle .....   | 864            |
| PropertyChangeSupport, Klasse .....           | 864            |
| Property-Design-Pattern .....                 | 861            |
| Property-Sheet .....                          | 861            |
| PropertyVetoException .....                   | 867            |
| protected, Schlüsselwort .....                | 552            |
| Protocols .....                               | 66             |
| Prozess .....                                 | 933            |
| Pseudo-Primzahltest .....                     | 1239           |
| public, Schlüsselwort .....                   | 487            |
| Punkt-Operator .....                          | 252            |
| Pure abstrakte Klasse .....                   | 589            |
| put(), Map .....                              | 995            |
| <br><b>Q</b>                                  |                |
| qNaNs .....                                   | 1216           |
| Quadratwurzel .....                           | 1225           |
| Quantifizierer .....                          | 411            |
| Quasiparallelität .....                       | 933            |
| Queue, Schnittstelle .....                    | 979            |
| Quiet NaN .....                               | 1216           |
| quote(), Pattern .....                        | 383            |
| quoteReplacement(), Matcher .....             | 425            |
| <br><b>R</b>                                  |                |
| Race condition .....                          | 963            |
| Race hazard .....                             | 963            |
| Random .....                                  | 1236           |
| random(), Math .....                          | 300, 1229      |
| Random, Klasse .....                          | 1236           |
| RandomAccessFile, Klasse .....                | 1095           |
| Range-Checking .....                          | 59             |
| Rangordnung .....                             | 167            |
| Raw-Type .....                                | 807            |
| Reader, Klasse .....                          | 1122           |
| readLine(), BufferedReader .....              | 1131           |
| readPassword(), Console .....                 | 915            |
| Real-time Java .....                          | 77             |
| Rechenungsgenauigkeit .....                   | 202            |
| Rechtsassoziativität .....                    | 169            |
| ReentrantLock, Klasse .....                   | 967            |
| Reference Concrete Syntax .....               | 1137           |
| Referenz .....                                | 57             |
| Referenzierung .....                          | 176            |
| Referenztyp .....                             | 131, 136, 560  |
| Referenztyp, Vergleich mit == .....           | 284            |
| Referenzvariable .....                        | 251            |
| reg .....                                     | 931            |
| regionMatches(), String .....                 | 372            |
| Registry .....                                | 925            |
| Regular expression → Regulärer Ausdruck ..... |                |
| Regulärer Ausdruck .....                      | 409            |
| Reihung .....                                 | 287            |
| Reine abstrakte Klasse .....                  | 589            |
| Rekursionsform .....                          | 235            |
| Rekursive Methode .....                       | 233            |
| rekursiver Type-Bound .....                   | 813            |
| Relationales Datenbanksystem .....            | 1188           |
| Remainder Operator .....                      | 156            |
| replace(), String .....                       | 382            |
| replaceAll(), String .....                    | 382            |
| replaceFirst(), String .....                  | 382            |
| Rest der Division .....                       | 1227           |
| Restwert-Operator .....                       | 154, 156, 1227 |
| Resultat .....                                | 131            |
| rethrow, Ausnahmen .....                      | 657            |
| return, Schlüsselwort .....                   | 221, 310       |
| Reverse-Engineering-Tool .....                | 247            |
| RFC 1521 .....                                | 447            |
| Rich Internet Applications (RIA) .....        | 69             |
| rint(), Math .....                            | 1223           |
| round(), Math .....                           | 1223           |
| RoundingMode, Aufzählung .....                | 1248           |
| Roundtrip-Engineering .....                   | 248            |
| rt.jar .....                                  | 893            |
| RTSJ .....                                    | 77             |
| Rückgabetyp .....                             | 213            |
| Rückgabewert .....                            | 217            |
| Rumpf .....                                   | 213            |
| run(), Runnable .....                         | 938            |
| Runden .....                                  | 1222           |
| Rundungsfehler .....                          | 157            |
| Rundungsmodi, BigDecimal .....                | 1247           |
| runFinalizersOnExit(), System .....           | 763            |

Runnable, Schnittstelle ..... 703, 938  
 Runtime, Klasse ..... 917  
 RuntimeException ..... 633  
 Runtime-Interpreter ..... 50

**S**

SAM (Single Abstract Method) ..... 589  
 SAP NetWeaver Developer Studio ..... 112  
 SAX ..... 1150  
 SAXBuilder, Klasse ..... 1162  
 Scala ..... 53  
 Scanner, Klasse ..... 429, 623  
 ScheduledThreadPoolExecutor, Klasse ..... 955  
 Scheduler ..... 933, 959  
 Schema ..... 1144  
 Schlange ..... 979  
 Schleifen ..... 192  
 Schleifenbedingung ..... 195, 201  
 Schleifen-Inkrement ..... 198  
 Schleifentest ..... 198  
 Schleifenzähler ..... 198  
 Schlüssel ..... 980  
 Schlüsselwort ..... 118  
*reserviertes* ..... 118  
 Schnittstelle ..... 66, 593  
 Schnittstellentyp ..... 252  
 Schriftlinie ..... 1076  
 Schwergewichtige Komponente ..... 1015  
 Scope ..... 230  
 Sealing, Jar ..... 700  
 SECOND, Calendar ..... 889  
 SECONDARY, Collator ..... 469  
 SecondString-Projekt ..... 374  
 SecureRandom, Klasse ..... 1236  
 Security-Manager ..... 57  
 sedezial ..... 1203  
 Sedezimalsystem ..... 1203  
 Sedezialzahl ..... 1203  
 Seed ..... 1236–1237  
 Seiteneffekt ..... 481  
 Sekunde ..... 889  
 Selbstbeobachtung ..... 861  
 Semantik ..... 113  
 Separator ..... 114

SEPTEMBER, Calendar ..... 886  
 SEQUEL ..... 1176  
 Sequenz ..... 972, 978  
 Sequenzdiagramm ..... 247  
 Service Provider Implementation ..... 1285  
 Set, Schnittstelle ..... 979, 987  
 setChanged(), Observable ..... 850  
 setDefaultCloseOperation(), JFrame ..... 1035, 1045  
 setFont(), Graphics ..... 1076  
 setLayout(), Container ..... 1056  
 Setter ..... 492, 863  
 setText(), JButton ..... 1053  
 setText(), JLabel ..... 1038  
 setText(), JTextField ..... 1067  
 Set-Top-Box ..... 48  
 setVisible(), Window ..... 1036  
 SGML ..... 1136  
 Shallow copy ..... 749  
 Shefferscher Strich ..... 165  
 Sheridan, Mike ..... 48  
 Shift ..... 167  
 Shift-Operator ..... 1208  
 short, Datentyp ..... 137, 146, 1202  
 Short, Klasse ..... 722  
 Short-Circuit-Operator ..... 166  
 Sichtbarkeit ..... 230, 487, 552  
 Sichtbarkeitsmodifizierer ..... 487  
 signaling NaN ..... 1216  
 Signatur ..... 214  
 Silverlight ..... 67, 69  
 Simple API for XML Parsing ..... 1150  
 SIMPLIFIED\_CHINESE, Locale ..... 877  
 SIMULA ..... 66  
 Simula-67 ..... 241  
 sin(), Math ..... 1228  
 Single inheritance ..... 551  
 Singleton ..... 527  
 sinh(), Math ..... 1229  
 Sinus ..... 1228  
 sizeof ..... 176  
 Slash ..... 1087  
 sleep(), Thread ..... 945  
 Slivka, Ben ..... 92  
 Smalltalk ..... 54, 241

|                                               |               |
|-----------------------------------------------|---------------|
| Smiley .....                                  | 348           |
| sNaN .....                                    | 1216          |
| Software-Architektur .....                    | 847           |
| Sommerzeitabweichung .....                    | 889           |
| sort(), Arrays .....                          | 317, 1005     |
| sort(), Collections .....                     | 1005          |
| SortedMap, Schnittstelle .....                | 994           |
| Sortieren .....                               | 1005          |
| Soundex-Algorithmus .....                     | 373           |
| Späte dynamische Bindung .....                | 579           |
| SPI-Pakete .....                              | 1285          |
| split(), Pattern .....                        | 429           |
| split(), String .....                         | 427           |
| SpringLayout, Klasse .....                    | 1056          |
| Sprungmarke, switch .....                     | 188           |
| Sprungziel, switch .....                      | 188           |
| SQL .....                                     | 1176          |
| SQL 2 .....                                   | 1176          |
| SQuirreL .....                                | 1184          |
| Stabil sortieren .....                        | 1005          |
| Stack .....                                   | 237           |
| Stack-Case-Labels .....                       | 191           |
| Stack-Inhalt .....                            | 684           |
| StackOverflowError .....                      | 237, 645      |
| Stack-Speicher .....                          | 250           |
| Stack-Trace .....                             | 619, 682      |
| StackTraceElement, Klasse .....               | 682           |
| Standard Extension API .....                  | 875           |
| Standard Generalized Markup<br>Language ..... | 1136          |
| Standard-Konstruktor .....                    | 261, 514      |
| Star Seven .....                              | 48            |
| Stark typisiert .....                         | 135           |
| start(), Thread .....                         | 939           |
| startsWith(), String .....                    | 372           |
| Statement .....                               | 122           |
| static final .....                            | 594           |
| static, Schlüsselwort .....                   | 133, 500      |
| Statisch typisiert .....                      | 135           |
| Statische Eigenschaft .....                   | 499           |
| Statische innere Klasse .....                 | 692           |
| Statischer Block .....                        | 532           |
| Statischer Import .....                       | 272           |
| Stellenwertsystem .....                       | 1203          |
| Steuerelement, grafisches .....               | 1013          |
| Stilles NaN .....                             | 1216          |
| StreamEncoder .....                           | 1131          |
| Streng typisiert .....                        | 135           |
| strictfp, Schlüsselwort .....                 | 1235          |
| StrictMath, Klasse .....                      | 1235          |
| String .....                                  | 125, 357      |
| Anhängen an einen .....                       | 378           |
| Länge .....                                   | 361           |
| StringBuffer, Klasse .....                    | 358, 397      |
| StringBuilder, Klasse .....                   | 358, 397      |
| StringIndexOutOfBoundsException .....         | 363, 375      |
| Stringkonkatenation .....                     | 167           |
| String-Literal .....                          | 359           |
| StringReader, Klasse .....                    | 1114          |
| String-Teil vergleichen .....                 | 372           |
| Stringteile extrahieren .....                 | 362, 374      |
| StringTokenizer, Klasse .....                 | 436           |
| Stroustrup, Bjarne .....                      | 243           |
| Structured English Query Language .....       | 1176          |
| Subinterface .....                            | 605           |
| Subklasse .....                               | 548           |
| Substitutionsprinzip .....                    | 562           |
| substring(), String .....                     | 374           |
| Subtraktion .....                             | 154           |
| Suffix .....                                  | 372           |
| Summe aller Einträge .....                    | 1181          |
| Sun Microsystems .....                        | 49            |
| sun.boot.class.path .....                     | 895           |
| sun.misc, Paket .....                         | 447           |
| sun.nio.cs .....                              | 1131          |
| SunWorld .....                                | 49            |
| super .....                                   | 558           |
| super() .....                                 | 552, 555, 558 |
| super, Schlüsselwort .....                    | 570           |
| Superklasse .....                             | 548           |
| suppressed exception .....                    | 633           |
| Surrogate-Paar .....                          | 349           |
| switch-Anweisung .....                        | 187           |
| Symbolische Konstante .....                   | 506           |
| Symmetrie, equals() .....                     | 743           |
| sync() .....                                  | 1110          |
| Synchronisation .....                         | 764, 958      |
| SynerJ .....                                  | 90            |
| Syntax .....                                  | 113           |

Synthetische Methode ..... 694, 843  
**S**  
 System.err ..... 132, 621  
 System.in ..... 918, 1118  
 System.out ..... 132  
 Systemeigenschaft ..... 87, 903  
 System-Klassenlader ..... 895

Tabulator ..... 437  
 Tag ..... 889, 1135  
 Tag des Jahres ..... 889  
 TAIWAN ..... 877  
 tan(), Math ..... 1228  
 tangle ..... 1259  
 tanh(), Math ..... 1229  
 TCFTC ..... 630  
 Teilstring ..... 363  
 Terminiert ..... 948  
 TERTIARY, Collator ..... 469  
 this\$0, innere Klasse ..... 698  
 this() ..... 558  
 this(), Beschränkungen ..... 523  
 this(), Konstruktoraufruf ..... 522  
 this, Vererbung ..... 707  
 this-Referenz ..... 483, 558  
 this-Referenz, innere Klasse ..... 696  
 Thread ..... 53, 934  
 Thread, Klasse ..... 703, 939  
 Thread-Pool ..... 955  
 ThreadPoolExecutor, Klasse ..... 955  
 Thread-safe ..... 960  
 Thread-sicher ..... 960  
 throw, Schlüsselwort ..... 646  
 Throwable, Klasse ..... 635  
 throws Exception ..... 640  
 throws, Schlüsselwort ..... 626  
 Tiefe Kopie, clone() ..... 749  
 toBinaryString(), Integer/Long ..... 393  
 toCharArray(), String ..... 377  
 toHexString(), Integer/Long ..... 393  
 Token ..... 113, 437  
 toLowerCase(), Character ..... 354  
 toLowerCase(), String ..... 379  
 toOctalString(), Integer/Long ..... 393

Top-Level-Container ..... 1033  
 toString(), Arrays ..... 316  
 toString(), Object ..... 566, 738  
 toString(), Point ..... 255–256  
 toUpperCase(), Character ..... 354  
 toUpperCase(), String ..... 379  
 TreeMap, Klasse ..... 972, 981, 993  
 Trennzeichen ..... 114, 427  
 trim(), String ..... 380  
 true ..... 136  
 TRUE, Boolean ..... 731  
 try mit Ressourcen ..... 668  
 try, Schlüsselwort ..... 616  
 Tupel ..... 1175  
 Türme von Hanoi ..... 237  
**Typ**  
*arithmetischer* ..... 136  
*generischer* ..... 785  
*integraler* ..... 136  
*numerischer* ..... 131  
*primitiver* ..... 136  
 Typanpassung ..... 170  
*automatische* ..... 170  
*explizite* ..... 170  
 type erasure ..... 800  
 TYPE, Wrapper-Klassen ..... 738  
 Typecast ..... 170  
 Typ-Inferenz ..... 789, 796  
 Typlösung ..... 800  
 Typ-Token ..... 834  
 Typvariable ..... 785  
 Typvergleich ..... 168  
**U**  
 U+, Unicode ..... 343  
 Überdecken, Methoden ..... 582  
 Überladene Methode ..... 227  
 Überladener Operator ..... 61  
 Überlagert, Methode ..... 566  
 Überlauf ..... 1230  
 Überschreiben, Methoden ..... 566  
 Übersetzer ..... 86  
 UCSD-Pascal ..... 51  
 UK, Locale ..... 877

|                                                  |                |
|--------------------------------------------------|----------------|
| Umbrella-JSR .....                               | 71             |
| Umgebungsvariablen, Betriebssystem .....         | 908            |
| Umkehrfunktion .....                             | 1228           |
| UML .....                                        | 244            |
| Umlaut .....                                     | 347            |
| Unärer Operator .....                            | 152            |
| Unäres Minus .....                               | 158            |
| Unäres Plus/Minus .....                          | 167            |
| Unbenanntes Paket .....                          | 270            |
| Unboxing .....                                   | 732            |
| UncaughtExceptionHandler,<br>Schnittstelle ..... | 953            |
| Unchecked exception .....                        | 635            |
| Und .....                                        | 165            |
| <i>bitweises</i> .....                           | 168            |
| <i>logisches</i> .....                           | 168            |
| UNDECIMBER, Calendar .....                       | 886            |
| Unendlich .....                                  | 1213           |
| Ungeprüfte Ausnahme .....                        | 655            |
| Unicode 5.1 .....                                | 343            |
| UNICODE_CASE, Pattern .....                      | 414            |
| Unicode-Escape .....                             | 347            |
| Unicode-Konsortium .....                         | 343            |
| Unicode-Zeichen .....                            | 115            |
| Unidirektionale Beziehung .....                  | 541            |
| Unified Method .....                             | 245            |
| unlock(), Lock .....                             | 966            |
| Unmarshaller, Schnittstelle .....                | 1155           |
| Unnamed package .....                            | 270            |
| UnsupportedOperationException .....              | 634, 677, 1002 |
| Unterklasse .....                                | 548            |
| Unterstrich in Zahlen .....                      | 148            |
| Unzahl .....                                     | 155            |
| update(), Observer .....                         | 854            |
| Upper-bound Wildcard-Typ .....                   | 823            |
| URL, Klasse .....                                | 623            |
| URLClassLoader, Klasse .....                     | 897            |
| US, Locale .....                                 | 877            |
| Use-Cases-Diagramm .....                         | 246            |
| useDelimiter(), Scanner .....                    | 434            |
| UTF-16-Kodierung .....                           | 345, 349, 1139 |
| UTF-32-Kodierung .....                           | 345            |
| UTF-8 .....                                      | 345, 1139      |
| Utility-Klasse .....                             | 527            |

**V**

|                                  |            |
|----------------------------------|------------|
| Valid, XML .....                 | 1140       |
| Value .....                      | 980        |
| valueOf(), Enum .....            | 768        |
| valueOf(), String .....          | 389        |
| valueOf(), Wrapper-Klassen ..... | 719        |
| Vararg .....                     | 311        |
| Variablendeklaration .....       | 139        |
| Variableninitialisierung .....   | 583        |
| Vector, Klasse .....             | 984        |
| -verbose .....                   | 1252, 1254 |
| Verbundoperator .....            | 159        |
| Verdeckte Variablen .....        | 483        |
| Vererbte Konstante .....         | 606        |
| Vererbung .....                  | 257, 547   |
| Vergleichsoperator .....         | 162        |
| Vergleichsstring .....           | 373        |
| Verkettete Liste .....           | 973        |
| Verklemmung .....                | 935        |
| Verschiebeoperator .....         | 1208       |
| Verzeichnis anlegen .....        | 1091       |
| Verzeichnis umbenennen .....     | 1091       |
| Vetorecht .....                  | 862        |
| Virtuelle Maschine .....         | 50         |
| Visage .....                     | 69         |
| Visual Age for Java .....        | 89         |
| void, Schlüsselwort .....        | 217        |
| Vorgegebener Konstruktor .....   | 515        |
| Vorzeichen, negatives .....      | 152        |
| Vorzeichenerweiterung .....      | 167        |
| Vorzeichenumkehr .....           | 158        |

**W**

|                               |      |
|-------------------------------|------|
| Wahrheitswert .....           | 135  |
| weave .....                   | 1259 |
| WEB .....                     | 1259 |
| Web-Applet .....              | 49   |
| WebRunner .....               | 49   |
| WEEK_OF_MONTH, Calendar ..... | 889  |
| WEEK_OF_YEAR, Calendar .....  | 889  |
| Weichzeichnen .....           | 1073 |
| Weißraum .....                | 353  |
| Wertebereich .....            | 491  |

Werte-Objekt ..... 720  
 Wertoperation ..... 153  
 Wertübergabe ..... 219–220  
 Wettkaufsituation ..... 963  
 while-Schleife ..... 193  
 WHITE, Color ..... 1082  
 Whitespace ..... 114  
 Widening conversion ..... 171  
 Widget ..... 1013  
 Wiederholungsfaktor ..... 411  
 Wiederverwendung per Copy & Paste ..... 1206  
 Wildcard ..... 821  
 Wildcard-Capture ..... 831  
 Win32-API ..... 91  
 windowClosed(), WindowListener ..... 1045  
 windowClosing(), WindowListener ..... 1045  
 WindowEvent, Klasse ..... 1043  
 WindowListener, Schnittstelle ..... 1040  
 Windows-1252 ..... 343  
 Windows-NT Konsole ..... 442  
 Windows-Registry ..... 931  
 Winkelfunktion ..... 1228  
 Wissenschaftliche Notation ..... 1216  
 Woche ..... 889  
 Woche des Monats ..... 889  
 Wohlgeformt ..... 1138  
 WORA ..... 68  
 Workbench, Eclipse ..... 100  
 Workspace ..... 95  
 World Wide Web ..... 48  
 World Wide Web Consortium (W3C) ..... 875  
 Wrapper-Klasse ..... 391, 718  
 Write once, run anywhere ..... 68  
 Writer, Klasse ..... 1120  
 Wurzelement ..... 1165

**X**

-Xbootclasspath ..... 895  
 Xerces ..... 1151  
 XHTML ..... 1148  
 XML ..... 1136  
 XMLOutputter, Klasse ..... 1163  
 -Xms ..... 1255  
 -Xmx ..... 1255

-Xnoclassgc ..... 1255  
 Xor ..... 165, 1070, 1201  
     bitweises ..... 168  
     logisches ..... 168  
 -Xprof ..... 1255  
 -Xrs ..... 1255  
 -Xss ..... 1255  
 -Xss:n ..... 237  
 -XX:ThreadStackSize=n ..... 237

**Y**


---

YEAR, Calendar ..... 889  
 yield(), Thread ..... 947  
 Yoda-Stil ..... 164

**Z**


---

Zahlenwert, Unicode-Zeichen ..... 115  
 Zehnersystem ..... 1203  
 Zeichen ..... 135, 149  
     Anhängen von ..... 404  
     ersetzen ..... 382  
 Zeichenkette ..... 125  
     konstante ..... 359  
     veränderbare ..... 397  
 Zeichenklassen ..... 411  
 Zeichenkodierung ..... 442  
 Zeiger ..... 57  
 Zeilenkommentar ..... 123  
 Zeilentrenner ..... 437  
 Zeilenumbruch ..... 905  
 Zeitgenauigkeit ..... 880  
 Zeitmessung ..... 910  
 Zeitzonенabweichung ..... 889  
 Zero-Assembler Project ..... 56  
 ZONE\_OFFSET, Calendar ..... 889  
 Zufallszahl ..... 1229, 1240  
 Zufallszahlengenerator ..... 1237  
 Zugriffsmethode ..... 491  
 Zustandsänderung ..... 871  
 Zuweisung ..... 154, 168  
 Zuweisung mit Operation ..... 168  
 Zweidimensionales Feld ..... 301  
 Zweierkomplement ..... 1202